Technical Report

UCAM-CL-TR-908
ISSN 1476-2986

Number 908

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

ASAP: As Static As Possible memory management

Raphaël L. Proust

July 2017

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 2016 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Magdalene College.

ISSN 1476-2986

Abstract

Today, there are various ways to manage the memory of computer programs: garbage collectors of all kinds, reference counters, regions, linear types – each with benefits and drawbacks, each fit for specific settings, each appropriate to different problems, each with their own trade-offs.

Despite the plethora of techniques available, system programming (device drivers, networking libraries, cryptography applications, etc.) is still mostly done in C, even though memory management in C is notoriously unsafe. As a result, serious bugs are continuously discovered in system software.

In this dissertation, we study memory management strategies with an eye out for fitness to system programming.

First, we establish a framework to study memory management strategies. Often perceived as distinct categories, we argue that memory management approaches are actually part of a single design space. To this end, we establish a precise and powerful lexicon to describe memory management strategies of any kind. Using our newly established vocabulary, we further argue that this design space has not been exhaustively explored. We argue that one of the unexplored portion of this space, the static-automatic gap, contributes to the persistence of C in system programming.

Second, we develop ASAP: a new memory management technique that fits in the static-automatic gap. ASAP is fully automatic (not even annotations are required) and makes heavy use of static analysis. At compile time it inserts, in the original program, code that deallocates memory blocks as they becomes useless. We then show how ASAP interacts with various, advanced language features. Specifically, we extend ASAP to support polymorphism and mutability.

Third, we compare ASAP with existing approaches. One of the points of comparison we use is the behavioural suitability to system programming. We also explore how the ideas from ASAP can be combined with other memory management strategies. We then show how ASAP handles programs satisfying the linear or region constraints. Finally, we explore the insights gained whilst developing and studying ASAP.

Résumé

De nombreuses techniques sont disponibles pour gérer la mémoire des programmes informatiques : ramasse-miettes, compteur de références, système de régions, système de types linéaires. Chaque méthode possède ses propres avantages et inconvénients, chaque méthode se prête plus ou moins bien à différents programmes.

Malgré l'abondance de techniques disponibles, le code système (pilotes de périphériques, implémentations de protocoles de communication, logiciels de chiffrement, etc.) est encore et toujours écrit en C. Ceci, bien que C soit notoirement dangereux en matière de gestion de la mémoire. De ce fait, on découvre régulièrement, dans le code système, d'innombrables bogues.

Dans cette dissertation, nous étudions la gestion de la mémoire dans le contexte du code système.

Tout d'abord, nous établissons un lexique précis pour décrire les techniques de gestion de la mémoire. Parce que ce lexique s'applique à la multitude de techniques connues qui, jusqu'à présent, étaient considérées comme distinctes, nous défendons l'idée que ces techniques font partie d'un même ensemble. L'étude de cet ensemble révèle des lacunes dont l'une, que nous nommons le « déficit statique-automatique, » nous intéresse plus particulièrement. Et nous défendons l'idée que le déficit statique-automatique contribue à la persistance de C dans le domaine du code système.

Dans un deuxième temps, nous présentons ASAP : une nouvelle technique de gestion de la mémoire qui comble le déficit statique-automatique. ASAP analyse les programmes afin d'y insérer, pendant la compilation, des instructions qui, pendant l'exécution, libèrent les blocs de mémoire au moment opportun. Nous étendons ensuite ASAP pour lui permettre de gérer la mémoire de programmes avec polymorphisme et mutation.

Dans un troisième temps, nous comparons ASAP aux autres techniques de gestion de la mémoire et, lorsque c'est possible, nous présentons des solutions hybrides qui empruntent certaines idées d'ASAP et les intègrent à ces autres méthodes. Nous observons ensuite la gestion, par ASAP, de la mémoire de programmes linéaires et à région. Enfin, nous étudions les liens qui existent entre, d'une part, la gestion de la mémoire en général et ASAP en particulier et, d'autre part, la gestion des ressources dans d'autres domaines de l'informatique.

