Optimizing Reference Counting with Borrowing

Anton Felix Lorenzen

Geboren am 27. Juli 1998 in Hamburg

29th November 2021

Masterthesis in Computer Science

Betreuer: Daan Leijen, Microsoft Research

Zweitgutachter: Prof. Dr. Heiko Röglin

Computer Science

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Abstract

Reference counting is a technique of automatic memory management that frees memory as soon as the number of references pointing to it drops to zero. The *Perceus* algorithm can insert instructions to maintain reference counts at compile time such that programs are provably *garbage-free*: memory will be freed as soon as it becomes unused. In this thesis, we show how *borrowing* can improve performance by reducing the number of reference count instructions. The resulting programs are not garbage-free, but *frame-limited*: their peak memory usage is not more than a constant amount times the current number of stack frames higher than without borrowing.

Zusammenfassung

Referenzzählung ist eine Technik der automatischen Speicherverwaltung, die ein Objekt freigibt sobald die Zahl der Referenzen, die auf es zeigen, null ist. Der *Perceus* Algorithmus kann Instruktionen in ein Programm einfügen, die diese Zahl so aktualisieren, dass Programme beweisbar *garbage-free* sind: Speicher wird direkt freigegeben wenn er nicht mehr verwendet werden kann. In dieser Arbeit zeigen wir das *borrowing* durch Reduktion der Zahl der Instruktionen die Geschwindigkeit von Programmen steigern kann. Diese Programme sind nicht garbage-free, sondern *frame-limited*: ihre höchste Speichernutzung ist nicht mehr als ein konstanter Faktor mal die aktuelle Zahl an stack frames höher als ohne borrowing.

Contents

Chapter 1

Introduction

There are various techniques that take the chore of manually requesting and freeing the memory that a program uses from the programmer. Reference counting is one of the easiest ones: It simply maintains the number of references that refer to an object and frees it when this number becomes zero. It is possible to implement this without much compiler support (e.g. `C++`'s `shared_ptr` or Rusts `RC<T>`), but an implementation inside a compiler will be able to insert the instructions that increase or decrease the reference counts statically (in other words: at compile time) and optimize them further.

In September 2019 Sebastian Ullrich and Leonardo de Moura presented the paper "Counting Immutable Beans" [Ud19], which introduced their method for inserting static reference count instructions into programs compiled with the functional programming language Lean. They described this algorithm in the context of Lean's internal calculus and gave several optimization strategies:

1. *Reuse analysis* searches for the situation that the reference count of an object is decreased followed by the creation of a new object of similar size. Then it replaces these instructions by a reuse instruction, which, at runtime, checks if the first object can be freed, and in that case uses it directly to construct the second object. This avoids calling a memory allocator, which has to mark the old object as freed and search for space for the new object at significant runtime expense.

2. Incrementing and decrementing reference counts is quite expensive itself and sometimes unnecessary: While traversing a datastructure we will increase the reference count of every node we enter and decrease the reference count of every node we leave – but at the end the reference counts are as before. Here, *borrowing* helps: For a certain time, we do not update the reference counts of objects that a certain variable refers to. Unfortunately, this can increase the size of memory needed by the program and they present one benchmark where their *borrow inference* algorithm applied borrowing such that the performance became worse.

3. Still, incrementing and decrementing reference counts is expensive even with borrowing, since the reference counts of thread-shared objects need to be atomics. By

distinguishing between possibly thread-shared and all other objects we can use normal integers for the reference counts of most objects in the program.

Aside from a short remark at the end of chapter 2, we[1] will largely ignore multi-core programs in this work. Similarly, these techniques have analogues in previous work, which we will not go into.

This work was extended in November 2020 by Alex Reinking, Ningning Xie, Leonardo de Moura and Daan Leijen in the paper "Perceus: Garbage Free Reference Counting with Reuse" [Rei+21]. They presented the λ^1 calculus that characterized the ways that reference counting instructions can be correctly inserted such that no space leaks may occur. Further, they gave precise semantics for how the heap of a program would change during runtime and a new formalization of the algorithm for inserting reference count instructions in this framework. Importantly, this allowed them to prove that the algorithm is garbage-free: The only time that memory may be in the heap even though it is not used by the program anymore is directly before it will be freed by the algorithm!

As a result, the peak memory usage of benchmark programs written in their programming language Koka is the lowest of all implementations they consider. Furthermore, the performance is quite competitive as can be seen in the chart 1.1.

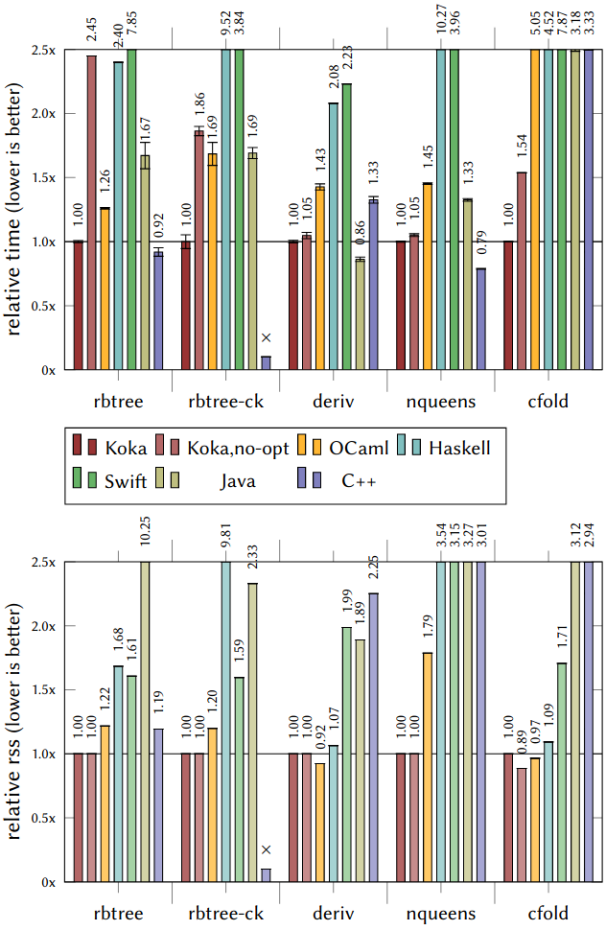

Figure 1.1: Relative execution time and peak working set with respect to the implementation of the Perceus algorithm in Koka (`no-opt` disables reuse analysis) [Rei+21].

They further show how programmers can write algorithms that take advantage of reuse analysis: It is possible to write recursive functions that look like pseudo-code but compile into efficient imperative loops that do not allocate any memory. They posit that this represents a paradigm shift and call it FBIP: functional but in-place.

[1]Since this thesis was written by a single author, the pronoun "we" should be understood as to refer to the shared intent of author and reader.

In this thesis, we will introduce briefly the theory of garbage collection, functional programming and programming language semantics. We will present further examples for the FBIP paradigm that increase both the readability and performance of widely studied algorithms. And, after a recapitulation of the Perceus algorithm, we will present an implementation of borrowing for it.

1.1 Notation

We will refer to several bytes in memory that "belong together" in the way the bytes of a struct belong together, as a *cell*, a *value* or an *object*. This terminology does not refer to any specific programming model (like object orientation). Such objects may contain *references*, pointers, to other objects and can be referred to by other objects or variables of the program. When an object cannot be accessed by the program anymore (because there is no path of references from a variable to it) we will call it *dead* or *unused* and else *live* or *used*. When discussing reference counting, we will assume that each such object stores a reference count. Then applying an `inc` or `dup` instruction to it will increase the reference count by one. Applying a `dec` or `drop` instruction will decrease the reference count by one and, if the reference count becomes zero this way, delete the object from memory. *Dupping* or *dropping* an object refers to this procedure.

1.2 Overview

The first part of this thesis approaches the topic from a practical standpoint.

- First, we give an overview over different memory management techniques and discuss the advantages and drawbacks of reference counting. This chapter is intended for an audience that is unfamiliar with reference counting and can safely be skipped.
- In the next chapter we dive into the Koka language and discuss the properties of functional programming that make reference counting especially effective in this paradigm. We then discuss some examples of static reference counts and reuse analysis.
- We conclude by discussing the design of datastructures in the FBIP paradigm. We present a short introduction to Zippers and then apply the visitor pattern to red-black trees and splay trees. We give benchmarks to show that they are competitive against the state of the art. Finally, we will discuss how better reuse behavior compares against higher memory consumption at the example of B-trees.

The second part discusses the formal aspects of borrowing.

- At first, we introduce the necessary calculi to reason about programs. This includes describing the internal representation of Koka and the methods for reasoning about its semantics: lambda calculus, evaluation contexts, sequent calculi, A-normalization. Then we describe the λ^1 calculus, Perceus algorithm and the heap semantics as in the paper by Reinking, Xie et al.

- We explore the design space around borrowing and introduce necessary conditions for borrowing. We describe the syntactical constructs and normalization to achieve these and then extend the λ^1 calculus. We describe Lean's borrowing inference, but give evidence that such an inference will not work well for Koka.
- Finally, we discuss a new notion that gives a reasonable bound on the space usage of transformed programs. This allows us to formally specify in which cases borrowing can be applied without worrying about potential increases in memory consumption.

1.3 Contributions

In this thesis we make the following contributions:

We extend the λ^1 calculus to allow borrowing and show that our formalization captures the essential properties of borrowing. In particular, we discuss how functions that use borrowing can be passed to higher-order functions, which was not covered explicitly in previous work. Based on this description, we provide an implementation of borrowing in Koka and show that it can improve performance on some benchmarks.

We develop further benchmarks for Koka based on datastructures for heaps and sets. We find that the FBIP paradigm applies to these algorithms and show that it can not only reduce memory consumption but also make the algorithms more elegant. We describe known techniques for deriving such algorithms and show that they are competitive against well-known textbook solutions.

While we require the programmer to choose when to apply borrowing, Ullrich and de Moura also give an inference that can make this choice automatically. We show that their inference can lead to sharp increases in memory usage which can negatively impact performance. We argue that a borrow inference should be *frame-limited*: In the worst case, it should only increase peak memory consumption by a constant factor times the number of stack frames. We give simple calculus for reasoning about frame-limited transformations and give rules that allow many interesting applications of borrowing while remaining frame-limited.

As part of the work on this thesis, but not described here, we also found that reuse analysis could be made significantly more powerful while not holding on to more memory than necessary for reuse. We proved, together with Daan Leijen, that reuse analysis is itself frame-limited [LL21].

1.4 Acknowledgements

I was very happy to write a thesis on an active topic in functional programming research and I am grateful to those who helped make this opportunity a reality: Daan Leijen who took a chance on a student he had no previous relationship with, Heiko Röglin who agreed to supervise a project outside his area of work, and the University of Bonn for accepting this thesis as part of a masters degree in Computer Science.

Further, I want to thank the proof readers who pointed out errors and helped make the prose more approachable: Joost Fischer, Jan Eube and Samantha Tröstrum.

Chapter 2

Memory management

Memory is not only necessary to do non-trivial computations, but it can also significantly speed up algorithms when used right. Yet, it is often scarce and slow to access and is therefore a resource that needs to be carefully managed. This chapter aims to give an overview of some techniques that are used today. We will not focus on completeness or an objective comparison between competing approaches in this treatment, but rather aim to give an intuition for the problems that memory management systems need to solve. We will illustrate this by four major paradigms: manual memory management, RAII, tracing garbage collectors and reference counting.

Yet, even manual memory management is not as manual as it sounds. First, a program usually does not have direct access to the physical memory of a computer. Even for security reasons alone it seems sensible to disallow this and so most operating system will assign each process its own memory regions to access. These memory regions are assigned in pages of 4kb to 16kb on todays systems[1]. The program can then access memory addresses relative to its assigned pages (called virtual memory), which the operating system maps to physical memory addresses (often with hardware support).

But most programmers do not want to deal with pages of size 4kb to 16kb but instead with custom sizes that can be smaller but also much bigger. Therefore, a memory allocator is usually used, that will request memory from the operating system and manage it. Thus calling `malloc` will cause the allocator to search its list of possible memory locations for free space, while calling `free` will return a region in memory to the list of free memory locations. There is a wide range of possible design choices for such an allocator and what choices are best for performance depends on the precise programming model. This is why Lean and Koka use their own memory allocator for the C backend [LBM19].

While allocating is simple enough for the programmer, freeing is much more difficult. Two main problems can occur here:

- When memory is freed too early and then accessed again, this will usually break the program. Either the memory region can still be accessed in which case the

[1] On Unix you can find out the page size by running `getconf PAGE_SIZE`.

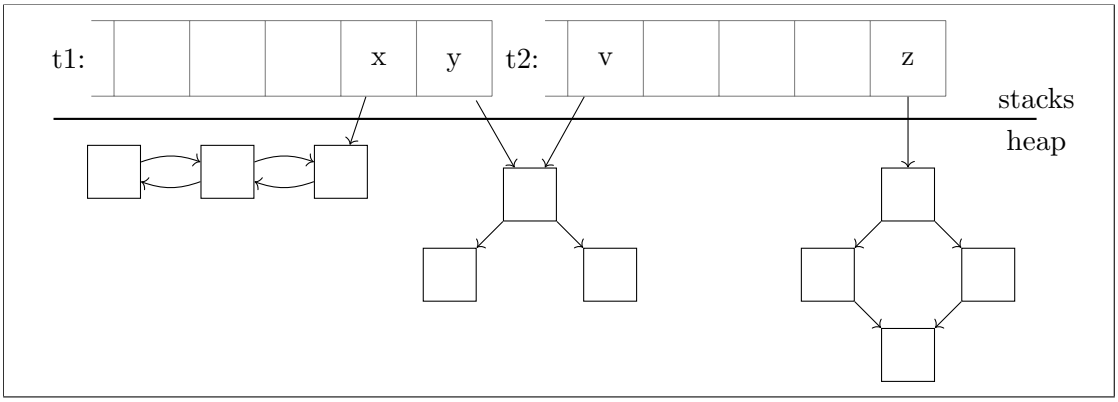

Figure 2.1: An allocator-managed heap used by two threads `t1`, `t2`

relevant data are likely to have been overwritten already with data used anywhere in the program. Or the allocator has returned the page to the operating system in which case the virtual address can not be mapped to a physical address causing a segmentation fault (SEGFAULT).

- When memory is freed way too late or never, this is called a *space leak*. This means that the operating system has assigned the program some memory which it can not access anymore because the allocator still has this memory region marked as used. While this can be acceptable sometimes if the amount of leaked space is small or the program runs only for a short time, it often carries a significant performance penalty: in the best case this is just caused by the additional overhead of asking the operating system for more memory. In the worst case, the computer may run out of RAM and proceed to swap some memory out to the disk which is several magnitudes slower.

The reason one of those two problems occurs is usually because data is *shared* between different parts of the program. The programmer might free the data even though it will later be used elsewhere or not free it under the mistaken assumption that it will be freed by some other procedure. This is best illustrated by thinking of the objects in memory as a graph: Each object forms a node and there is a directed edge from node a to node b if object a contains a reference to object b. The only objects that have no incoming edges in this graph are those stored in variables and dead, unused objects. Memory management would be easy if this graph was a forest of arborescences: Then, whenever a variable is last used, we could free the object it references and all its descendants.

But the graph is almost never a forest for three reasons:

- Several variables may refer to the same objects, either directly or through some descendants (see the tree referred to by both `y` and `v` in 2.1). This may happen when datastructures are shared between program parts that cooperate via this datastructure, i.e. a producer thread writing results into a queue where they are

read by a consumer thread. A further case where this happens is when looking up or inserting into a data structure that will be used later on: A variable points to the root of the datastructure while we will use another one to traverse it.

- There may be directed cycles in the graph. Imperative datastructures like graphs, doubly linked lists and binary search trees are often designed in such a way: See the doubly linked list of `x` in 2.1.
- There may be undirected cycles in the graph, like the diamond of `z` in 2.1. This can happen when some data is used at many places in a program (e.g. command line options that are immutable and stored in several places) or for performance reasons. A classic example of this are automatic theorem provers, which rewrite terms and share common subterms. Disallowing sharing would lead to an exponential increase in memory usage [SHM20].

A way to deal with the first reasons is by clarifying the *ownership* of objects. One of the incoming edges will get the ownership of the object and all the other incoming edges will *borrow* from that edge. Then we will free the object only when the object from which the owning edge started is freed. This often has to be applied by a programmer, since no borrowing edge may remain when the object is freed; else we might get a use-after-free error. But sometimes a compiler can do this: later we will see how traversals of datastructures can be borrowing by default.

A way to deal with the last two reason is by identifying the directed and undirected cycles in the graph. This requires deep knowledge about the behavior of the program and can thus only be done by the programmers – with significant time investment. Then custom logic can be written that frees these cycles and proxy objects can be introduced to stand for these cycles. In the terminology of our graph, we collapse these cycles and obtain a cycle-free graph that can be handled by more simple memory-management techniques. That is the approach usually taken by C and C++ programmers and we will describe it in more detail now.

2.1 Manual memory management & RAII

The C programming language gives the programmer access to the low-level facilities described in the last section, but almost no other tools. It is possible to group some bytes together in memory by a struct declaration and the size of this struct can be queried with `sizeof`. For example, to create an array of n complex numbers we could write:

```
typedef struct complex_s {
  int a; /* real */
  int b; /* imaginary */
} complex_t;

void foo() {
```

```
  ...
  complex_t *cs = (complex_t*) malloc(sizeof(complex_t) * n);
  ...
  free(cs);
};
```

This can feel liberating: Since C avoids the performance penalties associated with more automatic forms of garbage collection, it will perform extremely well in the hands of a competent programmer. More importantly, it is possible to observe and control the latencies introduced by calls to `malloc` and `free`, which is important for realtime applications like audio engines or games.

But it is also difficult to free all memory correctly and as such space leaks are not uncommon. Furthermore, a program may leak memory only in certain code paths (for example when an exception is thrown), which makes leaks hard to detect. And, somewhat surprisingly, an automatic memory management system can actually be faster than the free-logic written by a programmer if the programmer is not careful to free values shortly after they are last used [HB05].

In order to make memory management easier for programmers while essentially keeping the same trade-offs as C, C++ introduced the RAII paradigm, short for resource acquisition is initialization. The idea is that resources should be bound to an object of a class, which acquires the resource in its constructor and releases it in its destructor. When the constructor is called, an object is placed on the stack and once the program reaches the end of the current scope, the destructor of said object is called.

By using a memory region or other objects as the resource, this becomes an easy way to automatically insert the necessary calls to free. To reference our previous example:

```
class Complex {
  int a; // real
  int b; // imaginary
};

void foo() {
  ...
  std::vector<Complex> vec(n);
  ...
  bar();
  // vec freed automatically
}
```

However, notice how C++ compilers can not call the destructor when the object is last used, but only when it goes out of the lexical scope. As a result, memory allocated this way will stay alive for longer than necessary. Also, this can block tail call optimization from happening: While C++ compilers will not use a new stack frame for function calls that occur as the last instruction of a function, they will use one for the call to `bar` in

the function `foo` above. In order to achieve better code, it is therefore necessary to give a new lexical scope to `vec`:

```
void foo() {
  ...
  {
    std::vector<Complex> vec(n);
    ...
    // vec freed automatically
  }
  bar();
}
```

The RAII model of C++ has many advantages: it will run at essentially the same speed and latencies that manual memory management in C could achieve while being less error-prone. Could we use this model for a functional programming language like Koka? The answer appears to be no. An essential part of functional programming is the decision to make all values immutable[2]. As a result, functional datastructures are *persistent*: Their internal state will not change when an operation is called on them; instead these operations will return a modified copy that shares part of the internal state of the old datastructure. This is quite convenient for programmers since it is always possible to reason about a given value locally: it is not possible for a different thread or function to modify it. This can even be good for performance. For example:

```
set.insert(x);
foo(set);
set.remove(x);
```

can (for a persistent datastructure) be written as:

```
auto set1 = set.insert(x);
foo(set1);
// 'set' unchanged
```

thus saving the `remove` operation. But to make persistent datastructures work well, it is crucial that the internal state is shared. To illustrate this, imagine an insertion into a balanced binary search tree. While an imperative algorithm could modify the tree in place, a functional algorithm doesn't have this luxury as the old tree is immutable. Without sharing, an insertion must therefore use $O(n)$ time to copy the entire tree. With sharing, it is enough to create new nodes for the *spine*, the path that was taken to the new element in the tree, and the rebalanced nodes. Since all the other parts of the tree are shared with the old version, this insertion takes $O(\log n)$ time as desired.

[2] In Koka there are some exceptions to this rule, but they are not often used.

Nonetheless, some research has considered non-sharing memory management techniques for functional programming languages. Based on linear logic [Wad90] these can insert free-statements at the correct position and interface well with mutating, imperative algorithms [Carp21]. But in the context of this thesis we want to use persistent datastructures and thus need a more powerful version of memory management. We will first take a look at the predominant method of automatic memory management: tracing garbage collectors. Then we will investigate an alternative design, reference counting, in more detail.

2.2 Tracing garbage collectors

The idea behind tracing garbage collectors (GC) [McC60] is simple: An object is dead if it can not be used by the program anymore, in other words, if there is no path from any variable to this object in the memory graph. Thus, we can from time to time walk through memory starting with the current variables of the program, *mark* all objects that can be reached this way and then *sweep* through memory again linearly and free all the unmarked objects and delete the marks. Of course, a running program might change the memory graph and so a naive implementation of this needs to stop the program (a so-called *stop-the-world* collector). This requires no help from the programmer and will prevent all space leaks; but it is slow and every invocation of this procedure will lead to latencies, called *GC pauses*. These latencies will be relatively unpredictable because the garbage collector has to rely on heuristics on when to run: running too often will slow the program down unnecessarily while running too infrequently will lead to much memory lying around unused. Thankfully, there are some strategies that improve on a naive implementation:

A *concurrent* garbage collector can avoid to stop the world by using write barriers on the objects it is marking. This makes garbage collection itself slower, but since the GC can run on a different processor core, the program does not need to be stopped and can run with almost the same speed as before.

A *generational* tracing garbage collector ([LH83], [Moo84], [Ung84]) partitions the space of objects into several generations: Objects that have been allocated after the last GC phase and objects that have been live since a few phases. Heuristically, it is much more likely that newly created objects become unused than that old objects become unused. It is thus possible to run phases at different intervals for the different generations, which significantly reduces the GC pauses.

Finally, a *compacting* garbage collector [HW67] will also move the objects after sweeping the memory to close any free spaces in memory. This can even lead to better behavior than possible with manual memory management, since there it can happen that the space becomes fragmented with many free spaces that are too small to be used for new allocations. A compacting garbage collector will move the memory such that these spaces are filled and thus it can use a much simpler version of `malloc` and `free` than a C program: there is no need to track free spaces in memory and instead one can just assign each allocated object the next space in memory (called *arena allocation*). Unfortunately,

since the existing memory needs to be moved for this to work, this implementation of GC usually needs to stop the world.

Due to the simplicity for programmers and the remarkable optimizations outlined above this is currently the dominant paradigm of memory management used by imperative languages such as Java, C# and JavaScript (on V8) and functional programming languages such as OCaml and Haskell.

2.3 Reference counting

Reference counting was first implemented in Lisp [Col60] as a simple technique for keeping track of references. Reference counting stores an integer for every allocated object that denotes the number of references that point to this object. The integer is updated by the runtime and if it drops to zero, the object can be freed. This technique was quite popular for a while and languages like Objective-C, Swift and Python use it today. In general, reference counted programs will use significantly less memory than tracing garbage collected ones: it shows similar behavior as the RAII model of C++ while being more convenient.

By definition, reference counting can not clean up directed cycles between objects: The incoming edges of the cycle will keep the reference counts of the involved objects at at least one and so they (and any memory they reference) can not be cleaned up. Objective-C and Swift solve this problem by requiring the programmer to mark some pointers as *weak* such that they do not increase or decrease the reference counts. Python employs a separate tracing garbage collector to remove cycles. In (strict,) pure functional programming on the other hand it is impossible to create cycles and so Koka does not provide any further mechanism to deal with this[3]. We will discuss cycle-free datastructures in more detail in section 4.3.

Most implementations of reference counting decrement the reference counts only at the end of the current scope and not at the last use of an variable. This is easiest in practice, since it does not require special support for exceptions (we discuss this in more detail in section 3.4). For example, C++'s `shared_ptr` will decrement when the destructor is called at the end of the scope. Rust's `RC<T>` and Swift also show this behavior [Gal16]. But using the Perceus algorithm, Koka can generate reference count decrements precisely after the last use of a variable which can reduce the memory consumption drastically. We discuss this technique in detail in chapter 5.

Unfortunately, `dup` can not always be implemented as a simple `add` instruction. If the object can be accessed from several threads this can very easily lead to race conditions: Assume two threads both try to increase the reference count at the same time: read the current reference, add one to it and write it back. If the threads race the reference count would only be increased by one after this procedure and it is thus possible that the object

[3] In lazy functional programming languages (like Haskell) cycles can be created through a technique called *tying the knot*. In Koka, mutable references can be used to create cycles: these need to be broken up by the programmer.

is freed to early! Therefore, the reference counts need to stored as atomics when objects are thread-shared.

In imperative programming languages it is often not clear when an object is thread-shared or not. As a result, reference counts are often implemented as atomics by default at significant performance expense. There are several techniques to mitigate this: Deferred reference counting [DB76] keeps the values only referenced by variables in a special *zero-count-table* and tries to free them periodically. All other reference counts then only count the references to the cell from another cell (and not those from variables). This removes many reference count instructions but can lead to significantly higher memory usage. Alternatively one can use two reference counts: A non-atomic one for the 'owner' thread where the value was created and an atomic one for all other threads [CST18]. This *biased* reference counting works particularly well if most values are only used on one thread. Koka uses another technique first described for Lean [Ud19]: Any value is non-thread-shared by default and only a special operation to move this value to a new thread will make it thread-shared. This operation will then visit all the cells it references, set a flag at each cell and store the reference count in an atomic value in the same place.

Atomic operations are usually quite fast, provided that they are *uncontended*: the most common scenario where only one thread accesses the atomic at a time. As can be seen in figure 2.2, on modern Intel hardware Lean will incur a performance penalty of 20% to 130% when every single object is thread-shared (but only one thread is used, so all atomics are uncontended).

In this thesis we will describe another technique for reducing the cost of the reference count instructions: borrowed values are kept live by some other variable. This presents a form of space-safe deferred reference counting: We can avoid some unnecessary reference count instructions while obtaining better control over when values are freed.

	base	-*ST*
`binarytrees`	1.00̃	1.22̃
`deriv`	1.00̃	1.42
`const_fold`	1.00	1.23
`parser`	1.00	1.68
`qsort`	1.00	1.13
`rbmap`	1.00̃	1.71
`rbmap_10`	1.49	2.43
`rbmap_1`	4.72̃	8.02
`unionfind`	1.00̃	2.31

Figure 2.2: Benchmark performance of Lean vs the case where *all* reference counts are atomic. [Ud19]

2.4 A note on memory ordering

Modern CPUs often execute several statements at the same time and may change the order in which they access memory as long as this is not visible to the programmer *in the same thread*. Crucially, this need not hold for multi-threaded programs. This can lead to unexpected errors since CPUs show different behaviour: Intel X86 CPUs may only reorder loads with earlier stores to different locations [Int15], while ARM CPUs may reorder any loads and stores [Arm21]. Since atomics are often used to provide synchronization between different threads they can provide different guarantees regarding the ordering of instructions [cpp21]. When marked as `relaxed` they impose no constraints on the reordering of instructions which is the fastest option,

but may also prove counter-intuitive. For example, in Koka it is common that a *parent* value is dropped directly after a value it refers to (its *child*) is dupped. But if this is reordered and another thread drops the last reference to the parent value then the child may be freed before the dup is executed! Marked as `acquire` atomics guarantee that no later (atomic or non-atomic) loads will be moved before the atomic load. Similarly, as `release` they guarantee that no earlier (atomic or non-atomic) stores will be moved after the atomic store. Finally, `acq_rel` combines these two properties[4].

Reference counts are usually implemented in imperative programming languages with relaxed atomic increments for dups. Drops use a release operation to decrement the counter and check if the value is unique. If so, it is necessary to acquire the value before freeing. This ensures that all writes to the value are recorded before dropping it from a thread and that all information (for example regarding child references) is up-to-date while freeing [SBB21]. Since most Koka values are immutable[5] it seems like the acquire operation before freeing could be avoided. However, it is necessary to acquire so that the children (and most importantly their reference counts) are synchronized. But it would also be possible to avoid the release on the recursive drops on the children of a freed value, since all the stores after the initial release will be on values unique to the current thread (which we know since their reference count was one).

[4]The memory effects of a thread may also be propagated to other threads at different times leading to program executions that can not be linearized. Atomics can avoid this when marked as sequentially consistent (`seq_cst`).

[5]Reuse analysis may mutate values in-place but only when they are unique (and thus only accessible by one thread).

Chapter 3

A Short Tour of Koka

This thesis includes some code samples and we have chosen to write them in Koka, since our implementation of borrowing was done for that language. Koka is a *functional* programming language, which means that a typical Koka program will consist of some functions that operate on immutable datastructures. Datastructures are described in Koka by *algebraic data types* which can be thought of as a mixture of enums and structs. We will introduce these in the next section by example.

3.1 Algebraic data types

Algebraic data types (ADTs for short) can be used to define an enumeration of values; for example the type of booleans can be defined as an enum of two *constructors*: `True` and `False`.

```
type bool
  False
  True
```

An ADT can also be used with a single constructor and some arguments to define a struct. The arguments need to be given a explicit type, through the `name : type` syntax:

```
type complex
  Complex( real : int, imaginary : int )
```

Crucially, these two features can be combined: It is possible to define a type that carries certain arguments only for some of its constructors. This is especially useful for error messages: We can return a result if successful and a specific error message else.

```
type result
  Success( result : int )
  Error( msg : string )
```

This should be read as: "A result is either a success with an integer or an error with a message". This behavior can be simulated in C by defining an enum for the constructor *tags* `'Success'`, `'Error'` and a union of an int and a string, which is why this technique is also known as a *tagged union*. Unlike C, Koka maintains the invariant that the tag `Success` will never occur together will the `msg` string and vice versa. This way, it is possible to provide a `match` syntax:

```
match(compute-result())
  Success(result) -> ...
  Error(msg) -> ...
```

Unlike with a `switch` statement each of the *branches* of the match is independent from the other and no fall-through happens. Unlike with a `if` statement, the compiler can check at compile time that all possible scenarios are handled.

Another important difference to C is that `'msg'` and `'result'` are usually stored as pointers and not directly in the constructor. This allows us to store values of unknown size in constructors. For example, it is possible to parameterize a type over other types. We can use this to create an abstract type for errors and then derive the `result` type of above as an instance of this type:

```
type either<a, b>
  Left( left : a )
  Right( right : b )

alias result = either<int, string>
```

Furthermore, types can refer to themselves recursively. This can be used to create a simple version of the natural numbers:

```
type nat
  Zero
  Successor( n : nat )

val two = Successor( Successor( Zero ) )
```

This definition may remind a mathematically inclined reader of Peanos axioms of the natural numbers. In fact, the type theory that underlies Koka posits certain axioms about types that make the above definition a model of the Peano arithmetic:

- Any value of a type must be created by one of the constructors. Thus if `n : nat` then either `n = Zero` or there is some `m : nat` such that `n = Successor(m)`.
- Constructors are injective. In other words, we have `n = m` if and only if `Successor(n) = Successor(m)`.
- Different constructors are not equal, so for no `n : nat` we have `Zero = Successor(n)`.

In this sense, the `match` syntax gives a simple induction scheme for algebraic data types:

```
match(n)
  Zero -> ...           // handle case n = 0
  Successor(m) -> ... // handle case m -> m + 1 = n
```

For programmers it is often convenient to define a version of equality for some types that doesn't fulfill the above axioms. This is possible, however it should be stressed that such a user-defined notion of equality does not change the above-used *definitional equality* which describes which datatypes will be identical in memory.

We want to conclude this section by giving three ADTs that are common in functional programming. The `maybe` type can be used to simulate `NULL` pointers in a safe way:

```
type maybe<a>
  Nothing
  Just( value : a )
```

The additional safety comes through the fact that the only way to access the value contained in the `Just` constructor is by matching. However, when we match we also have to handle the case of the `Nothing` constructor. Thus, unlike in, say, C we can not forget to handle the `NULL` case.

ADTs are particularly elegant for datastructures that form a tree in memory (in the sense introduced in the last chapter). Among these are single linked lists and binary trees:

```
// The list [1,2,3] would be Cons(1, Cons( 2, Cons(3, Nil)))
type list<a>
  Nil
  Cons( head : a, tail : list<a> )

type tree<a>
  Tip
  Bin( elem : a, left : tree<a>, right : tree<a> )
```

Notice how both recursive definitions define a *base case*: `Nil` and `Tip`. While in most imperative languages this base case would be simulated by a null pointer, we need to be explicit here. In fact, if we left it out every list or tree would need to have infinite size!

It is of course also possible in Koka to deal with datastructures that do not have a tree structure in memory like double-linked lists or arrays, but we will largely ignore these features in this thesis. Instead we will, in section 4.3, discuss a technique that allows us to avoid using cyclic datastructures in some cases.

3.2 Functions on ADTs

Let us describe a few functions on ADTs to get used to the `match` statement. To extract the first element (or *head*) of a list, we can write:

```
fun head(xs : list<a>) : maybe<a>
  match(xs)
    Cons(x, _) -> Just(x)
    Nil        -> Nothing
```

Here, the `maybe<a>` in the first line is the *return type* of the function. Notice how this definition works for lists of any type of element: The compiler will *instantiate* `a` with the correct type when we apply `head` on a list of integers, trees, etc..

Lets move on to more interesting functions that call themselves recursively. We start by computing the length of a list:

```
fun length(xs : list<a>) : int
  match(xs)
    Cons(_, xx) -> 1 + length(xx)
    Nil         -> 0
```

Notice how this function gives precisely the inductive definition of the length of a list while being completely safe from `NULL` pointer errors. Unfortunately, it is not terribly fast: The runtime has to allocate a new stack frame for each recursive call.

However, we can do better. If the recursive function call is the last statement in a branch this is called *tail-recursion*. Tail-recursion can be compiled into a goto by the compiler and is therefore just as fast as an imperative for-loop.

```
fun length(xs : list<a>) : int
  fun go(xs : list<a>, acc : int) : int
    match(xs)
      Cons(_, xx) -> go(xx, acc + 1)
      Nil         -> acc
  go(xs, 0)
```

Here we have defined another function `go` (inside of `length` so that `go` can only be called by it), which is tail-recursive. The strategy of carrying an accumulator `acc` with a function so that the addition can be done before the function call is well-known and can often be applied for other associative operations as well.

It is also possible to apply a similar optimization, when the recursive call is only followed by a constructor application. We might expect a length function that returns the ADT-version of natural numbers to not be tail-recursive:

```
fun length(xs : list<a>) : nat
  match(xs)
    Cons(_, xx) -> Successor( length(xx) )
    Nil         -> Zero
```

While this function is clearly not tail-recursive, it is tail-recursive *modulo* a constructor (TRMC) [FW75]. This suffices to compile this recursive call into a goto-statement and so this version will also not allocate any stack frames. We will see further applications of this technique in the next chapter.

3.3 Higher-order functions

The strength of functional programming comes from being able to use functions as first-class objects. That means we can store functions in variables and pass them to other functions. Let us thus define a last version of our `length` function:

```
fun foldl(xs : list<a>, acc : b, f : (b, a) -> b) : b
  match(xs)
    Cons(x,xx) -> foldl(xx, f(acc, x), f)
    Nil        -> acc

fun length(xs : list<a>) : int
  foldl(xs, 0, fn(acc, x) -> acc + 1)
```

The `foldl` function folds a function from the left: It applies the given function to the accumulator and the first element, then the new accumulator and the second element and so forth until it reaches the end of the list. Due to its generality, we can easily use it for other purposes: for example to compute the sum of the elements of a list or ask whether a list of booleans contains only the value `True`:

```
fun sum(xs : list<int>) : int
  foldl(xs, 0, fn(acc, x) -> acc + x)

fun all(xs : list<bool>) : bool
  foldl(xs, True, fn(acc, x) -> acc && x)
```

In a similar spirit, we can also fold from right, but then we can not make use of tail-recursion anymore:

```
fun foldr(xs : list<a>, acc : b, f : (a, b) -> b) : b
  match(xs)
    Cons(x, xx) -> f(x, foldr(xx, acc, f))
    Nil         -> acc
```

Let's end our sequence of toy programs by considering how to change the elements of a list. A first idea for, say, increasing each element of a list of integers by one might be:

```
fun incr-one(xs : list<int>) : list<int>
  match(xs)
    Cons(x,xx) -> Cons(x + 1, incr-one(xx))
    Nil        -> Nil
```

The reader may have expected some form of assignment operator to change the list directly. Instead it appears that this definition is creating a new list! The reason for that is that in functional programming all values are considered immutable, so it is impossible to change the old list. Immutability is not necessary for either ADTs (as Rust demonstrates) or higher-order functions (as, say, Python demonstrates), but it can

make the organization of larger programs nicer. Still, it comes at the inefficiency that we potentially have to free a `Cons` while allocating a new one directly after. This is what Sebastian Ullrich and Leonardo de Moura [Ud19] called the *resurrection hypothesis*: Data will be freed shortly before it is allocated anew. We will discuss a compiler optimization for this soon, but let us first generalize the definition of `incr-one` by using a fold:

```
fun incr-one(xs : list<int>) : list<int>
  foldr(xs, Nil, fn(x, acc) -> Cons(x + 1, acc))
```

However, this definition is not tail-recursive (since `foldr` is not tail-recursive). The reader may want to pause for a moment to consider why we can not use `foldl` (directly) in this scenario. Also, this will obscure the fact that the resurrection hypothesis holds in this case from the compiler. Instead we will thus consider a different generalization:

```
fun map(xs : list<a>, f : a -> b) : list<b>
  match(xs)
    Cons(x,xx) -> Cons(f(x), map(xx,f))
    Nil        -> Nil

fun incr-one(xs : list<int>) : list<int>
  map(xs, fn(x) -> x + 1)
```

Note how this definition of `map` can make use of TRMC again and is thus as fast as a for-loop. Still, it has to construct a new list to return. Since functions such as map are very commonly used, this puts a lot of pressure on the garbage collector. We will shortly see how Koka can avoid these allocations when the list is not used anywhere else. But before, we have to discuss reference counting in Koka.

3.4 Static reference count instructions

Koka can insert reference count instructions statically into a program using the Perceus algorithm. This information is only used for compiling and can not be written directly in a Koka program by the user — but we will do that anyway to make the subsequent optimizations clearer. In other words, we work in a hypothetical version of Koka that corresponds to the way the real Koka transforms a program.