Preface

This dissertation is organised as follows:

Part A:

Chapter 1: We discursively lay out the context of our work: memory management and system programming. We focus on the interactions between these two domains.

Chapter 2: We provide the necessary technical background – mathematical, notational, lexical and otherwise. Note that this chapter essentially collects concepts published by others.

Part 1: Design Space

Chapter 3: We explore the design space of memory management strategies. To that end, we develop an original lexicon which lets us describe existing memory management strategies with an informative point of view.

Part 2: Asap

Chapter 4: We formally define paths: compile-time descriptions of heap structures. We use them to approximate heap structures in a bounded-size fashion during analyses. We also use them to synthesise code that scans through the described heap structures during execution. Note that, whilst the formalisation detailed in this dissertation is our own, it is influenced by the work of Khedker, Sanyal and Karkare [24].

Chapter 5: We present asap, a new automatic memory management strategy. We give full technical details, from analysis to code transformation.

Chapter 6: We introduce new language features, namely: mutability and polymorphism. These additions raise a number of challenges which are listed and addressed one by one.

Chapter 7: We briefly describe our prototype implementation of asap.

Part 3: Recontextualisation

Chapter 8: We revisit the design space of memory management strategy, with a focus on asap. Specifically, we compare asap with other approaches. Additionally, we show how to hybridise asap and other approaches together.

Chapter 9: We show that asap subsumes both linear type systems and region-based memory management. That is, when given a linear or region-based program, asap emits deallocation instructions similar to the ones generated by the linear or region-based memory manager.

Chapter 10: We explore insights from the comparisons and subsumptions mentioned above. These insights range over other forms of resources managements such as registers and system resources (e.g., file descriptors).

Part Ω:

Chapter 11: We collect concluding remarks.

Contents

Chapter 1

Introduction

Computer Science can be described as the art of designing, implementing and composing abstractions. These abstractions bridge a gap between fast and precise but dumb processing machines (also known as computers) on the one hand and slow and error-prone but meta-cognisant organisms (also known as humans) on the other hand. With no abstractions, it is very difficult for humans to make use of computers.

This description is especially true of Programming Language research which provides a specific type of abstractions called *programming languages* to a specific class of humans called *programmers*. A programming language is a set of abstractions and tools that can be used to create programs. A programming language is said to be low-level if its abstractions are few and expose the specificities of the underlayer. Conversely, a programming language is said to be high-level if its abstractions are many and hide entirely their underlayer.

In this dissertation we are concerned with one abstraction in particular: memory. At a low level of abstraction, memory is a sequence of bits that can hold either of two values: 0 or 1. Programming languages provide abstractions on top of this low level view. These abstractions let programs and programmers store and retrieve values from the memory: the values are encoded into sequences of 0 and 1 which are copied onto and loaded from the memory. One aspect of this process of storing and retrieving values is to decide where in the memory – i.e., at what offset of the sequence of bits – is any particular value stored. This aspect is further complicated by time-sharing: the concurrent execution of several programs on a single machine. Which program uses what part of the memory? How do programs reserve a section of the memory for their own use? This bookkeeping, both within a program to decide where each value is stored and between programs to decide which one has exclusive access to what part of the memory, is called *memory management*.

We study memory management: how programs and programmers reserve and release (allocate and deallocate) the memory of the computer during the execution of a program. Even more specifically, this dissertation introduces ASAP, a new method to deallocate memory where the *compiler* is the main actor. With ASAP, memory is entirely abstracted away from programmers who only need to deal with values, not their storage. In that respect, ASAP differs from C (where programmers manage storage) and feels more like a garbage collector (i.e., it is fully automatic). However, it differs from garbage collectors in that the compiler (rather than the runtime) manages storage.

We argue that ASAP possesses a unique combination of qualities. We further argue that this unique combination makes ASAP a prime option for managing the memory of a specific category of programs: system software.

$$\begin{tikzcd}
\boxed{\texttt{source code}} \arrow[r] & \text{parse tree} \arrow[r] & \mathrm{IR}_1 \arrow[r] & \cdots \arrow[r] & \mathrm{IR}_n \arrow[r] & \boxed{\texttt{binary code}}
\end{tikzcd}$$

Figure 1.1: A compiler pipeline

1.1 Compilers

Compilers transform code. They start from a high-level form readable and writeable by humans. They produce a low-level form executable by machines. For engineering purposes, compilation happens in several stages: from source code (a stream of characters) into an abstract syntax tree into a series of intermediate representations (IR) into a binary executable format (a stream of bytes or bits). This pipeline view of compilers is presented in Figure 1.1

The initial part of the compiler pipeline parses the source code and lightly transforms the result. It is called the *front-end*. The central part is responsible for the bulk of the transformation and is called the *middle-end*. It uses a series of IRs: a gradient of languages between the source and target languages. The terminal part of the pipeline is called the *back-end* and is responsible for the transformations that are specific to the target architecture.

In order to translate a program from one IR to the next, the compiler needs information about the program. The compiler runs analyses on the code to gather this information. Some analyses exist to enforce specific correctness criteria. These *type-like* analyses return a simple yes-or-no value which determines whether the compilation should continue or not. E.g., type checking is a type-like analysis: it distinguishes type-correct programs that the compiler processes and type-incorrect programs that the compiler rejects. Other analyses merely inform the next phase of the compilation. E.g., compilers analyse the liveness of variables to safely choose which registers to coalesce.

1.2 Memory management

Memory management is the set of techniques and abstractions used by programmers to manage storage for a program's values. Different approaches, each with their own advantages and drawbacks, are used in different programming languages. We list here approaches that are widely deployed or otherwise interesting to our study.

1.2.1 Thin abstraction *à la* C

In low-level languages it is the programmers' responsibility to allocate and deallocate memory. In C specifically, the only tools available are the five functions `malloc`, `free`, `calloc`, `realloc` and `alloca`. These offer a thin abstraction – compared to the approaches listed below.

The main advantage of the C approach is that it offers prodigious control to the programmers. However, its main advantage brings about its greatest weakness: it places the full burden of responsibility on the programmers[1].

Often, programmers promote C-like approach by invoking control over the timing of deallocation. However, an arguably more important aspect of this control concerns the layout of values. Specifically, C does not impose any restrictions on how data is represented in memory: the programmers decide, say, whether a nested struct is represented inline or as a pointer to a separate

[1]As Benjamin Parker remarked "With great power there must also come great responsibility!"

block. With this feature, programmers can decide to represent TCP segments in memory with the same sequence of bits as on the wire or in the air.

Whilst it provides a lot of control over the program execution behaviour, C's approach imposes on the programmers the full burden of responsibility. Specifically, the programmers must ensure two correctness criteria:

- the program does not read from nor write to a pointer to unallocated or deallocated space, and
- after values become useless, they are eventually deallocated once.

These criteria guarantee there are no crashes induced by memory management during execution. They also guarantee the program occupies only marginally more memory than is strictly necessary.

1.2.2 Garbage Collection

In most modern programming languages (ML, Java, Go, etc.), the main program, called the *mutator*, is linked with a generic piece of code known as the *garbage collector* (GC). The mutator performs the computations specified by the programmers, all the while effectively leaking memory. Once the heap is full, the mutator cedes control to the GC. The GC systematically explores the heap starting from global values and variables on the stack (known as *roots*) and discards anything that is unreachable. (We discuss variations on this design below.)

Note that unreachability is merely a *proxy* for uselessness: a value that is unreachable (i.e., cannot be accessed) is useless (i.e., will not be accessed). However, this is only an approximation: in most programs, there are many values that are useless but still reachable – as shown by Röjemo and Runciman [34].

The main advantage of GCs is reliability: they have been debugged into correctness, preventing both crashes and memory leaks.

That correctness comes at a cost: loss of control. More specifically, because GCs systematically explore the memory graph, they need to determine the size of memory blocks and distinguish pointers and integers during execution. Thus, GCs rely on *runtime-types*[2] (also known as *tags*) and impose a fixed pattern for the memory representation of values – alternatives, some of which do not restrict layout, are presented below.