We write `dup` for incrementing the reference count and `drop` for the operation that
- decrements the reference count if it is bigger than one
- and else, frees the object and call itself recursively on the fields of the object.

The `map` example then becomes:

```
fun map(xs : list<a>, f : a -> b) : list<b>
  match(xs)
    Cons(x,xx) ->
      dup(x); dup(xx); drop(xs); dup(f);
      Cons(f(x), map(xx,f))
    Nil -> drop(f); drop(xs); Nil
```

What happens here? It helps to think of the references that are passed around explicitly: At the start of the function we have two references (`xs` and `f`). In the `Cons` case we give variable names to the fields of the constructor and thus acquire two more references (`x` and `xx`). For these two references we have to increase the counters so we insert two `dup` statements. Then we do not need the reference to `xs` anymore so we get rid of it. By convention, Koka will need a reference to a function to apply it and so we `dup f`. Finally, the reference to `x` is passed to `f` and the references to `f` and `xx` to the recursive call of `map`. We end up holding no more references. Similarly, in the `Nil` case we will need neither `f` nor `xs` so we need to drop the references.

We describe Perceus in more detail in chapter 5. Here, we want to focus only on some consequences. First, we drop values (like `xs` above) directly when they become unused. An example from [Rei+21] illustrates how this can reduce the peak memory usage of a program:

```
fun foo()
  val xs = list(1,1000000)              // create large list
  val ys = incr-one(xs)                 // increment elements
  print(ys)
```

Here, `incr-one` will drop the list `xs` as it mapping over it. Furthermore, it will allocate the nodes of the list `ys` during the same process so that the program will only need enough memory to hold *one* list of integers at a time. This stands in contrast to memory management approaches that remove objects from memory based on lexical scope (like Swifts reference counting [Gal16] or C++s RAII-based `shared_ptr`) which might only free `xs` after `ys` has already been fully allocated. Similarly, a tracing garbage collector might not clean up `xs` in time unless it runs very frequently.

Second, we need to consider why this technique works at all. Consider the following program:

```
fun foo()
  val xs = ... // some object
  ...
  bar() // may throw an exception
  ...
  drop(xs)
  ...
  baz() // may also throw an exception
```

When an exception occurs in a function and is not handled, most programming languages unwind the stack (that is they free all objects on the stack and return to the calling function). However, here depending on where the exception occurs `xs` either needs to be freed or not. While this can be accounted for (eg. by setting a flag for `xs` on the stack when it is dropped), it still presents a considerable complication. Thankfully though, we do not have to worry about scenarios like these in Koka, because all side-effects

(reading files, printing to the console, exceptions, mutable variables and async/await) are encapsulated by *effects* and compiled into regular, non-side-effecting code [PP03; PP09; XL21]. This means that the control flow is *linear*: It can be fully represented in the syntax of match-statements and recursive calls introduced in the last sections. In particular, early exits (like `longjmp`) will be transformed into regular match statements.

Thirdly, since we drop objects as soon as they are not used anymore, we can find an elegant solution to the problem posed by the resurrection hypothesis. We will discuss this and related optimizations next.

3.5 Reuse analysis

Let us define a new instruction `drop-reuse`, which

- if the reference count is bigger than one, decrements it and returns a `NULL` pointer
- and else, calls `drop` on the fields of the object and returns the object.

The difference between this instruction and `drop` is just that this one will return the object instead of freeing it. We can then use the space in memory of this object for a new one. For the `map` example, we can write this like this:

```
fun map(xs : list<a>, f : a -> b) : list<b>
  match(xs)
    Cons(x,xx) ->
      dup(x); dup(xx);
      val ru = drop-reuse(xs); dup(f);
      ru@Cons(f(x), map(xx,f))
    Nil -> drop(f); drop(xs); Nil
```

We have written `ru@Cons` to mean that `Cons` should be written into the memory cell `ru` if possible. On the other hand, if `ru` is a `NULL` pointer, we will allocate a new cell for `Cons`. This solves the problem posed by the resurrection hypothesis: Instead of freeing an object and then allocating a similar one shortly after, we will simply *reuse* it.

In the most common case that the list we are mapping over is unique, the code above behaves exactly like a mutating imperative algorithm: No new space will be allocated. If the list is not unique, we will still create a copy of the list — which is then unique and can be reused if we map over it again. If only part of the list is unique and, say, the last half is shared the algorithm will adopt to this and mutate the first half in place while constructing a copy of the last half.

For this it is not necessary that the reused constructor is the same as the one on which the reuse token is applied: This can be done whenever the constructors have the same size and number of arguments[1]. We will use this insight extensively in the next chapter.

[1] To be more precise: the same number of arguments that are *represented as pointers*. Koka may store values (eg. integers) directly in the object of the constructor.

3.6 Drop specialization

Unfortunately, `dup` and `drop` are not cheap instructions: among other things they contain complex logic for dealing with thread-shared objects. Thus we want to consider two optimizations that can decrease the number of these instructions. For the first we observe what happens when we match on a unique value `v`:

- we call `dup` on the fields for which we defined variables.
- we drop all fields of `v`
- we free `v`

Here we duplicated work: We first `dup`'d some fields of `v` and then `drop`'d them immediately afterwards. With drop specialization, we will instead replace the involved `dup` and `drop` instructions by

```
if(is-unique(v))
  then drop unused fields of v; free(v)
  else dup used fields of v; drop(v)
```

Applied to our running example, the `map` function:

```
fun map(xs : list<a>, f : a -> b) : list<b>
  match(xs)
    Cons(x,xx) ->
      if(is-unique(xs))
        then free(xs);
        else dup(x); dup(xx); drop(xs);
      dup(f);
      Cons(f(x), map(xx,f))
    Nil -> drop(f); drop(xs); Nil
```

We can also integrate this pattern with the reuse analysis described above:

```
let ru = if(is-unique(v))
  then drop unused fields of v; &v
  else dup used fields of v; drop(v); NULL
```

Remark 3.1. This example also appears in [Rei+21], but they present it differently: They give a pseudo-code definition of `drop` as

```
fun drop(x)
  if(is-unique(x))
    then drop children of x; free(x)
    else decref(x)
```

Here `decref` is a new instruction that only decreases the reference count while not checking if the object needs to be freed. Drop specialization then just means inlining `drop` and moving the `dup`s on the children into the branches of the if-statement. Since a `dup`-`drop`-pair cancels out, we arrive at the above formulation.

Unfortunately, this is not thread-safe: It is possible for the reference count not to be unique during the check `is-unique(x)` (because another thread holds onto it) while being unique at the call to `decref(x)` (because the other thread dropped it meanwhile). It is possible to avoid this problem by defining `decref` in a thread-safe way (like Koka does), but then one loses the analogy to inlining the `drop` function.

3.7 Borrowing

Our second technique for reducing the amount of reference count instructions (and the main topic of this thesis) is best illustrated by a function that traverses a datastructure that we expect to remain used afterwards. We shall illustrate it here with the `length` function considered before:

```
fun length(xs : list<a>) : int
  fun go(xs : list<a>, acc : int) : int
    match(xs)
      Cons(_, xx) ->
        dup(xx); drop(xs); dup(acc);
        go(xx, acc + 1)
      Nil -> drop(xs); acc
  go(xs, 0)
```

Let us consider the case where we compute the length of a list which we will still use later. We then pass a new reference to the list to the `length` function, which will pass it to the `go` function. Then `go` will recursively make a new reference of the next list element, drop its current reference and call itself recursively on the new reference. In effect, we will create new references and then drop them in the next recursive call – to no lasting effect, similar to a telescope sum.

Note though that this is only a form of a telescope sum, if we hold on to a reference of the list: Else the `go` function will deallocate the list elements as it traverses the list. Our first step to removing the telescope sum is thus to require the callsite of `go` to hold on to a reference, in other words, we mark the parameter `xs` as *borrowed* (`^xs`). The length function then becomes:

```
fun length(xs : list<a>) : int
  fun go(^xs : list<a>, acc : int) : int
    match(xs)
      Cons(_, xx) ->
        dup(acc);
        go(xx, acc + 1)
      Nil -> acc
  go(xs, 0)
  drop(xs);
```

Borrowing has the potential to make a program 10% faster, as we will see later – but it also has clear drawbacks: Since we remove the `drop`s we can not use reuse analysis with it (and reuse analysis is much more important for performance than borrowing). Furthermore, we will not drop values as soon as possible. To illustrate this problem, let us consider the peak memory usage again:

```
fun make-tree(xs)
  match(xs)
    Cons(x,xx) -> Bin(x, Tip, make-tree(xx))
    Nil -> Tip

fun foo()
  val xs = list(1,1000000)              // create large list
  val ys = make-tree(xs)                // turn it into a tree
  // drop(xs); if xs is borrowed
  print(ys)
```

Here, one might want to make the `xs` of `make-tree` borrowed (and the inference of [Ud19] would mark it as such). But then the peak heap usage of this program is twice what it needs to be and we loose a core advantage of Perceus! We discuss this problem in more detail in the last part of this thesis.

Chapter 4

Link-inverted datastructures

In this chapter we will discuss datastructures using *link-inversion* since these fit in particularly well with the reuse analysis described in the last chapter. Link-inversion is a technique to avoid extra stack-space for recursions that are too complicated to be optimized away by tail-recursion-modulo-cons. Instead one modifies values on the heap, which in functional programming would usually imply new allocations. But with reuse analysis we can keep the elegance of the functional implementation while achieving the same efficiency as mutating implementation. We will first discuss some different approaches for deriving link-inverted algorithms and then propose new implementations for various balanced binary trees that improve on commonly used functional versions of these algorithms.

4.1 Binary trees

The binary-trees benchmark [Con21] is a commonly applied test for checking how efficient the garbage collector of a programming language is. It is simple to implement: First some binary trees with no shared subtrees of a given depth are allocated[1].

```
type tree
  Tip
  Node(l : tree, r : tree)

fun make-rec( depth : int ) : div tree
  if depth > 0
    then Node( make-rec(depth - 1), make-rec(depth - 1) )
    else Node( Tip, Tip )
```

Then we traverse these trees and count the number of nodes:

[1]Since Koka cannot determine if the function terminates, we have to add the `div` effect to its return type. This can be safely ignored.

```
fun checkr( t : tree ) : div int
  match t
    Node(l,r) -> l.checkr + r.checkr + 1
    Tip       -> 0
```

Both these definitions recurse twice, but Koka can optimize only one of the calls in `make-rec` away using TRMC and none of the recursive calls to `checkr`. However, there is a simple technique to transform the stack frames in both cases into heap allocations [SF98]. First, we make the functions tail-recursive by taking the rest of the computation and passing it to the first call as a callback function:

```
fun make-cps( depth : int, f : tree -> div tree ) : div tree
  if depth > 0
    then make-cps(depth - 1, fn(t) {
      make-cps(depth - 1, fn(t2) {
        f(Node(t, t2)) }) })
    else f(Node( Tip, Tip ))
```

The function `f` stores the computation that should happen after the call to `make-cps`. When we call the function we would supply for `f` the identity function. This technique (called CPS-transform) can thus be applied to move memory allocations from the stack to the heap: Instead of stack frames we allocate closures. But function closures are less efficient than datatypes and most importantly can not be reused. We can replace the closures by normal datatypes if we store the free variables: `depth, f` for the first closure and `t, f` for the second closure. We then replace the function `f` by a reference (`up`) to our datatype:

```
type builder
  Top
  BuildRight( depth : int, up : builder )
  BuildNode( left : tree, up : builder )
```

Here, `Top` corresponds to the identity function, `BuildRight` to the first closure and `BuildNode` to the second closure. This representation is called the *defunctionalization* of `make-cps`. We then add another function `make-up` that takes the datatype that represented a closure and executes the code that that closure used to execute.

```
fun make-down( depth : int, builder : builder ) : div tree
  if depth > 0
    then make-down( depth - 1, BuildRight(depth - 1, builder))
    else make-up( Node(Tip,Tip), builder)

fun make-up( t : tree, builder : builder ) : div tree
  match builder
    BuildRight(depth, up)
```

```
      -> make-down( depth, BuildNode(t, up))
    BuildNode(l, up) -> make-up( Node(l, t), up)
    Top -> t
```

This code is significantly more complex than the code we started out with, but it has the advantage that it doesn't use stack space. The recursive calls to `make-up` and `make-down` are in tail position and can thus be replaced by jumps.[2] Furthermore, it doesn't use more heapspace: In `make-up` the `builder` is either getting reused for a `BuildNode` (in the first branch) or a `Node` (in the second branch). Only the space allocated for the `Top` builder will be lost (and in practice, Koka represents `Top` as a form of `NULL` pointer in memory).

We can apply the same transformation to the `checkr` function.

```
fun checkc( t : tree, f : int -> div int ) : div int
  match t
    Node(l,r) -> checkc(l, fn(i) {
      checkc(r, fn(i2) {
        f(i + i2 + 1) }) })
    Tip       -> f(0)
```

However, we can obtain better code if we first apply the accumulator trick to reduce the number of closures:

```
fun checkt( t : tree, acc : int ) : div int
  match t
    Node(l,r) -> checkt(l, checkt(r, acc + 1))
    Tip       -> acc

fun checktc(t : tree, acc : int, f : int -> div int) : div int
  match t
    Node(l,r) -> checktc(r, acc + 1, fn(i) {
      checktc(l, i, f) })
    Tip       -> f(acc)
```

Unlike `checkc`, the `checktc` function only allocates a single closure. This corresponds to the tail-call optimization that can happen in the `checkt` function. We can again replace the closures by a datatype.[3]

```
type visit
  Done
  NodeR( right : tree, v : visit )
```

[2] As of this writing, Koka does not optimize mutually recursive tailcalls and we perform the optimization by hand. However, we expect Koka to acquire this optimization technique in the future.

[3] This example was contributed by Daan Leijen.

```
fun checkv( t : tree, v : visit, acc : int ) : div int
  match t
    Node(l,r) -> checkv( l, NodeR(r,v), acc.inc)
    Tip       -> match v
      NodeR(r,v') -> checkv( r, v', acc)
      Done        -> acc
```

Notice however, that the `checkv` function can only reuse the `Node` if the tree passed to it is unique. In the binarytrees benchmark we consider this is always the case, but in general this transformation might make code slower since it only moves the stack frames onto the heap in the form of `visit` values (and memory management is generally slower on the heap than on the stack).

In figure 4.1 we compare the performance of our different implementations against the official entries on the binarytrees benchmark. For the C++ implementation we selected the best performing entry on our system; keep in mind that the submitted programs where not optimized for the M1 processor. However, these results show that Koka works quite well and is among the fastest programming languages. Koka uses significantly more memory than other implementations due to the chosen parallelization strategy.

While using the `checkv` function makes the program faster, the `make-up` function is comparable to the `make-rec` function. We conjecture that this is because the TRMC optimization that applies to `make-rec` does not transfer to `make-up`: In `make-rec` we visit every node twice (thanks to TRMC) but in `make-cps` and `make-up` we visit every node three times. In contrast, the introduction of tail-recursion in `checkt` transferred over effortlessly to `checkv` where we could reduce the number of closures by one. In fact, if we do not apply the accumulator trick, we obtain a significantly worse performing program (see 'visit (no acc)'). This seems to be an interesting case for studying TRMC further.

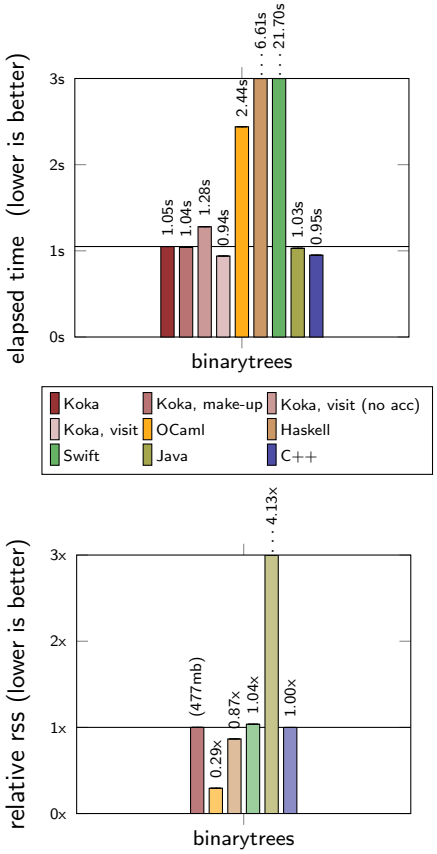

Figure 4.1: Benchmark results for binarytrees, Apple M1

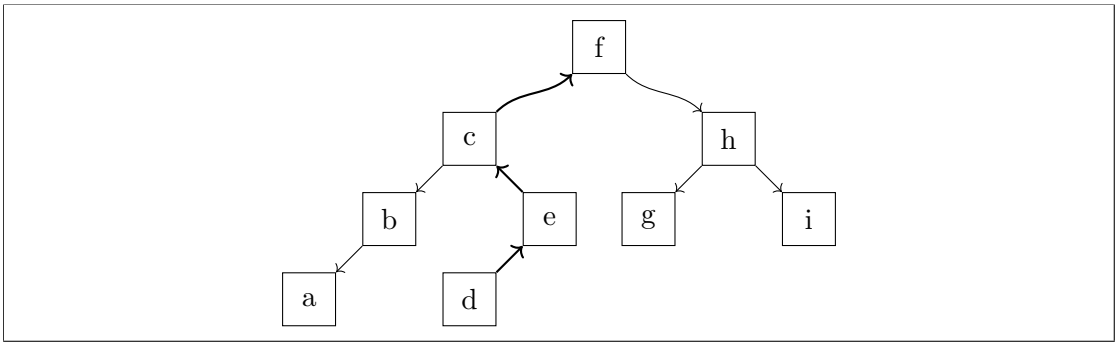

Figure 4.2: A binary tree with link-inversion on the path to `d`

4.2 Link inversion and the Zipper

We saw above how the nodes of the tree could be reused for the constructors of a `visit` datatype. In an imperative setting, we might want to perform this transformation explicitly and replace the pointer to the left or right subtree in a `Node` by a pointer to its parent. This technique is known as *link-inversion* since it inverts the pointer from parent to child (see figure 4.2).

It was first discovered independently by Deutsch and by Schorr and Waite [SW67] in the context of marking objects during garbage collection. Sobel and Friedman [SF98] later extended it to any anamorphism (a recursive function building up a datastructure). Reinking, Xie, de Moura and Leijen [Rei+21] describe how it interacts with reuse analysis. However, applying this transformation to more complicated tree-based algorithms is quite difficult and so it was often only applied to simple in-order traversals. But can it be used for more practical matters, like balancing binary trees?

To answer this question, we abstract from the technique of the last section: Datatypes like `builder` and `visit` above are known as *zippers* [Hue97]. In the general case (when we can't perform an optimization like the accumulator-trick), we can compute their shape based only on the shape of the tree we recurse on without considering a concrete algorithm. To state this description concisely we will denote datatypes algebraicly:

- The shape of a datatype is the *sum* of the shapes of its constructors
- The shape of a constructor is the *product* of the shapes of its fields. If a constructor contains no fields, we write it as 1.