Conservative GCs In order to avoid the loss of control over value representation, *conservative* GCs (CGCs), such as the one developed by Hans-Juergen Boehm [9], were created. A conservative GC scans the heap without any information about the layout of the memory. As a result, the conservative GC must make a conservative (hence the name) assumption about the words in memory: each word is treated as a pointer because it might happen to be one. (A GC that is not conservative is called *precise*.)

Whilst CGCs do not restrict value representation[3], they do instead abandon one of the correctness criteria. Specifically, under a CGC, not all memory blocks are reclaimed.

Additionally, CGCs do not support compaction (a technique by which GCs move blocks in the heap so they are closer together). Specifically, under CGCs, it is not possible to move a block and

[2]ML – and the like – are known for their type-erasure semantics: it is not possible at runtime to distinguish an integer (`int`) from a character (`char`). Whilst user-level types are indeed erased, runtime types remain. They are necessary, so that the GC can distinguish arrays from integers from pointers.

[3]In fact, CGCs requires pointers to be word-aligned which is almost always the case in practice.

update the pointers to that block. Indeed, the CGC cannot tell which are pointers to the block and which are word-sized values that happen to correspond to the address of the block.

Tagless GCs *Tagless* GCs [30] are a family of GC which do not use runtime-type information. Instead, tagless GCs reconstruct the necessary type information from hints sprinkled on the execution stack: the type of values stored at each offset of each frame. This information is collected by the compiler and carried through to the final binary for the GC to use.

Tagless GCs are not widely deployed. The current trend in GC development is to make them simpler and faster with a focus on avoiding long pauses. Specifically, there is a preference towards spending a little more resources in the mutator (setting up tag bits and such) if it significantly simplifies the GC and shortens pauses. This is only true to an extent: an important mutator cost is never traded for a small GC simplification. However, the current balance tilts away from tagless GCs.

Reference Counting A variant of GC is *reference counting* (RC): blocks of memory are prefixed with a counter that indicates the number of pointers (i.e., references) to that block. When a counter reaches zero its block is deallocated. Consequently, the counters of blocks it pointed to are decremented, possibly leading to a cascade of deallocations.

Even though it is often considered to be a distinct approach to memory management, RC is merely a variant of GC. Instead of scanning memory periodically to update the reachability graph and decide what can be safely deallocated, a program with RC updates the counters with each instruction and decides whether to deallocate values on the spot. Bacon, Cheng and Rajan propose a framework of which GC and RC are both instances [6]. In particular, they point out that optimisations of GCs (most notably, concurrent GCs) makes them closer to RCs. Vice versa, optimisations to RCs, makes them closer to GCs. In their framework, the main distinction between the two approaches is as follows: RCs explore and deallocate unreachable values (values are presumed useful until proven unreachable) whereas GCs scan reachable values and preserve them (values are presumed useless until proven reachable). The authors explain this distinction using the concepts of *matter* for the reachable-objects traced and saved by the GC and *anti-matter* for the unreachable-objects explored and deallocated by the RC. We reuse this vocabulary in later Sections.

Note that RCs still have the same issue as other GCs: they restrict the representation of values. Specifically, they require blocks to be prefixed with a counter and to include runtime type information for the cases when deallocations cascade.

1.2.3 Linear and region regimes

Linear type systems and *region-based memory managers* are other families of approaches to memory management. In these approaches, the source programming language is restricted by a type-like analysis – i.e., one that restricts the set of valid programs. Both the linear and region regimes are restrictions that make it safe for the compiler to replace unreachability with another characterisation of uselessness.