This definition is recursive if a field contains the datatype itself. In this case, we write $\mu x. \ldots x \ldots$. The datatypes we have seen in section 3.1 have the following algebraic representation:

$$
\begin{aligned}
\text{bool} &\equiv 1 + 1 \\
\text{complex} &\equiv \text{int} \cdot \text{int} \\
\text{result} &\equiv \text{int} + \text{string} \\
\text{either} &\equiv a + b \\
\text{nat} &\equiv \mu x.\, 1 + x \\
\text{maybe} &\equiv 1 + a \\
\text{list} &\equiv \mu x.\, 1 + a \cdot x \\
\text{tree (in 3.1)} &\equiv \mu x.\, 1 + a \cdot x^2 \\
\text{tree (in 4.1)} &\equiv \mu x.\, 1 + x^2
\end{aligned}
$$

This representation corresponds to the number of different elements (also called *inhabitants*) that have a specific type (at least when this number is finite): Inhabitants are constructed using exactly one constructor and may contain any selection of elements from the fields.[4] The zipper is now the datatype that replaces any x by a parent pointer p and duplicates the terms such that it is clear which path we took. For example, we can obtain the zipper for the binary tree above by writing $\mu p.\, 1 + p \cdot x + x \cdot p$. If we want to write an algorithm that builds up the tree from left to right like in `make-rec` we will replace the x we have not visited yet by the data we need to build the tree. We saw this in the last section: The `builder` zipper is $\mu p.\, 1 + p \cdot \text{int} + x \cdot p$.

Instead of keeping the parent pointer in the zipper directly, we could also use a list of zipper values. The `builder` would then be:

```
type builder-list
  BuildListRight( depth : int )
  BuildListNode( left : tree )
```

The reader should convince themselves at this point that a list of `builder-lists` is the same thing as a `builder`. However, we can also see this algebraicly:

$$
\begin{aligned}
&\mu x.\, 1 + (\text{int} + \text{tree}) \cdot x \\
&= \mu x.\, 1 + \text{int} \cdot x + \text{tree} \cdot x \\
&= \mu x.\, 1 + x \cdot \text{int} + \text{tree} \cdot x \\
&= \mu p.\, 1 + p \cdot \text{int} + \text{tree} \cdot p
\end{aligned}
$$

As we saw above, the `builder-list` is just a specialized version of the general zipper $\mu p.\, 1 + p \cdot x + x \cdot p$ for trees x. Its listified version is then $x + x$. This version can be computed by replacing any constructor with n recursive references (containing

[4] A function from a to b is written as b^a in this framework, again alluding to the size of the set of the inhabitants. Datatypes that do not contain functions are also called *polynomial* datatypes.

x^n as a factor) by n copies of this constructor with $n-1$ references to x each $(n \cdot x^{n-1})$. During this procedure we will keep other fields (factors), but discard constructors that contain no recursive references. In other words, the zipper of a type is just the list of its derivative [McB01].

4.3 Avoiding cycles with Zippers

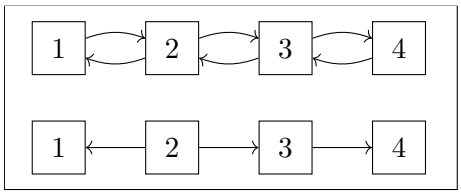

Figure 4.3: A doubly-linked list and its zipper centered at 2

We have promised to revisit the issue of cyclic datastructures and want to tackle it here. You may have recognized the list type in the last chapter as a *single-linked list* as opposed to a *double-linked list* where every `Cons` node would also have a pointer to its predecessor. Imperative programmers use the latter design to be able to construct their datastructures once and then move around freely in them; climbing around in the web of pointers like a spider searching for its prey. Functional programmers take a different view: They always stay at the same point and it is the web that moves around them. For a double-linked list the zipper is a pair of single linked lists: on the one side the elements to our right and on the other side the elements to the left of us.

```
type zipper<a>
  Empty
  At( l : list<a>, here : a, r : list<a> )

fun move-right(xs : zipper<a>) : zipper<a>
  match(xs)
    Empty -> Empty
    At(l, h, Nil) -> At(l, h, Nil)
    At(l, h, Cons(h', r)) -> At(Cons(h, l), h', r)
```

This has the drawback that it is not possible to jump around in the structure: We have to move to the position we want to see from the position we last visited. In particular, we can not keep a pointer to the first element, change a few elements in the middle and then return the first element without explicitly going back.[5].

However, we can now work with one pointer less per list cell. Since otherwise the list cells only contain one pointer to their element, this makes a big difference in memory usage. In the next two sections we will apply the zipper as a *visitor transformation* to trees to obtain versions of well-known balanced binary tree algorithms. Similarly to the simple zipper, we will invert the pointers from parents to children along the search path while going down and while going up balance and repair the inverted pointers. We will

[5]This seems to be one of the main problems when providing a purely functional implementation of graphs such as *algebraic graphs* [Mok17]

see that these algorithms are significantly faster than the algorithms usually taught to functional programmers and competitive with imperative implementations.

4.4 Splay trees

Splay trees [ST85] are self-balancing binary trees where the least recently accessed element is *splayed* to the top of the tree (see figure 4.4). We will cover them here for the elegance of our `splay` function that corresponds directly to the textbook definition (unlike imperative implementations that are often much longer and more complicated).

```
alias elem = int

type tree
  Node(left : tree, here : elem, right : tree)
  Leaf

type zipper
  Root
  LeftChild( parent : elem, up : zipper, right : tree )
  RightChild(parent : elem, up : zipper, left  : tree )
```

For simplicity, we store integers as elements in our splay tree. Note how the constructors of the zipper have the same number of arguments as the splay tree. To insert an element into the tree, we will first compare if the element should go into the left or the right branch (we will use our tree as a multi-set and so we do not care if the element is already contained in the tree). We then go down the chosen branch and create a zipper that holds the other branch. Since the sizes match, reuse analysis can turn `Node`s into `Left/RightChild`s during insertion:

```
fun insert(v : elem, tree : tree)
  fun go(tree, zipper)
    match(tree)
      Leaf -> splay(Leaf, v, Leaf, zipper)
      Node(a, b, c) ->
        if(v < b) then go(a, LeftChild(b, zipper, c))
        else go(c, RightChild(b, zipper, a))
  go(tree, Root)
```

Once we have reached the bottom of the tree, we go up the tree again and perform the splay rotations. For the sake of brevity, we have chosen very short variable names, but we invite the reader to check the individual cases against figure 4.4:

```
fun splay(e, f, g, zipper)
  match(zipper)
```

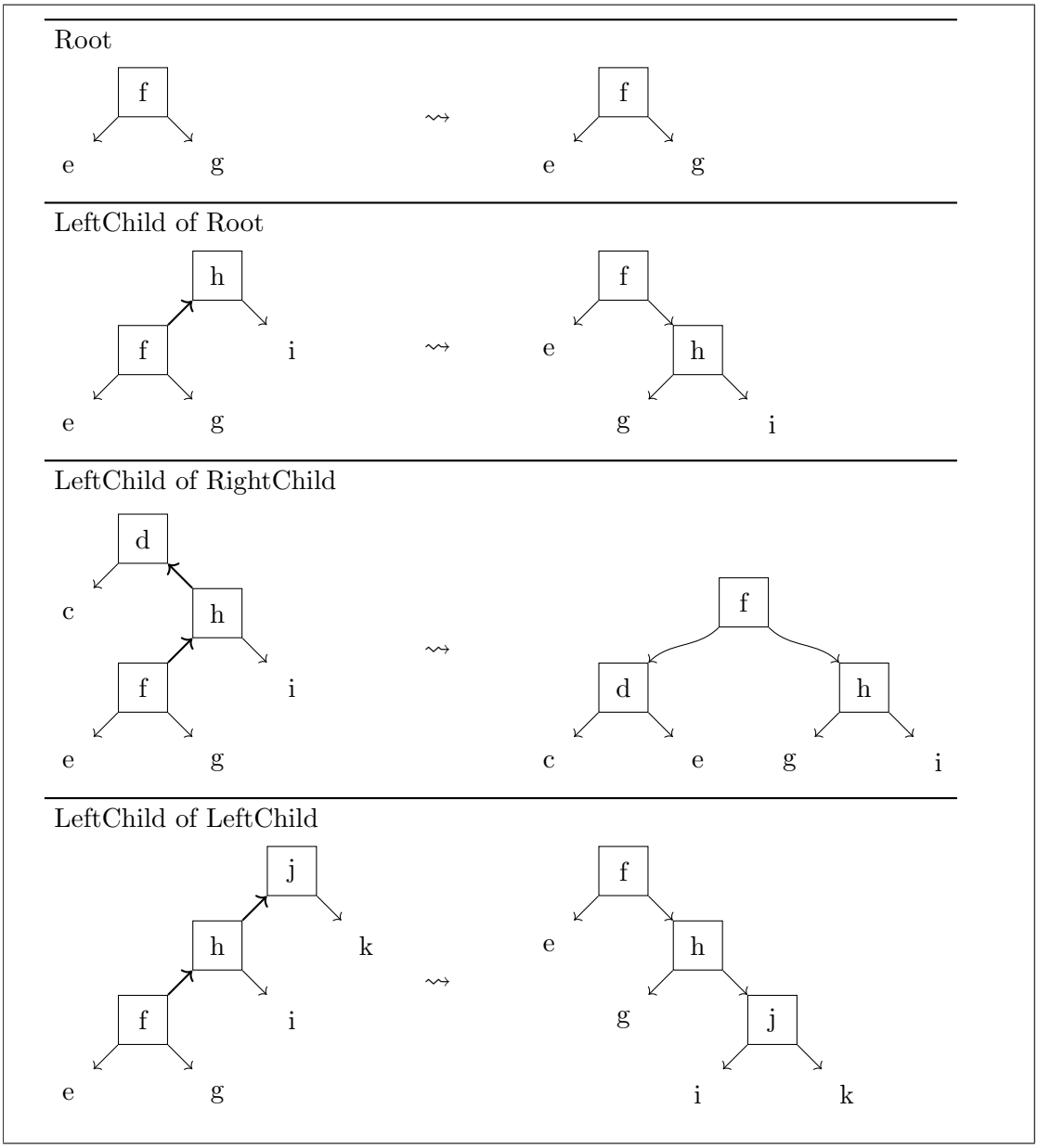

Figure 4.4: The first four cases of the `splay` function after accessing `f`. The variable names were chosen to correspond to the ordering of the tree.

```
  Root
    -> Node(e, f, g)
  LeftChild(h, Root, i)
    -> Node(e, f, Node(g, h, i))
  LeftChild(h, RightChild(d,z,c), i)
    -> splay(Node(c,d,e),f,Node(g,h,i),z)
  LeftChild(h, LeftChild(j,z,k), i)
    -> splay(e,f,Node(g,h,Node(i,j,k)),z)
  RightChild(d, Root, c)
    -> Node(Node(c, d, e), f, g)
  RightChild(d, RightChild(b,z,a), c)
    -> splay(Node(Node(a,b,c),d,e),f,g,z)
  RightChild(d, LeftChild(h,z,i), c)
    -> splay(Node(c,d,e),f,Node(g,h,i),z)
```

Splay trees perform well when similar elements are accessed closely after another: On the first access, the first element and the elements surrounding it in the tree are splayed to the top and are then quicker to access in later iterations. Splay trees can do especially well as heaps if there are longer phases in which the smallest element is extracted repeatedly as the smaller elements are close to the root in these phases. Splaying the smallest element to the root can be done by a simpler `access-min` function that only has to consider the 'left child of left child' and 'left child of root' cases:

```
fun access-min(tree)
  fun go(c, d, e)
    match(c)
      Leaf -> Node(c, d, e)
      Node(a, b, c) -> go(a, b, Node(c, d, e))
  match(tree)
    Leaf -> tree
    Node(a, b, c) -> go(a, b, c)
```

To evaluate the performance of these splay trees, we compare them against standard functional heap implementations [Oka99]. We compare the performance of heapsorting a list of 10 million random integers, and of heapsorting a list of 10 million already sorted integers (Figure 4.5). On random data, weight-balanced leftist heaps are best, but they can not take advantage of ordering and thus show similar performance on random and sorted data. Leftist heaps and Okasaki's splay heaps strike a good balance between performance on the two benchmarks with splay heaps being (unsurprisingly) significantly faster on sorted data. The zipper-based splay trees from above are slightly slower than Okasaki's splay heaps at random data, but significantly faster on sorted data. Pairing heaps and binomial heaps did not perform well at this task. The mergesort from Haskell's `Data.List`, ported to Koka, outperforms the heapsort even for the best heap implementation.

Remark 4.1. When using a splay tree to implement a set (instead of a multi-set like above), one can achieve a modest speedup by implementing the `splay` function differently. When we find that the element is already contained in the tree, we would call `splay(a, b, c)`, but this discards a node that we then have to allocate again. We can avoid this by passing `e, f, g` inside a type that has just one constructor with three fields. Similarly, one should implement a special case in `access-min` for `Node(Leaf, _, _)` so we don't have to allocate the root again if it is already the minimum element.

4.5 Red-black trees

Red-black trees [GS78] are balanced binary trees where each node is marked as either red or black. Then the tree maintains the invariants that every path from the root to a leaf contains the same number of black nodes and that no red node has a red child. It follows from this that the longest possible path in such a tree (where red and black nodes alternate) can be at most twice as long as the shortest possible path (with only black nodes), thus the tree is balanced.

We can define the datatype for such a red-black tree by defining a type of colors, a type of binary trees that stores color information and its zipper.

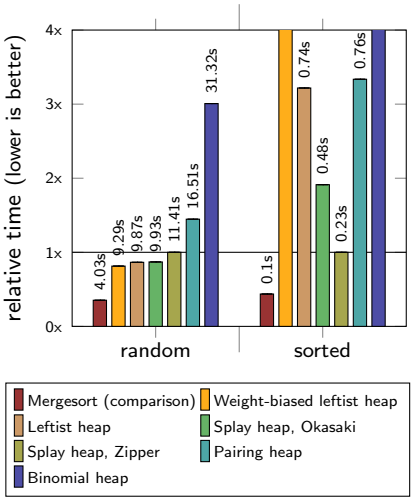

Figure 4.5: Benchmark results for heapsorts, Apple M1

```
alias elem = int

type color
  R
  B

type tree
  Node(c : color, l : tree, e : elem, r : tree)
  Leaf

type zipper
  NodeR(c : color, l : tree, e : elem, z : zipper)
  NodeL(c : color, z : zipper, e : elem, r : tree)
  Done
```

For example, when we are at the node that holds '6' in figure 4.6 we could describe this by a pair of a tree and a zipper:

```
( Node(B, Leaf, 6, Node(R, Leaf, 7, Leaf))
, NodeL(R, NodeR(B, a, 5, Done), 8, b))
```

To insert a node into such a red-black tree we will first find the correct position to insert our current element `x`. For that we compare it to the element stored in the top-most node and either go into the left or right subtree. While walking down one path in the tree this way we store the parent nodes in the Zipper. Notice how both `NodeR` and `NodeL` have the same size as `Node` so that this works without any new allocations thanks to reuse analysis.

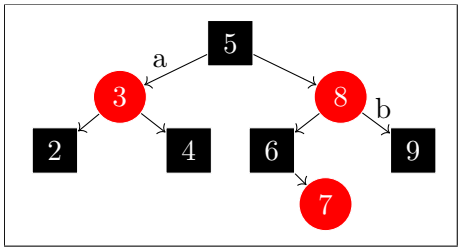

Figure 4.6: A red-black tree. Black nodes are drawn as squares, red nodes as circles.

```
fun ins(t : tree, k : elem, z : zipper) : tree
  match t
    Node(c, l, kx, r)
      -> if k < kx   then ins(l, k, NodeL(c, z, kx, r))
         elif k > kx then ins(r, k, NodeR(c, l, kx, z))
         else z.move-up(Node(c, l, kx, r))
    Leaf -> z.balance-red(Leaf, k, Leaf)
```

If the element is already in the tree, we can `move-up` the tree without any balance rotations. If we insert the element, we give it the color red and since its parent might be red as well we have to balance the new tree. For this we inspect the parent and grand-parent of the current node (using the zipper) and perform rotations such that all paths from the root to a leaf still have the same number of black nodes and the only red node with possible a red parent is the node we rotated to the top of the tree. At the end of the insertion, if the root of the tree is red, we can safely change its color to black.

```
fun move-up(z : zipper, t : tree)
  match z
    NodeR(c, l, k, z1) -> z1.move-up(Node(c, l, k, t))
    NodeL(c, z1, k, r) -> z1.move-up(Node(c, t, k, r))
    Done -> t

fun balance-red(z : zipper, l : tree, k : elem, r : tree)
  match z
    NodeR(B, l1, k1, z1) ->
      z1.move-up(Node(B, l1, k1, Node(R, l, k, r)))
    NodeL(B, z1, k1, r1) ->
      z1.move-up(Node(B, Node(R, l, k, r), k1, r1))
    NodeR(R, l1, k1, z1) -> match z1
      NodeR(_, l2, k2, z2) ->
      z2.balance-red(Node(B, l2, k2, l1), k1, Node(B, l, k, r))
      NodeL(_, z2, k2, r2) ->
      z2.balance-red(Node(B, l1, k1, l), k, Node(B, r, k2, r2))
```

```
    Done -> Node(B, l1, k1, Node(R, l, k, r))
  NodeL(R, z1, k1, r1) -> match z1
    NodeR(_, l2, k2, z2) ->
    z2.balance-red(Node(B, l2, k2, l), k, Node(B, r, k1, r1))
    NodeL(_, z2, k2, r2) ->
    z2.balance-red(Node(B, l, k, r), k1, Node(B, r1, k2, r2))
    Done -> Node(B, Node(R, l, k,r), k1, r1)
  Done -> Node(B, l, k, r)
```

Since `NodeR` and `NodeL` have the same size as `Node` we will allocate no new memory during this procedure – at least if the given tree is unique. If it is not unique we will only allocate new memory for the zipper which is then unique so that `balance-red` can always reuse. Furthermore `balance-red` and `ins` are tail-recursive so that they can be compiled to fast imperative loops. Therefore version is not only shorter but also almost as fast as an imperative implementation (like [Cor+09]) – except that it can not terminate rebalancing early like imperative implementations can: Most imperative implementations add an extra parent pointer to every node and can then implement `move-up` as a direct jump to the root of the tree. This can significantly improve performance but it is not possible with link-inversion since we have to invert all the inverted links again.

Remark 4.2. The insert function can be modified to carry the element `kx` in the recursive call and check for equality only once the bottom of the tree is reached. This can reduce the number of times the equality operator is used from $O(\log(n))$ to $O(1)$ and significantly improve performance. Similarly it is possible to improve lookup and deletion functions using this technique.

4.6 B-trees and constructor padding

In the last two sections we saw the importance of equally-sized constructors. But in many cases these will not occur naturally. Does it make sense then to fill smaller constructors with `NULL` pointers so that we can make use of reuse analysis at the cost of using slightly more memory?

We want to investigate this question at the example of B-trees of minimum degree `t = 2`, also called 2-3-4 trees. We will present only a small fragment of such an implementation to show the relevant pieces. When a node in a B-tree becomes too big it needs to be split into two nodes: the `try-split` function below shows this operation.

```
type tree
  Root0
  Node1(a:tree, b:elem, c:tree)
  Node2(a:tree, b:elem, c:tree, d:elem, e:tree)
  Node3(a:tree, b:elem, c:tree, d:elem, e:tree, f:elem, g:tree)
  Leaf1(a : elem)
  Leaf2(a : elem, b : elem)
```

```
  Leaf3(a : elem, b : elem, c : elem)

type split
  NoSplit
  Split(l : tree, a : elem, r : tree)

fun try-split(tree)
  match(tree)
    Node3(a, b, c, d, e, f, g)
      -> Split(Node1(a, b, c), d, Node1(e, f, g))
    Leaf3(a, b, c) -> Split(Leaf1(a), b, Leaf1(c))
    tree -> NoSplit
```

As can be seen above, the `try-split` can use reuse analysis only to reuse the `Leaf3` node for the `Split` node. One may wonder if it would help if all nodes of a B-tree had the same size (as usual in imperative implementations). For this we will *pad* the constructors with integers that we will all set to 0. The `try-split` function can then reuse the old node for one of the new nodes.

```
type tree
  Root0(a:int,  b:int,  c:int,  d:int,  e:int,  f:int,  g:int)
  Node1(a:tree, b:elem, c:tree, d:int,  e:int,  f:int,  g:int)
  Node2(a:tree, b:elem, c:tree, d:elem, e:tree, f:int,  g:int)
  Node3(a:tree, b:elem, c:tree, d:elem, e:tree, f:elem, g:tree)
  Leaf1(a:elem, b:int,  c:int,  d:int,  e:int,  f:int,  g:int)
  Leaf2(a:elem, b:elem, c:int,  d:int,  e:int,  f:int,  g:int)
  Leaf3(a:elem, b:elem, c:elem, d:int,  e:int,  f:int,  g:int)

fun try-split(tree)
  match(tree)
    Node3(a, b, c, d, e, f, g)
      -> Split(Node1(a,b,c,0,0,0,0), d, Node1(e,f,g,0,0,0,0))
    Leaf3(a, b, c, _, _, _, _)
      -> Split(Leaf1(a,0,0,0,0,0,0), b, Leaf1(c,0,0,0,0,0,0))
    _ -> NoSplit
```

We can optimize this version slightly: In Koka an `int` can have an arbitrary size and thus needs to be reference counted. We can therefore reuse the integers we already have for the integer slots of the new constructors and thus save us some `drop` operations:

```
fun try-split(tree)
  match(tree)
    Node3(a, b, c, d, e, f, g)
      -> Split(Node1(a,b,c,0,0,0,0), d, Node1(e,f,g,0,0,0,0))
```

```
    Leaf3(a, b, c, d, e, f, g)
      -> Split(Leaf1(a,0,0,d,e,f,g), b, Leaf1(c,0,0,0,0,0,0))
    _ -> NoSplit
```

However, padding everything turns out to be much slower; unsurprisingly since we now use significantly more memory. A simple optimization would be to separate the padding by `Leafs` and `Nodes`. This will remove some opportunities for reuse analysis, but since the `Leafs` outnumber the `Nodes` this can shave off at least a quarter of the memory usage of the full padding above.

```
type tree
  Root0
  Node1(a:tree, b:elem, c:tree, d:int,  e:int,  f:int,  g:int)
  Node2(a:tree, b:elem, c:tree, d:elem, e:tree, f:int,  g:int)
  Node3(a:tree, b:elem, c:tree, d:elem, e:tree, f:elem, g:tree)
  Leaf1(a : elem, b : int,  c : int)
  Leaf2(a : elem, b : elem, c : int)
  Leaf3(a : elem, b : elem, c : elem)
```

4.7 Conclusion

Link inversion can also be applied to more complex tree-algorithms and zippers can make these algorithms particularly elegant to describe. While they are not the right choice in each situation they provide competitive performance and can sometimes provide significant speedups. However, this area can still benefit from more research. For example, the zippers we considered above are unique at runtime by construction and so the reuse in the balance functions always succeeds. But Koka can not detect this and so needs to check at runtime whether the reuse succeeded before it can use the memory. Removing these checks manually provided further speedups of around 10%.

Reuse analysis happens very late in the compiler: After Perceus inserted reference count instructions, which itself only happens most other optimizations (e.g. inlining, specialization, removing effects). This is very important, since new reuse opportunities may arise due to optimizations, but it also makes it difficult to reason about what is reused by what without inspecting the intermediate C code Koka produces. It would be beneficial to give better feedback to the user, especially as some algorithms like those outlined above crucially depend on reuse analysis working well.

Chapter 5

Calculi for program transformations

In order to reason about program transformations, we need an abstract model in which we can express these. The most common model in computer science for discussing programs is the turing machine - but it is so far removed from programming languages that it is all but impossible to use it for our purpose. Thankfully, Alonzo Church developed the lambda calculus in the 1930s which is as strong as the turing machine but corresponds more closely to programs as they are written in functional programming languages[1]. In the next section, we will introduce this theory and show how it relates to Koka.

5.1 Computation

The lambda calculus can be described very quickly: A *term* is either a variable x, an application (MN) where M, N are itself terms or a lambda abstraction $(\lambda x, N)$ where x is a variable and N a term. We will sometimes omit the brackets for clarity; application associates to the right (abc means $(ab)c$). This can be compactly written:

$$e \coloneqq x \mid e\, e \mid \lambda x, e$$

Here $\lambda x, N$ intuitively corresponds to an anonymous function with one argument and application corresponds to function application. We say a variable x is bound if it is contained in some subterm $(\lambda x, N)$ and else we say it is free. We write $N[x := M]$ for the term which is equal to N except that every free occurrence of x is replaced by M (and all variables are appropriately renamed such that free variables in M are not bound in the new term). Then we can reduce a term by "calling" the anonymous functions: We replace a subterm $(\lambda x, N)M$ by $N[x \coloneqq M]$.

This procedure is called β-reduction and applying it as long as possible yields the β-normal form. Interestingly, it does not matter in which order β-reductions are applied if several are possible at the same time and all of them terminate: this is known as the Church-Rosser theorem. Thus, we have a basis for computation: If we want to know if a program evaluates to a certain value we can simply β-reduce it and check if we obtain

[1] In fact, functional programming was largely developed as an extension of the lambda calculus.

the value we expect. In some cases, it will be necessary to rename bound variables (for example $\lambda x, x$ should be equal to $\lambda y, y$), which we will call α-conversion.

We can write this informal description down in a formal calculus. For this we will use *sequents*. A sequent rule consists of several preconditions and a conclusion. If the preconditions can be shown to hold, then also the conclusion holds.

$$\frac{\text{precondition} \qquad \text{precondition} \qquad \text{precondition} \qquad \ldots}{\text{conclusion}}\ \text{rule name}$$

We denote β-reduction by $N \rightsquigarrow M$ and then the β-reduction rule reads:

$$\frac{}{(\lambda x, N)M \rightsquigarrow N[x := M]}\ \beta\text{-reduce}$$

However, the rule above can only be applied at the top level of a term – but we want to apply it anywhere in a term. We thus need to add three more rules that take a reduction as a precondition and insert it into the syntactical constructs above.

$$\frac{}{x \rightsquigarrow x}\ \text{var} \qquad \frac{N \rightsquigarrow N' \qquad M \rightsquigarrow M'}{NM \rightsquigarrow N'M'}\ \text{app} \qquad \frac{N \rightsquigarrow N'}{\lambda x, N \rightsquigarrow \lambda x, N'}\ \text{lam}$$

Here, the `var` rule is necessary because otherwise there would be no way to derive $N \rightsquigarrow N$ for any N. But that is necessary because otherwise we couldn't apply the `app` rule if it is possible to β-reduce M but not N.

Most programmers will not be surprised that function applications are important for computation; but they may ask where the datatypes are in this calculus. We will first present a simple example with booleans to illustrate the principle and then show how arbitrary algebraic data types can be represented in this calculus. We will write $\lambda x_1 \ldots x_n, N$ for $\lambda x_1, \ldots \lambda x_n, N$ and encode 'true' as $\lambda x\, y, x$ and 'false' as $\lambda x\, y, y$. Then the function 'and' can be written as:

$$\lambda b_1\ b_2\ x\ y,\ b_1\ (b_2\ x\ y)\ y$$

We can refer to the definitions above by name by wrapping our term N in, say, $(\lambda true, N)(\lambda x\, y, x)$. Since this is common, we usually write this as val $true = \lambda x\, y, x$; N.

From this example we can see that it is possible to represent an enumeration data type of n possible values as a function of n parameters: it represents the i-th possible value iff it returns the i-th parameter. If a function accepts an enumeration datatype and has n possible behaviors for each of the n possible values of the enumeration, these become the n arguments.

What about constructors that store some values? These can be represented by passing these values to the parameter of the function. For example, a pair of a, b might be written as $\lambda x, x\, a\, b$. Of course, this can be combined with an enumeration as we have seen with the 'list' datatype:

$$\begin{aligned} \text{Cons(a, xs):}\ & \lambda c\, n, c\, a\, xs \\ \text{Nil:}\ & \lambda c\, n, n \end{aligned}$$

We can even write a small 'map' function:

$$\text{val } mmap = \lambda map\, f\, list,\ list\ (\lambda a\ xs\ c\ n, c\,(f\ a)\,(map\, map\, f\ xs))\ (\lambda c\ n, n);$$
$$\text{val } map = mmap\, mmap;$$

Programmers use this style only infrequently in the real world since using explicit constructors leads to code that is both faster and easier to understand — but still this technique is sometimes useful and then called *continuation passing style*. In this way all algebraic data types introduced in chapter 3 can be represented. This is the magic of the lambda calculus: It is functions all the way down.

Now that we have established the general principle however, we can safely use some more syntax to make our intent clearer. In 5.1 we introduce a 'match' syntax that allows us to match on constructors by name as in Koka. Unlike in Koka we will not allow nested patterns – instead a pattern is just a constructor where we can bind the fields to variables or let them be unbound by writing an underscore.

$$\begin{aligned} e &\coloneqq x \mid e\, e \mid \lambda x, e \mid C\, e_1 \ldots e_n \\ &\quad \mid \text{match } e\ \{\overline{p_i \to e_i}\} \\ &\quad \mid \text{val } x = e;\ e \\ p &\coloneqq C\, b_1 \ldots b_n \\ b &\coloneqq x \mid \_ \end{aligned}$$

Figure 5.1: Abstract syntax of Koka

However, now that our syntax is so much richer, it would be very verbose to write down all the different reduction rules explicitly. We will therefore use a short-hand known as *evaluation contexts*, which allow us to specify the reduction rules separately and then use a combined rule to allow the reduction rules to match anywhere in the expression. In 5.2 we first specify values v that shouldn't be reduced further. Then we specify the evaluation context E as an expression with hole $\square$ in it and give an `eval` rule. If an expression e reduces to e' then the `eval` rule tells us that we can plug that reduction into the term at an arbitrary position. But the evaluation context also specifies an order of evaluation: For example the i-th entry of a constructor application can be evaluated only once the entries 1 to $i-1$ have been evaluated. We can now give the reduction rules for this syntax:

$$\begin{aligned} v &\coloneqq x \mid \lambda x, e \mid C\, v_1 \ldots v_n \\ \mathrm{E} &\coloneqq \square \mid \mathrm{E}\ e \mid v\ \mathrm{E} \mid \text{val } x = \mathrm{E};\ e \\ &\quad \mid \text{match E } \{\overline{p_i \to e_i}\} \\ &\quad \mid C\, v_1 \ldots \mathrm{E} \ldots e_n \end{aligned}$$
$$\frac{e \rightsquigarrow e'}{\mathrm{E}[e] \rightsquigarrow \mathrm{E}[e']}\ \text{eval}$$

Figure 5.2: Evaluation of 5.1

$$\frac{}{(\lambda x, e)\, v \rightsquigarrow e[x \coloneqq v]}\ \text{app} \qquad \frac{}{\text{val } x = v;\ e \rightsquigarrow e[x \coloneqq v]}\ \text{bind}$$

$$\frac{p_i = C\, x_1 \ldots x_n \text{ for exactly one } i}{\text{match } (C\, v_1 \ldots v_n)\ \{\overline{p_i \to e_i}\} \rightsquigarrow e_i[x_1 \coloneqq v_1 \ldots x_n \coloneqq v_n]}\ \text{match}$$

With this new syntax we can write the 'map' function as:

$$
\begin{aligned}
&\text{val } \mathit{mmap} = \lambda \mathit{map}\, f\, \mathit{list}, \text{ match } \mathit{list} \\
&\quad \text{Cons } a\ \mathit{xs} \to \text{Cons } (f\ a)\ (\mathit{map}\ \mathit{map}\ f\ \mathit{xs})) \\
&\quad \text{Nil } \to \text{Nil}; \\
&\text{val } \mathit{map} = \mathit{mmap}\ \mathit{mmap};
\end{aligned}
$$

Exercise 5.1 (Nontermination)**.** Turing machines can express non-terminating programs and so can the lambda calculus. Show that β-reducing $(\lambda x, x\, x)(\lambda x, x\, x)$ will never terminate. Can you give a term such that β-reduction will increase the size of the term each time it is applied? Can you give a term where β-reduction terminates if we choose the subterms to apply it to in some order, but where it doesn't terminate if we choose them in another order?

Exercise 5.2 (Booleans)**.** Define an 'or' function in pure lambda calculus and show that 'true or false = true' with the sequent calculus.

Exercise 5.3 (Evaluation context)**.** Why is there no rule 'val $x = v$; E' or 'λx, E' in 5.2?

Exercise 5.4 (Implementation)**.** Consider how you could write the evaluation procedure for 5.1 in Koka. Write down the ADTs that describe the syntax and implement the `eval` rule. Is the description of the evaluation order ambiguous?

5.2 A-normalization

Many optimization techniques for functional programming languages can be derived directly from the calculus defined above. For example, *inlining* a function call corresponds to applying the `bind` rule selectively: We replace only one occurrence of the function symbol x by the definition v. Similarly, executing a computation on a known constant (*constant folding*) can be done by applying the `app` and `match` rules.

We will describe one such optimization in detail because it is an important difference between Lean and Koka and will give us a new technique for evaluation contexts. A-normalization [Fla+93] transforms a program into *A-normal* form which exposes more opportunities for optimization. One example of A-normal form in action:

$$
\begin{aligned}
&\text{match } (\text{val } x = \ldots;\ A\ x) \\
&\quad A\ x \to \ldots \\
&\quad B \to \ldots
\end{aligned}
\qquad \rightsquigarrow \qquad
\begin{aligned}
&\text{val } x = \ldots; \\
&\text{match } (A\ x) \\
&\quad A\ x \to \ldots \\
&\quad B \to \ldots
\end{aligned}
$$

While the left-hand side might stay unoptimized by a simple compiler, the right hand side is a clear opportunity for constant folding: The compiler can choose the first branch at compile time and thus avoid some runtime overhead. While a programmer would be unlikely to write code like the piece shown on the left hand side explicitly, this situation can occur due to inlining (in fact, one of the main reasons for inlining is exposing new transformations [JM02]).

We can describe A-normalization using evaluation contexts – but with a catch: this time we will move and copy the term that the evaluation context represents. In Figure 5.3 we give the definition of the evaluation context $\mathcal{A}$. For the $\mathcal{A}$-reductions given below we will assume that all variable names in the program are unique and that $\mathcal{A} \neq \square$.

$$\begin{aligned}\mathcal{A} ::= \square \mid{}& \mathcal{A}\ e \mid v\ \mathcal{A} \mid \text{val } x = \mathcal{A};\ e \\ \mid{}& \text{match } \mathcal{A}\ \{\overline{p_i \to e_i}\} \\ \mid{}& C\ v_1 \ldots \mathcal{A} \ldots e_n\end{aligned}$$

Figure 5.3: The evaluation context of A-normalization

$$\begin{aligned}\mathcal{A}[\text{val } x = v;\ e] &\longrightarrow \text{val } x = v;\ \mathcal{A}[e] \\ \mathcal{A}[\text{match } v\ \{\overline{p_i \longrightarrow e_i}\}] &\longrightarrow \text{match } v\ \{\overline{p_i \longrightarrow A[e_i]}\} \\ \mathcal{A}[v_1\ v_2] &\longrightarrow \text{val } t = v_1\ v_2;\ \mathcal{A}[t] \\ &\qquad \text{where } \mathcal{A} \neq \mathcal{A}'[\text{val } z = \square;\ e], t \text{ is a fresh variable}\end{aligned}$$

While A-normalization is quite powerful for optimizing programs, it will duplicate code every time the evaluation context is used more than once in the rules above. Thus, it can lead to an exponential increase in code size. One can avoid this either by only performing the `match` normalization when code duplication is unlikely or impossible, or by creating a new function for the term of the evaluation context (a *join point* [Mau+17]). While Koka uses the first option, Lean uses the second.

5.3 Static reference count instructions

Koka inserts reference counting instructions using the Perceus algorithm [Rei+21]. They split their description into three parts: The linear resource calculus λ^1 gives a formal specification of valid ways to place reference count instructions such that the resulting program is sound: No value is freed before it is last used and in the end no dead memory remains. This property is proved using a simplified model of the Heap. Finally, the Perceus algorithm is given as a calculus that places dups and drops in a valid way such that the resulting program does not hold on to memory longer than necessary.

The λ^1 calculus uses the language we developed in 5.1. In part this is a practical decision: `match`-syntax is easy to read and maps well to if-statements that any backend will support. But it also has a theoretical reason: A lambda will need to capture all its free variables since it might live longer than other references to its free variables. But the syntactic sugar we have defined before uses lambdas that will be called only once

$$\frac{}{\Delta \mid x \ \vdash \ x \rightsquigarrow x}\ \text{var} \qquad \frac{\Delta \mid \Gamma, x \ \vdash \ e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \ \vdash \ e \rightsquigarrow \text{dup}\ x;\ e'}\ \text{dup}$$

$$\frac{\Delta \mid \Gamma \ \vdash \ e \rightsquigarrow e'}{\Delta \mid \Gamma, x \ \vdash \ e \rightsquigarrow \text{drop}\ x;\ e'}\ \text{drop}$$

$$\frac{\varnothing \mid \Gamma, x \ \vdash \ e \rightsquigarrow e' \qquad \Gamma = \text{free variables of } \lambda x,\ e}{\Delta \mid \Gamma \ \vdash \ \lambda x,\ e \rightsquigarrow \lambda^{\Gamma} x,\ e'}\ \text{lam}$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \ \vdash \ e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2 \ \vdash \ e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \ \vdash \ e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ \text{app}$$

$$\frac{x \notin \Delta, \Gamma_1, \Gamma_2 \qquad \Delta, \Gamma_2 \mid \Gamma_1 \ \vdash \ e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2, x \ \vdash \ e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \ \vdash \ \text{val}\ x = e_1;\ e_2 \rightsquigarrow \text{val}\ x = e_1';\ e_2'}\ \text{val}$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \ \vdash \ e \rightsquigarrow e' \qquad \Delta \mid \Gamma_2, \text{bv}(p_i) \ \vdash \ e_i \rightsquigarrow e_i'}{\Delta \mid \Gamma_1, \Gamma_2 \ \vdash \ \text{match}\ e\ \{\overline{p_i \to e_i}\} \rightsquigarrow \text{match}\ e'\ \{\overline{p_i \to e_i'}\}}\ \text{match}$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \ \vdash \ e_i \rightsquigarrow e_i' \qquad 1 \leq i \leq n}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \ \vdash \ C\, e_1 \ldots e_n \rightsquigarrow C\, e_1' \ldots e_n'}\ \text{con}$$

Figure 5.4: The λ^1 calculus

and we can generate better code for them if we know of this property. We will see an example shortly.

Each transformation of the λ^1 calculus is parameterized over two multi-sets of variables Δ and Γ. The multi-set Γ contains all variables for which the expression holds a reference and Δ contains all variables for which we know that some other part of the program holds a reference. Both are empty at the start. Any rule will then be of the form $\Delta \mid \Gamma \ \vdash \ e \rightsquigarrow e'$. For the `var` rule we assume that we hold only a reference to this variable ($\Gamma = \{x\}$) and some arbitrary Δ-environment. We can modify the multi-set Γ with the `dup` and `drop` rules: As long as there is *any* reference to a variable in either Γ or Δ we can create a new one and we can always remove old references.