Linear type systems In a linearly typed program, each value must be used exactly once. This constraint is enforced by a type-like analysis inspired by linear logic [28]. Under this constraint, a value becomes useless right after it has been used, and can therefore be safely deallocated. Note

```
let rec sum = function
  | [] -> 0
  | x::xs -> x + sum xs
;;
let rec len = function
  | [] -> 0
  | _::xs -> 1 + len xs
;;
let rec get () =
  (*read from stdin and parse*)
  let data = readIntegers () in


  if sum data / len data > 1 then
    data
  else

    get ()
;;
```

(a) Conventional OCaml code

```
let rec sum = function
  | [] -> 0
  | x::xs -> x + sum xs
;;
let rec len = function
  | [] -> 0
  | _::xs -> 1 + len xs
;;
let rec get () =
  (*read from stdin and parse*)
  let data = readIntegers () in
  let (d1,d2) = deepCopy data in
  let (d3,d4) = deepCopy d1 in
  if sum d2 / len d3 > 1 then
    d4
  else
    ignore d4;
    get ()
;;
```

(b) Linear OCaml code

Figure 1.2: Before and after: administrative overhead of linear regimes

that in this case uselessness is akin to (but not exactly the same as) non-liveness – we explore this relation in Chapter 10.

Variants of linear type systems relax the value-usage restrictions. *Quasi-linear* type systems [25] allow programmers to mix linearly and non-linearly typed values in a single program: the former are freed on use, the latter managed by a GC. In an *affine* type systems values can be used at most once, but are not required to be used. Both variants relax the constraints of the linear regime, trading off some of its benefits for increased expressivity.

We illustrate the necessary program changes that (non-relaxed) linear types induce in Figure 1.2. These changes constitute the administrative overhead programmers must go through to satisfy the type-like analysis. They include explicitly copying `data` because it is used multiple times. They also include explicitly ignoring `d4` because it would otherwise go unused in the `else` branch. This administrative overhead highlights a practical penalty as well as a conceptual one: programmers are once again involved in memory management. Taking this conceptual penalty to the extreme, one can interpret the `ignore` function like C's `free` and see linear types as compiler-checked, manual memory management.

Region-based memory management With a region-based memory manager [38], values are allocated in a *region*. Additionally, a type-like analysis enforces that values do not escape their region. I.e., under the region regimes, pointer from an outer into an inner region are prohibited. When a region falls out of scope, all its values are deallocated.

We illustrate the administrative overhead that a region regime induce in Figure 1.3. Because the `altzip` mixes together the elements from both of its argument, it imposes they are allocated in the same region. Consequently, the elements of `us` and `vs` (both passed to `altzip`) must be allocated in the same region. As a result, their life-time is tied: elements of `vs` cannot be deallocated before the elements of `us`. Unfortunately, the `us` is returned from `f`: its memory can be arbitrarily long-lived (depending on the caller). Thus, the memory of `vs` is kept for an arbitrary

```
let rec altzip = function
  | x::xs, y::ys ->
    (x,y)::altzip (ys,xs)
  | _ -> []
;;
let f () =
  let us = readData () in
  let vs = readData () in
  (*compute*)

  let ws = altzip (us, vs) in
  use (ws);
  us
;;
```

(a) Conventional OCaml code

```
let rec altzip = function
  | x::xs, y::ys ->
    (x,y)::altzip (ys,xs)
  | _ -> []
;;
let f () =
  let us = readData () in
  let vs = readData () in
  (*compute*)
  let (us1, us2) = deepCopy us in
  let ws = altzip (us1, vs) in
  use (ws);
  us2
;;
```

(b) Region-friendly OCaml code

Figure 1.3: Before and after: administrative overhead of region regimes

long time even though it is not useful. This is fixed in Figure 1.3b: `us` is copied; one replica entangles inconsequently with `vs`; the other replica is returned. Note that these changes are not necessary for the program to run; but they reduce memory footprint of the program.

Whilst both systems are safe and efficient, they limit expressiveness. Indeed, programmers must express their ideas within a (purposefully) limiting framework. In some cases, programmers are required to add annotations or even memory management instructions (such as copying) to their program. This administrative overhead highlights the same conceptual penalty as for linear types: programmers are once again involved in memory management.

Regions (in conjunction with other techniques) are used in Cyclone [20] to manage memory safely. A mix of linear types and regions is used in Rust [2].