The `lam` rule introduces the conversion of lambda-expressions. Here we assume that Γ is a set, namely the set of free variables of the lambda. We will then "store" these references in the lambda and denote this by λ^{Γ}. Since the lambda may live longer than the references in the Δ-environment, we cannot use them for the expression inside the lambda.

The `app` and `val` rules are very similar: We split the Γ-environment between two different expressions. Since in our evaluation order (given in 5.2) we always consider first e_1 and then e_2 we can add the references we assign to the Γ of the second expression to the Δ of the first expression. The similarity should not be surprising: We have seen that $\text{val}\ x = e_1; e_2$ can also be written as $(\lambda x,\ e_2)\ e_1$. But since we know that the thus created

$\mathrm{H} : x \to (\mathbb{N}^+, v)$

$\mathrm{E} \coloneqq \Box \mid \mathrm{E}\ e \mid x\ \mathrm{E} \mid \mathrm{val}\ x = \mathrm{E};\ e$
$\quad \mid C\ x_1 \ldots x_{i-1}, \mathrm{E}, e_{i+1} \ldots e_n \mid \mathrm{match}\ \mathrm{E}\ \{\overline{p_i \to e_i}\}$

$$\frac{\mathrm{H} \mid e \longrightarrow_r \mathrm{H}' \mid e'}{\mathrm{H} \mid \mathrm{E}[e] \longrightarrow_r \mathrm{H}' \mid \mathrm{E}[e']}\ \mathrm{eval}$$

(lam_r)	$\mathrm{H} \mid (\lambda^{ys} x,\ e)$	$\longrightarrow_r$	$\mathrm{H}, f \mapsto^1 \lambda^{ys} x,\ e \mid f$	fresh f
(con_r)	$\mathrm{H} \mid C\ x_1 \ldots x_n$	$\longrightarrow_r$	$\mathrm{H}, z \mapsto^1 C\ x_1 \ldots x_n \mid z$	fresh z
(app_r)	$\mathrm{H} \mid f\ z$	$\longrightarrow_r$	$\mathrm{H} \mid \mathrm{dup}\ ys;\ \mathrm{drop}\ f;\ e[x \coloneqq z]$ with $(f \mapsto^n \lambda^{ys} x,\ e) \in \mathrm{H}$	
$(match_r)$	$\mathrm{H} \mid \mathrm{match}\ x\ \{\overline{p_i \to e_i}\}$	$\longrightarrow_r$	$\mathrm{H} \mid \mathrm{dup}\ ys;\ \mathrm{drop}\ x;\ e[xs \coloneqq ys]$ with $p_i = C\ xs$ and $(x \mapsto^n C\ ys) \in \mathrm{H}$	
$(bind_r)$	$\mathrm{H} \mid \mathrm{val}\ x = y;\ e$	$\longrightarrow_r$	$\mathrm{H} \mid e[x \coloneqq y]$	
(dup_r)	$\mathrm{H}, x \mapsto^n v \mid \mathrm{dup}\ x;\ e$	$\longrightarrow_r$	$\mathrm{H}, x \mapsto^{n+1} v \mid e$	
$(drop_r)$	$\mathrm{H}, x \mapsto^{n+1} v \mid \mathrm{drop}\ x;\ e$	$\longrightarrow_r$	$\mathrm{H}, x \mapsto^n v \mid e$	if $n \geq 1$
$(dlam_r)$	$\mathrm{H}, x \mapsto^1 \lambda^{ys} z,\ e \mid \mathrm{drop}\ x;\ e$	$\longrightarrow_r$	$\mathrm{H} \mid \mathrm{drop}\ ys;\ e$	
$(dcon_r)$	$\mathrm{H}, x \mapsto^1 C\ ys \mid \mathrm{drop}\ x;\ e$	$\longrightarrow_r$	$\mathrm{H} \mid \mathrm{drop}\ ys;\ e$	

Figure 5.5: The heap semantics for λ^1

lambda will be called immediately and can not live longer than the other references in the program, we can use the Δ environment for e_2 and the Γ_2 multi-set for the Δ-environment of e_1.

Similarly, the `match` rule also benefits from the extra syntax. We define $\mathrm{bv}(p_i)$ to be the variables bound by (occurring in) p_i. Then the rule is exactly like you would infer it from `matchs` representation in pure lambda calculus – except that we can once again pass the Δ-environment into the branches. We could even remove the assumption that x must be in the Γ-environment, which Koka does in practice. Finally, the `con` rule simply transforms every expression applied to the constructor[2].

Any expression e' that can be derived by the λ^1 calculus ($\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$) and evaluates to a value using the strict semantics 5.2 can also be evaluated using the heap semantics 5.5. It models the behaviour that a Koka program shows when evaluated on a real heap. Any used memory location is modelled by a variable: the Heap is then a function from variables to a pair of a reference count and a value. When the heap is applied to an expression, we first evaluate some subexpressions according to the evaluation context E. The expression is then in a normalized form and we only give rules for these forms.

[2] [Rei+21] write the expressions applied to the constructor as values, but this is inconsistent with the behavior of Perceus which may insert `dup` calls there (which are not values according to their classification).

If the heap encounters a value in the (lam_r) or (con_r), it stores this value at a new memory location (given by a fresh variable). When it encounters a variable, the evaluation inside the innermost frame of the current evaluation context ends and we either process the second-innermost evaluation context or, if none exists, end the evaluation. When we encounter a function application, we increase the reference counts of the free variables and drop the function: This corresponds directly to the `lam` rule, as we put the free variables into the Γ environment there. Similarly, we need to dup the bound variables in a match statement as we added them to the Γ statement in the `match` rule and drop the scrutinee as we deleted it from Γ. We can handle a binding (in $bind_r$) by writing the memory location of the bound variable into the places where we use the new name.

Dups and drops are implemented by increasing or decreasing the reference counts. When a value with a reference count of one is dropped, we delete the value from the heap and drop its children.

5.4 Perceus

While λ^1 already restricts the places where reference count instructions can be placed, it still leaves considerable freedom. The Perceus algorithm presents one way of choosing when exactly to dup and drop variables. For this it maintains four invariants that already characterize the algorithm in full. For any derivation $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$:

- Both Γ and Δ are sets, that is each variable occurs only once in them
- $\Delta \cap \Gamma = \varnothing$
- $\Gamma \subseteq \text{fv}(e)$, the set of free variables of e

Together these invariants guarantee that a variable is dropped as soon as possible because otherwise for this variable x we have $x \in \Gamma$ but $x \notin \text{fv}(e)$. Furthermore they specify that any variable x will also be dupped as late as possible because else either Γ would not be a set or $x \in \Gamma \cap \Delta$. As any algorithm that implements λ^1, Perceus also maintains that $\text{fv}(e) \subseteq (\Gamma \cup \Delta)$. The full calculus can be found in Figure 5.6.

Just like the evaluation 5.2 algorithm, the Perceus algorithm is *syntax-directed*: It can be implemented by pattern-matching on the syntactic constructs. While the evaluation algorithm might change the syntactic representation and is thus not linear time, the Perceus algorithm merely inserts instructions it will not consider again and is thus linear time. One has to be careful with the handling of free variables though: Finding the free variables of a term takes linear time and an implementation can easily become quadratic if they are recomputed every time they are needed for a rule. In Koka's Perceus implementation the free variables are computed together with the insertion of reference count instructions in a single pass – with the exception of function definitions where an extra pass is necessary to find all free variables of the function. Caching the free variables of every function then achieves a linear time algorithm.

$$\frac{}{\Delta \mid x \ \vdash_s \ x \rightsquigarrow x}\ \text{svar} \qquad \frac{}{\Delta, x \mid \varnothing \ \vdash_s \ x \rightsquigarrow \text{dup}\ x;\ x}\ \text{svar-dup}$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \ \vdash_s \ e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2 \ \vdash_s \ e_2 \rightsquigarrow e_2' \qquad \Gamma_2 = \Gamma \cap \text{fv}(e_2)}{\Delta \mid \Gamma \ \vdash_s \ e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ \text{sapp}$$

$$\frac{x \in \text{fv}(e) \qquad \Gamma \subseteq ys = \text{fv}(\lambda x,\ e) \qquad \Delta_1 = ys - \Gamma \qquad \varnothing \mid ys, x \ \vdash_s \ e \rightsquigarrow e'}{\Delta, \Delta_1 \mid \Gamma \ \vdash_s \ \lambda x,\ e \rightsquigarrow \text{dup}\ \Delta_1;\ \lambda^{ys} x,\ e'}\ \text{slam}$$

$$\frac{x \notin \text{fv}(e) \qquad \Gamma \subseteq ys = \text{fv}(\lambda x,\ e) \qquad \Delta_1 = ys - \Gamma \qquad \varnothing \mid ys, x \ \vdash_s \ e \rightsquigarrow e'}{\Delta, \Delta_1 \mid \Gamma \ \vdash_s \ \lambda x,\ e \rightsquigarrow \text{dup}\ \Delta_1;\ \lambda^{ys} x,\ (\text{drop}\ x;\ e')}\ \text{slam-d}$$

$$\frac{\begin{array}{c} x \in \text{fv}(e_2) \quad x \notin \Delta, \Gamma \qquad \Delta \mid \Gamma_2, x \ \vdash_s \ e_2 \rightsquigarrow e_2' \\ \Gamma_2 = \Gamma \cap (\text{fv}(e_2) - \{x\}) \qquad \Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \ \vdash_s \ e_1 \rightsquigarrow e_1' \end{array}}{\Delta \mid \Gamma \ \vdash_s \ \text{val}\ x = e_1;\ e_2 \rightsquigarrow \text{val}\ x = e_1';\ e_2'}\ \text{sval}$$

$$\frac{\begin{array}{c} x \notin \text{fv}(e_2), \Delta, \Gamma \qquad \Delta \mid \Gamma_2, x \ \vdash_s \ e_2 \rightsquigarrow e_2' \\ \Gamma_2 = \Gamma \cap \text{fv}(e_2) \qquad \Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \ \vdash_s \ e_1 \rightsquigarrow e_1' \end{array}}{\Delta \mid \Gamma \ \vdash_s \ \text{val}\ x = e_1;\ e_2 \rightsquigarrow \text{val}\ x = e_1';\ \text{drop}\ x;\ e_2'}\ \text{sval-d}$$

$$\frac{\Delta \mid \Gamma_i \ \vdash_s \ e_i \rightsquigarrow e_i' \qquad \Gamma_i = (\Gamma, \text{bv}(p_i)) \cap \text{fv}(e_i) \qquad \Gamma_i' = (\Gamma, \text{bv}(p_i)) - \text{fv}(e_i)}{\Delta \mid \Gamma, x \ \vdash \ \text{match}\ x\ \{\overline{p_i \to e_i}\} \rightsquigarrow \text{match}\ x\ \{\overline{p_i \to \text{drop}\ \Gamma_i';\ e_i'}\}}\ \text{smatch}$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \ \vdash_s \ e_i \rightsquigarrow e_i' \qquad \Gamma_i = (\Gamma - \cup_{j=i+1}^{n} \Gamma_j) \cap \text{fv}(e_i) \quad 1 \le i \le n}{\Delta \mid \Gamma \ \vdash \ C\ e_1 \ldots e_n \rightsquigarrow C\ e_1' \ldots e_n'}\ \text{scon}$$

Figure 5.6: The Perceus algorithm

5.5 Properties

The first property that should interest us with an algorithm like Perceus is *soundness*: It should not change the semantics of our program. For this we will in a first step relate any λ^1 derivation and its heap semantics to the basic evaluation given in 5.2. We write $[H]e$ for the expression where every free variable in e is recursively replaced by its value in H.

Theorem 5.1 (Theorem 1 in [Rei+21]). If $\varnothing \mid \varnothing \ \vdash \ e \rightsquigarrow e'$ and $e \rightsquigarrow v$, then $\varnothing \mid e' \longrightarrow_r \mathrm{H} \mid x$ and $[\mathrm{H}]x = v$.

This result says that at the end of the program we can read the correct result from the Heap. Then the fact that Perceus is sound follows from the fact that it is an implementation of the λ^1 calculus:

Theorem 5.2 (Theorem 3 in [Rei+21]). If $\Delta \mid \Gamma \ \vdash_s \ e \rightsquigarrow e'$ then also $\Delta \mid \Gamma \ \vdash \ e \rightsquigarrow e'$.

The other properties that we are interested in concern the values that remain in the heap. First we can show that any λ^1 derivation will leave no garbage when finished. For

this we want to ensure that any value that is still live at the end of the program, can be accessed by the result value:

Definition 5.1.

A variable x is *reachable* at an evaluation step $\mathrm{H} \mid e$ if either

- $x \in \mathrm{fv}(e)$
- there is some y reachable at $\mathrm{H} \mid e$ such that $y \mapsto^n v \in H$ and x is reachable at $\mathrm{H} \mid v$.

Then we can write the claim that λ^1 leaves no garbage as:

Theorem 5.3. If $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$ and $\varnothing \mid e' \longrightarrow_r \mathrm{H} \mid x$ then any $y \in \mathrm{dom}(\mathrm{H})$ is reachable at $\mathrm{H} \mid x$.

A generalization of 5.3 might be:

Theorem 5.4 (Theorem 2 in [Rei+21])**.** If $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$ and $\varnothing \mid e' \longrightarrow_r \mathrm{H} \mid x$ then for every intermediate state $\mathrm{H}_i \mid e_i$, all $y \in \mathrm{dom}(\mathrm{H}_i)$ are reachable at $\mathrm{H}_i \mid e_i$.

But while this theorem reveals more about the inner workings of the heap, it is not stronger than 5.3: Since values can only leave the heap through a `drop` operation, they need to be reachable to be cleaned up in a later state. However it becomes interesting when we restrict reachability to exclude the free variables used only in `dup` and `drop` operations: Then variables need to be freed as soon as they become unused. We write $\lceil e \rceil$ for an expression e with its `dup` and `drop` operations removed.

Theorem 5.5 (Theorem 4 in [Rei+21])**.** If $\varnothing \mid \varnothing \vdash_s e \rightsquigarrow e'$ and $\varnothing \mid e' \longrightarrow_r \mathrm{H} \mid x$ then for every intermediate state $\mathrm{H}_i \mid e_i$ such that $e_i \neq \mathrm{E}[\mathrm{drop}\ z;\ e_i']$ and $e_i \neq \mathrm{E}[\mathrm{dup}\ z;\ e_i']$, all $y \in \mathrm{dom}(\mathrm{H}_i)$ are reachable at $\mathrm{H}_i \mid \lceil e_i \rceil$.

In other words, Perceus may keep unused values around but only as long as the heap is processing `dup` and `drop` statements. For example, when a variable x becomes unused but we want to continue to use its child y (e.g. after a `match` on x), the heap will process dup y; drop x; and will be garbage-free afterwards.

Chapter 6

Borrowing

In this chapter we will describe the technicalities of borrowing. In particular, we describe how to extend the λ^1 calculus to support borrowing and under which conditions it is safe (or not safe) to borrow.

The intuitive description of section 3.7 can be axiomatized in the λ^1 calculus 5.4 roughly as follows: We allow the argument to certain lambdas to be a variable that is in the Δ environment. Then in the `lam` rule we do not add x to the Γ environment, but instead to the Δ environment. This already achieves the effect we hoped for: instead of passing a reference to the function we instead assert that a reference exists and allow it to access it without handing over control. There are three problems with this approach though:

- The argument to a borrowing lambda may not be a variable but instead a value. Then we need to bind this value to a variable first. We handle this by giving a normalization algorithm.
- In Koka functions take several arguments, but in the λ^1 calculus they take only one. Replacing a multi-argument function $\lambda x\ y$ by λx, λy, places x in the free variables of the λy. But free variables need to be in the Γ environment, as we show later. Thus we need to add multi-variable lambdas to our calculus.
- If a borrowing function is passed as a higher-order parameter, we have to make sure that the information about borrowing is preserved. While this could be done at runtime, it complicates our algorithms too much so we instead provide a simple method for *wrapping* a borrowing function into a non-borrowing one.

6.1 Normalization

We design our normalization algorithm in the style of A-normalization 5.2. For this we use two evaluation contexts: One, called $\mathcal{T}$, to traverse the syntax tree and another, called $\mathcal{M}$, to move expressions out of applications to a binding site. We give their definition in Figure 6.1.

$$\frac{\Delta, x \mid \Gamma \ \vdash \ e \rightsquigarrow e'}{\Delta \mid \Gamma, x \ \vdash \ e\ x \rightsquigarrow e'\ x}\ \text{app} \qquad \frac{}{\Delta \mid x_1 \ldots x_n \ \vdash \ C\, x_1 \ldots x_n \rightsquigarrow C\, x_1 \ldots x_n}\ \text{con}$$

$$\frac{\Delta \mid \Gamma, \text{bv}(p_i) \ \vdash \ e_i \rightsquigarrow e_i'}{\Delta \mid \Gamma, x \ \vdash \ \text{match}\ x\ \{\overline{p_i \to e_i}\} \rightsquigarrow \text{match}\ x\ \{\overline{p_i \to e_i'}\}}\ \text{match}$$

Figure 6.2: Simplified rules of the λ^1 calculus

$$\begin{aligned}
\mathcal{T} &:= \square \mid \mathcal{T}\ x \mid \text{val}\ x = \mathcal{T};\ e \\
&\mid \text{match}\ e\ \{\overline{p_i \to \mathcal{T}}\} \\
&\mid \text{val}\ x = e;\ \mathcal{T} \\
\mathcal{M} &:= \square \mid e\ \mathcal{M} \\
&\mid \text{match}\ \mathcal{M}\ \{\overline{p_i \to e_i}\} \\
&\mid C\, x_1 \ldots \mathcal{M} \ldots e_n \\
&\frac{\mathcal{M}[e] \longrightarrow e'}{\mathcal{T}[\mathcal{M}[e]] \longrightarrow \mathcal{T}[e']}\ \text{norm}
\end{aligned}$$

Figure 6.1: The evaluation contexts of borrowing normalization

The rationale behind this approach is that we want to move expressions out of certain syntactical constructs (like applications or the scrutinee of a `match`) but not too far out: We do not want to move them past other `val` bindings they may depend on. A-normalization solves this by transforming the entire program, but we are content with changing only some constructs. We therefore use the $\mathcal{T}$ context to move our attention to a specific place in the syntax tree and then use the $\mathcal{M}$ context to move everything except variables out of these constructs.

For the $\mathcal{M}$-reductions given below we will assume that all variable names in the program are unique, x denotes a fresh (unused) variable, $\mathcal{M} \neq \square$, and $\mathcal{M} \neq \mathcal{M}'[\text{val}\ z = \square;\ e]$.

$$\begin{aligned}
\mathcal{M}[x] &\longrightarrow x \\
\mathcal{M}[\text{val}\ y = e_1;\ e_2] &\longrightarrow \text{val}\ y = e_1;\ \mathcal{M}[e_2] \\
\mathcal{M}[v] &\longrightarrow \text{val}\ x = v;\ \mathcal{M}[x] \\
&\quad \text{where } v \text{ is not a variable} \\
\mathcal{M}[\text{match}\ y\ \{\overline{p_i \to e_i}\}] &\longrightarrow \text{val}\ x = \text{match}\ y\ \{\overline{p_i \to e_i}\};\ \mathcal{M}[x] \\
\mathcal{M}[e_1\ e_2] &\longrightarrow \text{val}\ x = e_1\ e_2;\ \mathcal{M}[x]
\end{aligned}$$

In practice, Koka does not perform a complete normalization and instead only normalizes the scrutinees of `match` statements and arguments that will be borrowed. This does not change performance or correctness, but helps to keep the C-output readable. Still, our normalization algorithm here is more general and allows us to derive a shorter version of the λ^1 calculus (see figure 6.2).

6.2 Multi-variable lambdas

$$\frac{x \mid \Gamma \;\vdash\; e \rightsquigarrow e' \qquad \Gamma = \text{free variables of } \lambda_b\, x,\, e}{\Delta \mid \Gamma \;\vdash\; \lambda_b\, x,\, e \rightsquigarrow \lambda_b^{\Gamma}\, x,\, e'}\ \text{blam}$$

Figure 6.3: An unsuitable borrowing rule

$$e \coloneqq \lambda x_1 \ldots x_n,\ e \\ \mid e\ e_1 \ldots e_n \mid \ldots$$

Figure 6.4: Syntax extension of 5.1 for borrowing

Assume for a moment we introduced a new syntax $(\lambda_b x, e)$ for borrowed parameters x together with the rule in 6.3. The rule looks innocuous enough and it is not wrong – it just doesn't work as we might expect it to. Consider how reference count instructions would be inserted into: $\lambda_b\, x,\ \lambda y,\ e$. Assuming that $x \in \text{fv}(e)$, also $x \in \text{fv}(\lambda y,\ e)$ and so x must be in the Γ environment for the inner lambda. Thus we arrive at: $\lambda_b\, x,\ \text{dup}\ x;\ \lambda y,\ e$. It is not possible to remove the x from the Γ environment of the inner lambda safely because we can not guarantee that the lambda is fully applied and thus x might be used by e after its last remaining reference. Instead we are forced to introduce new syntax 6.4 to get a better calculus – similar to how we had to introduce a `match` statement to use the Δ environment in the branches.

In our further analysis we will assume that every application of a lambda will use exactly the right number of arguments. For many typed languages (like Koka) this is already the case. For others, like Haskell, that make generous use of partial application a normalization algorithm needs to include η-expansion.

We can now state a preliminary version of borrowing that copies every function with n arguments 2^n-times; with every possible combination of arguments borrowed. This lets us avoid the problems of wrapping (which we deal with in the next section) for a moment. We transform every lambda into a set of lambdas indexed by a *borrowing vector* b and choose a borrowing vector during application which we then use to index into the set. For a sequence of variables $x_1 \ldots x_n$ and $b \in \{0,1\}^n$ we write $b\overline{x}$ for the set of variables x_i where $b_i = 1$. Similarly we write $(1-b)\overline{x}$ for the set of those variables x_i where $b_i = 0$. The `app` rule is thus a special case of `bapp` in 6.5 where $b \equiv 0$ and every function has

$$\frac{\Delta, \overline{x} \mid \Gamma \;\vdash\; e \rightsquigarrow e' \qquad b\overline{x} \in \Delta \qquad b \in \{0,1\}^n}{\Delta \mid \Gamma, (1-b)\overline{x} \;\vdash\; e\ x_1 \ldots x_n \rightsquigarrow e'_b\ x_1 \ldots x_n}\ \text{bapp}$$

$$\frac{\Gamma, b\overline{x} \mid (1-b)\overline{x} \;\vdash\; e \rightsquigarrow e_b \quad \forall b \in \{0,1\}^n \qquad \Gamma = \text{fv}(\lambda\overline{x},\ e)}{\Delta \mid \Gamma \;\vdash\; \lambda x_1 \ldots x_n,\ e \rightsquigarrow \{\lambda^{\Gamma}\, x_1 \ldots x_n,\ e_b\}_{b \in \{0,1\}^n}}\ \text{blam}$$

$$\frac{\Delta, \text{bv}(p_i) \mid \Gamma \;\vdash\; e_i \rightsquigarrow e'_i \qquad x \in \Delta}{\Delta \mid \Gamma \;\vdash\; \text{match}\ x\ \{\overline{p_i \to e_i}\} \rightsquigarrow \text{match}\ x\ \{\overline{p_i \to e'_i}\}}\ \text{bmatch}$$

Figure 6.5: Copying functions for borrowing

$$\frac{x \neq f_b}{\Delta \mid x \;\vdash\; x \rightsquigarrow x}\ \text{var} \qquad \frac{\Delta, x_1, \ldots x_n \mid \Gamma \;\vdash\; e \rightsquigarrow e' \qquad e \neq f_b}{\Delta \mid \Gamma, x_1 \ldots x_n \;\vdash\; e\ x_1 \ldots x_n \rightsquigarrow e'\ x_1 \ldots x_n}\ \text{app}$$

$$\frac{\Delta \mid \Gamma, x \;\vdash\; e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \;\vdash\; e \rightsquigarrow \text{dup}\ x;\ e'}\ \text{dup} \qquad \frac{\Delta \mid \Gamma \;\vdash\; e \rightsquigarrow e'}{\Delta \mid \Gamma, x \;\vdash\; e \rightsquigarrow \text{drop}\ x;\ e'}\ \text{drop}$$

$$\frac{x \notin \Delta, \Gamma_1, \Gamma_2 \qquad \Delta, \Gamma_2 \mid \Gamma_1 \;\vdash\; e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2, x \;\vdash\; e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \;\vdash\; \text{val}\ x = e_1;\ e_2 \rightsquigarrow \text{val}\ x = e_1';\ e_2'}\ \text{val}$$

$$\frac{\varnothing \mid \Gamma, x_1 \ldots x_n \;\vdash\; e \rightsquigarrow e' \qquad \Gamma = \text{free variables of } \lambda x,\ e}{\Delta \mid \Gamma \;\vdash\; \lambda x_1 \ldots x_n,\ e \rightsquigarrow \lambda^{\Gamma}\ x_1 \ldots x_n,\ e'}\ \text{lam}$$

$$\frac{\Delta \mid \Gamma, \text{bv}(p_i) \;\vdash\; e_i \rightsquigarrow e_i' \qquad x \in \Delta, \Gamma;\ x \neq f_b}{\Delta \mid \Gamma \;\vdash\; \text{match}\ x\ \{\overline{p_i \to e_i}\} \rightsquigarrow \text{match}\ x\ \{\overline{p_i \to \text{dup bv}(p_i);\ e_i'}\}}\ \text{match}$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \;\vdash\; x_i \rightsquigarrow e_i' \qquad 1 \leq i \leq n}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \;\vdash\; C\ x_1 \ldots x_n \rightsquigarrow C\ e_1' \ldots e_n'}\ \text{con}$$

$$\frac{}{\Delta \mid f_b \;\vdash\; f_b \rightsquigarrow \lambda \overline{x},\ \text{val}\ z = f_b\ \overline{x};\ \text{drop}\ b\overline{x};\ z}\ \text{wrapping}$$

$$\frac{b\overline{x} \in \Delta \qquad b \in \{0,1\}^n}{\Delta \mid f_b, (1-b)\overline{x} \;\vdash\; f_b\ x_1 \ldots x_n \rightsquigarrow f_b\ x_1 \ldots x_n}\ \text{bapp}$$

$$\frac{\begin{array}{c} b\overline{x} \mid \text{fv}(\lambda \overline{x},\ e_1), (1-b)\overline{x} \;\vdash\; e_1 \rightsquigarrow e_1' \\ \Delta \mid (\Gamma \setminus \text{fv}(\lambda \overline{x},\ e_1)), f_b \;\vdash\; e_2 \rightsquigarrow e_2' \qquad b \in \{0,1\}^n, f \notin \Delta, \Gamma \end{array}}{\Delta \mid \Gamma \;\vdash\; \text{val}\ f = \lambda x_1 \ldots x_n,\ e_1;\ e_2 \rightsquigarrow \text{val}\ f = \lambda^{\Gamma}\ x_1 \ldots x_n,\ e_1';\ e_2'}\ \text{blam}$$

$$\frac{\Delta, \text{bv}(p_i) \mid \Gamma \;\vdash\; e_i \rightsquigarrow e_i' \qquad x \in \Delta, x \neq f_b}{\Delta \mid \Gamma \;\vdash\; \text{match}\ x\ \{\overline{p_i \to e_i}\} \rightsquigarrow \text{match}\ x\ \{\overline{p_i \to e_i'}\}}\ \text{bmatch}$$

Figure 6.6: Borrowing resource calculus λ^{1b}

only one argument. The `blam` rule is also a generalization of the `lam` rule in the same spirit, but we also move the free variables Γ into the borrowed environment. This will help us later to reason about the heap semantics. Furthermore, we generalize the `match` rule to allow us to also match on borrowed values.

While this calculus is quite general, it relies on new primitives (sets) in our syntax and copying function code for several different borrowing configurations will only be useful in very narrow circumstances. Next we consider how we can use just a single borrowing configuration.

6.3 Wrapping

The `bapp` rule of Figure 6.5 can invoke any expression with any amount of borrowing. This is useful, since it allows us to pass arguments with ownership if we have it and as borrowed else. But in practice, we want to define functions only for one combination of borrowed and owned parameters. However, then we can not invoke any expression with any amount of borrowing. Instead, an expression needs to carry the borrowing configuration that it can be applied to.

For simplicity, we allow only variables to carry this information. In the `bapp` rule of Figure 6.6 we then have to use the borrowing configuration b that f carries (f_b). In the `blam` rule, we define and immediately bind a lambda expression. A borrowing configuration b can be chosen that marks some variables as borrowed and then f an be used in the expression e_2 with that configuration f_b.

We have to pay attention that variables do not lose their configuration when they are passed to other functions. Functions that appear as a parameter are assumed to accept all their arguments as owned. When we pass a function f_b to a higher-order function we thus have to wrap it into a function that takes all its arguments as owned. For this it simply calls f_b and drops all borrowed arguments of f_b later (in the `wrapping` rule). We assume that functions in the free variables are stored in a way that allows them to be called with their borrowing configuration intact.

Otherwise the calculus is largely unchanged. We only modified the `match` rule to dup the bound variables directly, so we do not have to drop x. This creates a nice parallel to the `bmatch` rule that also does not have to drop its arguments and allows us to use the same heap semantics for both: One can simply remove the dups and drops from $match_r$.

Remark 6.1. While it is a sensible default to mark functions passed as higher-order parameters as owned-only, it may be interesting to allow other defaults. For example, we could allow constructors to contain borrowing functions or pass them as higher-order parameters. But extending our calculus with this functionality seems unwise: we have to track the borrowing vectors as a form of type that adds clutter to the rules. Instead it would be better to check this during a type inference stage and insert `cast` tokens so that the borrowing vectors of argument and parameter match. The λ^{1b} calculus can then transform them back to pure lambda calculus using the `cast` rule in 6.7.

$$\frac{\Gamma_1 = b'\overline{x} \setminus b\overline{x} \qquad \Gamma_2 = (1-b')\overline{x} \setminus (1-b)\overline{x}}{\Delta \mid f_b \;\vdash\; \mathrm{cast}_{b'}(f_b) \rightsquigarrow \lambda_{b'}\overline{x},\ \mathrm{dup}\ \Gamma_1;\ \mathrm{val}\ z = f_b\ \overline{x};\ \mathrm{drop}\ \Gamma_2;\ z}\ \text{casting}$$

Figure 6.7: Casting rule

6.4 When to borrow?

In the last section we have not discussed how the borrowing vector b should be chosen for a given function. In our implementation, we have decided to allow the programmer to choose this vector and we would like to give some heuristics in this section.

Ullrich and de Moura [Ud19] point out that borrowing should not inhibit reuse optimization or tail call optimization, since these are more powerful optimizations that can improve program performance more than borrowing can. Reuse optimization is inhibited when we borrow a parameter that could have been reused else (or of which a child could have been reused else). Tail call optimization can be inhibited if we create new values and pass them to a recursive call in a borrowed position. Ullrich and de Moura give the following example:

```
fun f(x)
  match x
    A(r) -> r
    _ ->
      val y1 = B;
      val y2 = A(y1)
      f(y2)
```

Here, `f(ys2)` should be a tail-call, but if we borrow `x` then it is not, because `y2` has to be dropped afterwards.

But even when no other optimizations get less effective as a result, borrowing can be quite tricky to apply well. For most function calls the cost of incrementing and decrementing the reference counts of its arguments will be negligible compared to the cost of running the function. In the cases where the function is so small that the cost of reference counting would be significant, the function can usually be inlined by the compiler. However, internal functions (implemented in C) and recursive functions can not be inlined and can profit from borrowing.

The `nqueens` benchmark (Figure 6.8) computes the number of possible placements of queens on a chessboard such that all queens are safe from attacks from other queens. For this we compute inductively a list of possible solutions for an $n \times n$ chessboard and then enumerate all solutions for an $(n+1) \times (n+1)$ chessboard that arise by adding a single queen to an existing solution. In such a solution, a queen has a fixed column and can be represented by its row number. It is safe from other queens in the solution, if it is not in the same row and not on the same diagonal:

```
fun safe( queen : int, diag : int, xs : solution ) : bool
  match xs
    Cons(q,qs) -> (queen != q && queen != (q+diag)
      && queen != (q - diag) && safe(queen,diag+1,qs))
    _          -> True
```

We compare Kokas performance on this benchmark against other implementations. For this we consider 6 variants. The 'Koka, int' variants use arbitrary-precision integers that are reference counted, while the 'Koka' variants use 32-bit integers that are passed by value. The 'no borrow' variants do not use any borrowing, while the plain variants use borrowing only for internal functions. Finally the 'borrow safe' variants borrow the parameter `xs` of the `safe` function. Borrowing `xs` is a good choice here as it does not inhibit any other optimizations and other function in the benchmark (not shown here) never pass their last reference of `xs` to `safe`.

As expected there is little difference between borrowing internal functions or not for 32-bit integers that do not profit from it. Borrowing `xs` has a positive effect for both variants, but the effect is much bigger for the 32-bit variant: The arbitrary-precision variant still has to handle dup and drop operations on `queen` and `diag` while the 32-bit variant has to handle no other reference count instructions. We believe that this allows the C compiler to compile the code into a tight loop that makes better use of registers.

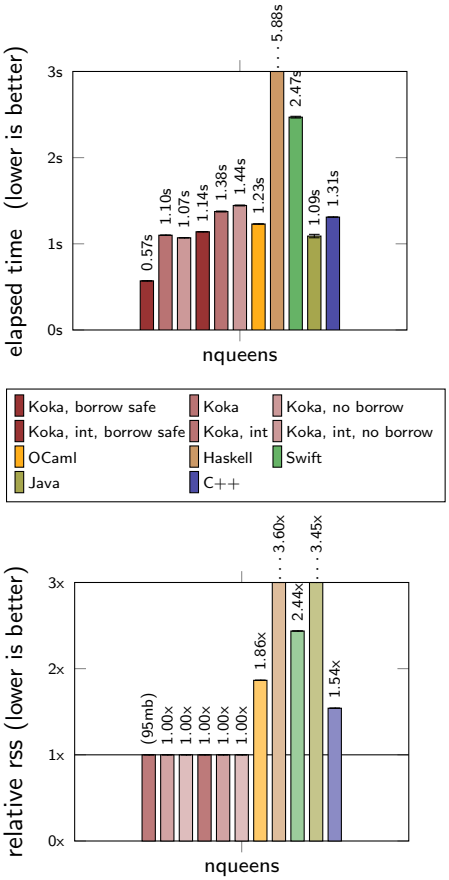

Figure 6.8: Benchmark results for nqueens, Apple M1

6.5 Lean's borrow inference

Ullrich and de Moura [Ud19] describe a borrow inference for Lean. We present an adaption for Koka in Figure 6.9 where we write $e \gg S$ to mean that the variables in

$$\frac{}{x \gg \varnothing}\ \text{var} \qquad \frac{}{C\,x_1 \ldots x_n \gg \varnothing}\ \text{cons} \qquad \frac{}{C@r\,x_1 \ldots x_n \gg \varnothing}\ \text{cons-reuse}$$

$$\frac{}{f_b\,\overline{x} \gg (1-b)\overline{x}}\ \text{bapp} \qquad \frac{}{f\,y \gg \{f, y\}}\ \text{app} \qquad \frac{}{\lambda\overline{x},\, e \gg \text{fv}(\lambda\overline{x},\, e)}\ \text{lam}$$

$$\frac{e \gg S}{\text{val}\ r = \text{drop-reuse}(x);\ e \gg S \cup \{x\}}\ \text{drop-reuse} \qquad \frac{e_1 \gg S_1 \qquad e_2 \gg S_2}{\text{val}\ x = e_1;\ e_2 \gg S_1 \cup S_2}\ \text{val}$$

$$\frac{e_i \gg S_i \qquad \exists i,\, p_i \cap S_i \neq \varnothing}{\text{match}\ x\ \{\overline{p_i \to e_i}\} \gg \bigcup_i S_i \cup \{x\}}\ \text{match} \qquad \frac{e_i \gg S_i \qquad \forall i,\, p_i \cap S_i = \varnothing}{\text{match}\ x\ \{\overline{p_i \to e_i}\} \gg \bigcup_i S_i}\ \text{bmatch}$$

Figure 6.9: Lean's Borrow Inference

S should be marked as *owned* based on the evidence gathered in e. In particular, they mark parameters as owned if they are again passed as owned to a function or reused to make sure that no reuse opportunities are obstructed based on borrow inference. They do not describe the `lam` and `val` rules explicitly as their internal calculus is A-normalized, but we believe that these rules are accurate representations of Lean's behaviour on the A-normalizations of these constructs.

However, this inference made some programs slower, both in their benchmarks as well as our experience. We believe that a significant part of this slowdown can be found in the fact that Lean's borrow inference can hold on to memory for much longer than necessary. This is especially problematic if a program allocates memory at the same time, since the peak memory consumption can grow significantly in this case.

In section 3.7 we already gave an example where borrowing makes performance worse. If we borrow the parameter `xs` of `make-tree` we will free the list `xs` too late:

```
fun make-tree(xs)
  match(xs)
    Cons(x,xx) -> Bin(x, Tip, make-tree(xx))
    Nil -> Tip
```

This has two bad effects: First, the peak memory usage is twice as high as it needs to be. Second, performance will be worse because we need to request lots of new memory from the OS and we need to free `xs` in a separate pass. Since fetching memory is much slower than executing instructions on a modern CPU, it is much better to free the `Cons` nodes as we traverse the list in `make-tree` than to consider the entire list twice. Unfortunately, the borrow inference in Figure 6.9 would mark `xs` as borrowed. In the next chapter, we will discuss a criterion for deciding whether a variable should be borrowed based on the worst-case increase in memory usage.

Chapter 7

Frame-limited transformations

While with Perceus a program will use the least amount of heap space possible, we need to use slightly more to enable useful performance optimizations. For example, a pair of a dup and a drop on the same variable can be eliminated if they occur directly after another. But is it wise to move dups and drops to be able to eliminate some of them? A common pattern in Perceus's output is the following:

```
match x
  X(y, z) ->
    dup(y); drop(x);
    val a = ...
    drop(y);
    ...
```

We could easily eliminate the dup and drop on `y` if we moved the drop on `x` behind the definition of `a`: `y` only needs to be borrowed here and we know that `x` holds on to it, so we can move the dup on `y` behind the definition of `a` as well and eliminate it with the drop. But then we will drop `x`, and by extension `z`, late and we might use an arbitrary amount of heap space more if `a` and `z` are big.

The central insight of this chapter is that we can model both reuse analysis and borrowing as a question of moving dups and drops around. This can itself be modelled using just a simple addition to the simplified λ^1 calculus (Figure 7.1): In the normalized calculus, we can move a drop behind a val-binding by moving its variable from the Γ_1 environment to the Γ_2 environment. Whether this move is a good idea can then be decided using the $(\star)$ condition. In this chapter we first discuss how reuse and borrowing can be modelled in the simplified λ^1 calculus and then discuss possible $(\star)$ conditions.