1.2.4 Memory re-use

Another area of interest is *memory re-use* – or simply *re-use*. A re-use system endeavours to replace allocations by mutations of dead memory blocks. That is, when a re-use system is successful, the program avoids the inefficient sequence: deallocate a dead block, allocate a new block, initialise the freshly allocated block. Instead, the program simply mutates the dead block to represent the new value. In order to decide re-use it is necessary to detect dead blocks and to find subsequent allocations for blocks of the same size.

Manual re-use is possible in programming languages that feature mutation.

Automatic re-use is always used in conjunction with another system – such as a GC [29], or a linear-type system [19]. The re-use system decreases the rate of allocations which reduces the workload of the memory management system.

1.3 System programming

System programming consists of writing and maintaining software in the lower layers of the operating system: hypervisors, device drivers, network stacks, file systems, cryptographic libraries, etc.

Memory management in system programs is an important topic. The MINIX author, Andrew Tanenbaum, writes [37]:

> My initial decision back in 1984 to [...] avoid dynamic memory allocation (such as malloc) and a heap in the kernel [...] avoids problems that occur with dynamic storage management (such as memory leaks and buffer overruns).

These specific problems are common in many programs written in C. Despite this drawback, C is the most common language used for system programming. As a result, serious memory management bugs are continuously discovered in system software.

There are historical reasons for C's prevalence: Unix and all its successors were written in C. As a result, there is, to this day, a cultural preference for C amongst systems programmers.

However, there is another reason for C to persist as the leading language for system programming. Indeed, consider the specificity of system software: it interacts directly with hardware. As a result, system software handles values in native format – e.g., ethernet frames as they appear on the wire. For this reason, programmers need full control over the representation of values in memory. This was pointed out by the FoxNet project [8]: a full network stack in SML suffering from efficiency issues due to the repeated copies and conversions back and forth between the system's native representation and the GC-compatible layout.

GCs, RCs and region regimes prevent the programmers from controlling the layout of values in memory. As such, they are not compatible with the memory management requirements of system programming. Conservative GCs are leaky, which is problematic for long-lived programs such as servers and daemons. They too are ill-suited for system programming. Linear type systems are restrictive: the complete opposite of C. Their adoption faces the inertia of the cultural prevalence of C in system programming.

A sound option for system programming is Rust – which was developed for that specific purpose. Rust can be used with a GC, but, by default, manages the memory through a hybrid linear-region regime that does not restrict the value layout. Rust's hybrid linear-region regime is inspired by manual memory management guidelines widespread amongst C programmers. Thus, Rust's constraints are looser than linear or region regimes and they broadly follow existing customs of system programmers. We explore Rust in more details in Chapter 3, and again in Chapter 10.

1.4 Asap

The rest of this dissertation presents a novel approach to memory management: AS static AS possible (ASAP). Asap is fully automatic: programmers are oblivious to the memory management just like with a GC. Asap is agnostic of the memory representation which can be taken care of by the programmers when necessary and left to the compiler otherwise. To provide these properties, ASAP analyses the program to detect value usage (at which points of the program are which values accessed) and aliasing (in what way which values alias). Based on this information, ASAP inserts instructions within the program to deallocate memory when appropriate. For the cases when it is not possible to decide at compile time if the memory representing a value can be safely reclaimed, ASAP inserts specialised code that will determine safety during execution. Interestingly, because the type of values is known when the code is generated, it is able to scan and deallocate values without runtime types nor tags.

1.5 Plan

We first give some technical background in Chapter 2. Specifically, we introduce intermediate representations and three-value logic and give an overview of data-flow analysis techniques.

We then explore the design space of memory management strategies in Chapter 3. We establish a lexicon specialised for memory management. This new-found vocabulary allows us to describe existing memory management strategies in a novel way and reveal a few unexplored areas in their design space. One of these lacunae, the static-automatic gap, we explore further.

We introduce *paths* in Chapter 4, a compile-time abstraction of the heap. In Chapter 5, we present ASAP: a novel memory-management strategy that fits in the static-automatic gap. We then extend ASAP to support more programming-language constructs in Chapter 6. We present our prototype implementation of ASAP in Chapter 7.