7.1 Modelling reuse analysis and borrowing

Even though borrowing was quite difficult to describe, we can model it easily if we only want to consider its impact on heap usage. Assume we have a borrowing function call $f_b\ \overline{x}$. We will model this in our calculus as a function that takes all its arguments as

$$\frac{}{\Delta \mid x \vdash x \rightsquigarrow x}\ \text{var} \qquad \frac{\Delta, x \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e\ x \rightsquigarrow e'\ x}\ \text{app}$$

$$\frac{\Delta \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \rightsquigarrow \text{dup } x;\ e'}\ \text{dup} \qquad \frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \text{drop } x;\ e'}\ \text{drop}$$

$$\frac{\Delta \mid \Gamma, r \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \text{drop-reuse } x;\ e'}\ \text{drop-reuse}$$

$$\frac{}{\Delta \mid x_1 \ldots x_n \vdash C\, x_1 \ldots x_n \rightsquigarrow C\, x_1 \ldots x_n}\ \text{con}$$

$$\frac{\varnothing \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad \Gamma = \text{free variables of } \lambda x,\ e}{\Delta \mid \Gamma \vdash \lambda x,\ e \rightsquigarrow \lambda^{\Gamma} x,\ e'}\ \text{lam}$$

$$\frac{\Delta \mid \Gamma, \text{bv}(p_i) \vdash e_i \rightsquigarrow e_i' \qquad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash \text{match } x\ \{\overline{p_i \to e_i}\} \rightsquigarrow \text{match } x\ \{\overline{p_i \to \text{dup bv}(p_i);\ e_i'}\}}\ \text{match}$$

$$\frac{x \notin \Delta, \Gamma_1, \Gamma_2 \qquad \Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2, x \vdash e_2 \rightsquigarrow e_2' \qquad (\star)}{\Delta \mid \Gamma_1, \Gamma_2 \vdash \text{val } x = e_1;\ e_2 \rightsquigarrow \text{val } x = e_1';\ e_2'}\ \text{val}$$

Figure 7.1: Simplified λ^1 calculus

owned, but where we *need* to insert a dup for every variable in $b\overline{x}$. Then we have at least one copy of these variables in the Γ set after the function call and, if this was the last use of the variable, have to drop it. But the simplified λ^1 calculus does not allow us to place a drop directly behind a function call: We can only place it behind the function call if we use a val-binding. As such, we can decide whether a function should borrow by checking if all its *application sites* can be derived using the λ^1 calculus with an appropriate $(\star)$-rule.

We model reuse analysis by defining `drop-reuse` and `@r` as:

$$\text{val } r = \text{drop-reuse } x;\rightsquigarrow \text{drop } x;\ \text{val } r = C\bot_1 \ldots \bot_n;\ \text{drop } r;$$
$$C@r \rightsquigarrow \text{drop } r;\ C$$

In other words: Instead of reusing a constructor, we allocate a new one of the same size with no children and drop it immediately. We can then decide using the $(\star)$-rule if it is feasible to push the drop down to the constructor where the reuse would be applied. This scheme yields the same heap structure as reuse analysis and so we can transfer facts about the one to the other.

7.2 Garbage-free star-rule

If we set $(\star) = \Gamma_2 \subseteq \text{fv}(e_2)$, we obtain a calculus that is very similar to the Perceus algorithm. Unlike the Perceus algorithm, the simplified λ^1 calculus does not prescribe

where a drop should be placed precisely and that makes it possible that memory is held on longer than necessary. For example, we can derive:

```
val x = list(1, 100) // unused binding
match y
  A -> drop(x); ...
  B -> drop(x); ...
```

Here, x is freed only after the match and not before (as Perceus would). But this momentary increase in the memory usage compared to Perceus seems rather insignificant, since no allocations can happen while we hold on to x. This can be seen in the rules of the calculus: Allocations can happen in the con, lam and app rules but none of these rules allows you to place a drop directly behind them. Rather, the only way to place a drop behind them is through the val rule, where the drop can happen at the start of e_2'. A step in the evaluation where an allocation happens, can be written as $\mathrm{E}[v]$ (as then the lam_r or con_r rule is executed next). With this it can be formally proved that:

Theorem 7.1 (Garbage-free)**.** If $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$, $(\star) = \Gamma_2 \subseteq \mathrm{fv}(e_2)$ and $\varnothing \mid e' \longrightarrow_r \mathrm{H} \mid x$ then for every intermediate state $\mathrm{H}_i \mid \mathrm{E}[v]$ all $y \in \mathrm{dom}(\mathrm{H}_i)$ are reachable at $\mathrm{H}_i \mid \lceil \mathrm{E}[v] \rceil$.

A proof of this claim can be found in [LL21]. While this is slightly weaker than the notion of garbage-freeness we introduced in chapter 5, this still guarantees that the heap can not grow while dead memory remains in it. Consequently, the peak heap usage can not increase.

7.3 Frame-limited star-rule

Reuse analysis holds on to a single cell in memory in order to reuse it later. This means that the memory it holds on to will be small; in particular there is a constant c_1 such that $\mathrm{sizeof}(r) \leq c_1$ for any reuse token r. As there is only a constant amount of reuse opportunities per function, we can allow reuse in λ^1 by setting $(\star)$ as $\Gamma_2 = \Gamma', \Gamma''$, $\Gamma' \subseteq \mathrm{fv}(e_2)$, $\mathrm{sizeof}(\Gamma'') \leq c$. The resulting evaluation is not garbage-free, but the total size of the heap is still bounded: Any time we allow Γ'' to remain live, we add a new val-frame to our evaluation context.

Theorem 7.2 (Frame-limited)**.** If $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$, $(\star)$ as above and $\varnothing \mid e' \longrightarrow_r \mathrm{H} \mid x$ then for every intermediate state $\mathrm{H}_i \mid \mathrm{E}[v]$, there is a partition $\mathrm{H}_i^1, \mathrm{H}_i^2$ of H_i such that all $y \in \mathrm{dom}(\mathrm{H}_i^1)$ are reachable at $\mathrm{H}_i \mid \lceil \mathrm{E}[v] \rceil$ and $|\mathrm{H}_i^2| \leq c \cdot |\mathrm{E}|$.

Again, a proof of this claim can be found in [LL21]. We might hope to be able to bound this extra increase by a constant factor, but that is usually not possible: For all we know e_1 in the val-rule might contain a recursive call or a call to an anonymous function that we can only know at runtime (and that may contain a similar optimization). To be able to bound the increase by a constant factor, we would need to ensure that at any given moment only a constant amount of function calls on the stack use an optimization like this.

In fact, reuse analysis can use more than a constant amount of extra memory. The following program uses `iter`-times more heap space with reuse than without:

```
fun big-list()
  list(1, 10000000)

fun go(n, t)
  match(t)
    Cons(_, xx) ->
      val y = go(n, xx)
      Cons(1, y)
    Nil | n >= 1 ->
      go(n - 1, big-list())
      Nil
    Nil -> Nil

fun main()
  val iter = 10
  go(iter, big-list())
```

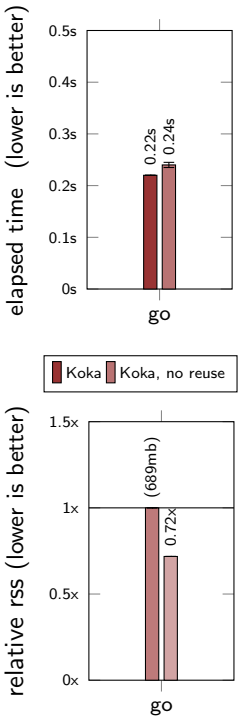

Figure 7.2: Benchmark results for 'go', Apple M1

Without reuse analysis `go` will discard the list `t` as it traverses it and will have freed the list once it creates a new one in the second branch (Koka does not perform TRMC in the first branch since the call to `go` in the second branch is not in tail position). With reuse analysis every `Cons` constructor is kept live in the first branch and so the list is still in memory when we allocate a new one. However, this analysis omits that we will use even more stack space. The benchmark in Figure 7.2 illustrates this: The rss (which also includes stack size) is significantly larger for Koka with reuse than without. But the increase is much smaller than one might expect and the program is still faster with reuse analysis.

Remark 7.1. We have used the evaluation context to model the stack above. This is an accurate model only if no tail-call modulo cons optimization takes place: TRMC replaces a function call with a jump which leads to fewer stack frames. It also moves an allocation from *after* the recursive call to *before*. As such it can itself increase the heap space by a constant factor every time a jump happens. It would be possible to study TRMC in the framework of this chapter as well by relaxing the notion of 'frame' to also include TRMC-jumps.

7.4 Peak frame-limited star-rule

We can combine the ideas of the two previous rules to obtain a rule that works well for borrowing. We first saw the argument that it is fine to hold on to an arbitrary amount

of memory a few moments longer as long as we do not allocate. We then saw that we could allow a constant amount of dead memory (per stack-frame) while allocating an arbitrary amount. Dually, we can also allow an arbitrary amount of dead memory while allocating only a constant amount. This is important for borrowing since many functions only traverse a datastructure and then allocate a bit at the end. For example, we can traverse a list and return the first element that fulfils a predicate:

```
fun lookup( xs : list<a>, pred : a -> bool) : maybe<a>
  match xs
    Cons(x,xx) -> if pred(x) then Just(x) else lookup(xx,pred)
    Nil -> Nothing
```

This function allocates only a single constructor and so we can borrow `xs` without worrying about an increase in memory consumption. Again, we can describe this using a $(\star)$ rule. For this we assume that there is a predicate $\text{cost}(x, e)$ that gives the maximum allocation size in e after x would have been dropped. We write $\text{cost}(\Gamma'', e)$ for the maximum of the costs of $x \in \Gamma''$. We then set $(\star)$ as $\Gamma_2 = \Gamma', \Gamma'',\ \Gamma' \subseteq \text{fv}(e_2),\ \text{cost}(\Gamma'', e_1) \leq c$. The resulting evaluation is not frame-limited, but the peak size of the heap is still bounded: While there might be a large amount of dead memory at any allocation, this memory must have been allocated before and we can only add a constant amount (per stack-frame) to it.

Theorem 7.3 (Peak frame-limited (conjectured))**.** If $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$, $(\star)$ as above and $\varnothing \mid e' \longrightarrow_r \text{H} \mid x$ then for every intermediate state $\text{H}_i \mid \text{E}[v]$, we have $|\text{H}_i| \leq |\text{H}_{max}| + c \cdot |\text{E}|$.

But we can even allow more: We can combine this rule with the frame-limited star-rule to allow dead memory to be either small or of little cost. Then we set $(\star)$ as: $\Gamma_2 = \Gamma', \Gamma'', \Gamma''',\ \Gamma' \subseteq \text{fv}(e_2),\ \text{sizeof}(\Gamma'') \leq c,\ \text{cost}(\Gamma''', e_1) \leq c$. We conjecture that this rule is again peak frame-limited.

Unfortunately, we can not describe the predicates sizeof and cost in our calculus: The untyped lambda calculus introduced in chapter 5 allows unbounded recursion in too many places and it is thus difficult to bound the amount of allocations happening in a single function. We will therefore describe our rules for borrow inference only informally in a calculus that is closer to Koka: Simply typed lambda calculus with mutually recursive functions.

The sizeof predicate can be computed using the algebraic notation introduced in section 4.2. We use t_1, t_2 to refer to parts of a type and define inductively:

$$\begin{aligned}
\text{sizeof}(1) &= 1 \\
\text{sizeof}(a) &= \infty \\
\text{sizeof}(t_1 + t_2) &= \max\{\text{sizeof}(t_1), \text{sizeof}(t_2)\} \\
\text{sizeof}(t_1 \cdot t_2) &= \text{sizeof}(t_1) + \text{sizeof}(t_2) \\
\text{sizeof}(\mu x.t_1) &= \infty
\end{aligned}$$

In other words, any type that is parameterized over an unknown type a or refers to a recursive type must be assumed to have infinite size. Only types that are composed of other small types like booleans or ints can be assumed to be small themselves.

To define the cost predicate, we call for a given expression e a tuple m that contains exactly one branch of every `match`-statement in e a *branch selection* of e. We define $e[m]$ to be the expression that arises from e if we replace each match statement with the branch chosen in m. A *path* through e is a member of the equivalence class (w.r.t. $e[m]$) of branch selections. We can define $\text{cost}(x, e)$ as the most expensive cost of a path through e. The cost of a path $e[m]$ is zero if x is not dropped on that path and else $e[m]$ contains a subexpression drop $x; e'$ and we define the cost as the size of the constructors in e' plus $\text{cost}(f)$ for every function call f in e'.

For a recursion-less function its cost is the cumulative size of the allocated constructors and the cost of the functions it calls on its most expensive path. For a self-recursive function its cost is infinite if there is a path that contains a recursive call and allocates or calls functions with cost not zero. Else, its cost is its most expensive path in the sense of recursion-less functions. For a mutually recursive function its cost is infinite if there is a path that contains a call to its partner and allocates (directly or through a function call). Else, its cost is determined as if it was self-recursive.

7.5 Conclusion

In this chapter we have seen how we can make a borrow inference that does not increase the peak memory usage of a program too much and thus can not lead to space leaks. We can pair this with the rule that borrowing should not inhibit tail calls or reuse analysis (see section 6.4). But is that enough? We have found that borrowing can still make some programs slower, even in cases were no extra memory is held on to. For example the `lookup` function from the last section becomes slower through borrowing, even if it does not hold the last reference to `xs`. However, we believe that this is related to more low-level optimizations like prefetching, that can be addressed separately from borrow inference. We expect that further research will be able to give a borrow inference that can make programs faster with little theoretic or practical downside.

Bibliography

[Arm21] Arm Ltd. *ARMv8-A Memory systems: The memory model*. 2021. URL: https://developer.arm.com/documentation/100941/0100/The-memory-model?lang=en.

[Carp21] The Carp Community. *Carp Documentation: Memory management - a closer look*. 2021. URL: https://github.com/carp-lang/Carp/blob/master/docs/Memory.md.

[Col60] George E Collins. 'A method for overlapping and erasure of lists'. In: *Communications of the ACM* 3.12 (1960), pp. 655–657.

[Con21] Contributors. *Binary Trees - The Computer Language Benchmarks Game*. 2021. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html.

[Cor+09] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[cpp21] cppreference. *std::memory_order*. 2021. URL: https://en.cppreference.com/w/cpp/atomic/memory_order.

[CST18] Jiho Choi, Thomas Shull and Josep Torrellas. 'Biased reference counting: Minimizing atomic operations in garbage collection'. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–12.

[DB76] L Peter Deutsch and Daniel G Bobrow. 'An efficient, incremental, automatic garbage collector'. In: *Communications of the ACM* 19.9 (1976), pp. 522–526.

[Fla+93] Cormac Flanagan et al. 'The essence of compiling with continuations'. In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 1993, pp. 237–247.

[FW75] D Friedman and S Wise. 'Unwinding stylized recursions into iterations'. In: *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep* 19 (1975).

[Gal16] Matt Gallagher. 'Reference Counted Releases in Swift'. Blog post. Dec. 2016. URL: https://www.cocoawithlove.com/blog/resources-releases-reentrancy.html.

[GS78] Leo J Guibas and Robert Sedgewick. 'A dichromatic framework for balanced trees'. In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE. 1978, pp. 8–21.

[HB05] Matthew Hertz and Emery D Berger. 'Quantifying the performance of garbage collection vs. explicit memory management'. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, pp. 313–326.

[Hue97] Gérard Huet. 'The zipper'. In: *Journal of functional programming* 7.5 (1997), pp. 549–554.

[HW67] Bruce K Haddon and William M Waite. 'A compaction procedure for variable-length storage elements'. In: *The Computer Journal* 10.2 (1967), pp. 162–165.

[Int15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: 8.2 Memory Ordering*. 2015. URL: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-volume-3a-system-programming-guide-part-1.html.

[JM02] Simon Peyton Jones and Simon Marlow. 'Secrets of the glasgow haskell compiler inliner'. In: *Journal of Functional Programming* 12.4-5 (2002), pp. 393–434.

[LBM19] Daan Leijen, Zorn Ben and Leo de Moura. 'Mimalloc: Free List Sharding in Action'. In: *Programming Languages and Systems*. LNCS 11893 (2019). APLAS'19. DOI: 10.1007/978-3-030-34175-6_13.

[LH83] Henry Lieberman and Carl Hewitt. 'A real-time garbage collector based on the lifetimes of objects'. In: *Communications of the ACM* 26.6 (1983), pp. 419–429.

[LL21] Anton Lorenzen and Daan Leijen. 'Reference Counting with Frame Limited Reuse'. In: *MSR-TR-2021-30* (2021).

[Mau+17] Luke Maurer et al. 'Compiling without continuations'. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 482–494.

[McB01] Conor McBride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*. (Extended Abstract). 2001. URL: http://strictlypositive.org/diff.pdf.

[McC60] John McCarthy. 'Recursive functions of symbolic expressions and their computation by machine, part I'. In: *Communications of the ACM* 3.4 (1960), pp. 184–195.

[Mok17] Andrey Mokhov. 'Algebraic Graphs with Class (Functional Pearl)'. In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 2–13. ISBN: 9781450351829. DOI: 10.1145/3122955.3122956.

[Moo84] David A Moon. 'Garbage collection in a large Lisp system'. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. 1984, pp. 235–246.

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[PP03] Gordon Plotkin and John Power. 'Algebraic operations and generic effects'. In: *Applied categorical structures* 11.1 (2003), pp. 69–94.

[PP09] Gordon Plotkin and Matija Pretnar. 'Handlers of algebraic effects'. In: *European Symposium on Programming*. Springer. 2009, pp. 80–94.

[Rei+21] Alex Reinking et al. 'Perceus: Garbage Free Reference Counting with Reuse'. In: *Proceedings of the 42rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '21. Pittsburgh, USA: ACM, 2021.

[SBB21] Andrey Semashev, Tim Blechmann and Helge Bahmann. *Boost documentation: Reference counting*. 2021. URL: https://www.boost.org/doc/libs/1_77_0/doc/html/atomic/usage_examples.html#boost_atomic.usage_examples.example_reference_counters.

[SF98] Jonathan Sobel and Daniel P Friedman. 'Recycling continuations'. In: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. 1998, pp. 251–260.

[SHM20] Daniel Selsam, Simon Hudon and Leonardo de Moura. 'Sealing Pointer-Based Optimizations behind Pure Functions'. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408997. URL: https://doi.org/10.1145/3408997.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. 'Self-adjusting binary search trees'. In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686.

[SW67] H. Schorr and W. M. Waite. 'An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures'. In: *Commun. ACM* 10.8 (Aug. 1967), pp. 501–506. ISSN: 0001-0782. DOI: 10.1145/363534.363554. URL: https://doi.org/10.1145/363534.363554.

[Ud19] Sebastian Ullrich and Leonardo de Moura. 'Counting Immutable Beans – Reference counting optimized for purely functional programming'. In: *Proceedings of the 31st symposium on Implementation and Application of Functional Languages (IFL'19)*. Singapore, Sept. 2019.

[Ung84] David Ungar. ‘Generation scavenging: A non-disruptive high performance storage reclamation algorithm’. In: *ACM Sigplan notices* 19.5 (1984), pp. 157–167.

[Wad90] Philip Wadler. ‘Linear types can change the world!’ In: *Programming concepts and methods*. Vol. 3. 4. University of Glasgow. 1990, p. 5.

[XL21] Ningning Xie and Daan Leijen. ‘Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C’. In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–30.