We compare ASAP to existing strategies in Chapter 8. In this comparison, we also show ways in which ASAP can be combined with other strategies, providing several hybrid solutions to memory management. We take a specific look at linear and region regimes in Chapter 9. Finally, we explore insights gained during the creation of ASAP.

Chapter 2

Prerequisites

We present here formalisms, notations and elements of theory that are used in later chapters. Most are known prior art which we present here for self-containedness and for the reader to get familiar with specific notations and variants. The exception is the intermediate representation μL (in Section 2.5.3–2.5.9) which we designed.

2.1 Memory

Memory is a storage mechanism that programs can use to place and retrieve values. Memory is composed of *words*: fixed-size chunks of consecutive bits[1] the processor can store or load directly – i.e., with a single instruction.

Words are used to represent values – which need to be encoded in binary.

When a value is too big to be stored in a word, it can be stored in several words. Consecutive words of memory that represent a value form a *block*.

Words are also used to represent pointers. In this case, their bit pattern represents the address of another word or block of memory. Following or dereferencing a pointer is loading the value stored at the address it represents.

Some values have tags. A tag is a set of bits from the memory representation of the value. The pattern of the tag bits does not contribute to the representation of the value per se. Instead, the tag bits constitute metadata most often about the sort of value represented by the other, non-tag bits. E.g., in OCaml[2], integers are tagged: their least significant bit is always 1 – leading to 63-, or 31-bit arithmetic. This distinguishes them from pointers, all of which have 0 as their least significant bit.

Two portions of the memory are distinguished: the stack and the heap. The stack stores small values local to a function as well as arguments and return values for function calls. The heap stores large values, values of non-statically-known size, and values that are not local to a function[3]. To give a mathematical abstraction of the memory we consider a set of identifiers $variable$, a set of memory addresses $address$. A stack is a function $\sigma : variable \to address \cup word$. A heap is a function $\eta : address \to block$ where a block is a tuple of $address \cup word$. We use this formalism, in Chapter 4, to define an additional layer of abstraction: paths.

[1]The actual size depends on the architecture of the machine the program runs on.
[2]A similar technique is used in some other ML, Lisp, or Ruby implementations.
[3]Note that these are guidelines: the stack can be used to store large values and the heap can be used to store values local to a function. However, this atypical use is generally less efficient.

2.2 Program, execution tree, trace

A *program* is a structured set of constructs which specifies how to carry out a computation. A *program point* is a location within the structure that is the program. We denote program points with a circled name like so: $\textcircled{\pi}$.

Carrying out the computation specified by the program is *executing* the program. During an execution, several program states are visited. A *state* is the condition a computer is in during an execution. It is determined by a program point as well as a heap and a stack[4]. Note that multiple states can correspond to a single program point. E.g., a program point located within a loop is visited several times with potentially different heaps and stacks. We write $State(\textcircled{\pi})$ for the set of pairs (η, σ) such that the program may reach the program point $\textcircled{\pi}$ with the heap η and the stack σ.

A *trace* is a sequence of states that was visited during a specific execution. Note the past tense in the previous sentence: a trace can only be obtained after an execution terminates. In particular, it is not possible to obtain a trace for a program that does not terminate.

Note that there are multiple possible executions of a single program: inputs affect the heap and stack which in turn affect the branching behaviour of the execution. The *execution tree* of a program is a tree of program states wherein each path descending from the root towards the leaves is a trace for a possible execution of the program.

Note that both traces and execution trees can be handled as abstract mathematical concepts. Also note that the execution tree is not, in general, computable. However, there are several techniques in computer science that are designed to approximate the execution tree and the set of visited states – e.g., abstract interpretation.

The notions defined above are used in Chapter 3 to formally define those values which are not useful for the rest of an execution.

2.3 Mathematical notations

We here list the various mathematical concepts used later on in this dissertation.

2.3.1 Typographic conventions

We reserve $\mathit{this\ italics\ font}$ for mathematics and `this monospace one` for code.

Note that, for program fragments, the code font is used for keywords and symbols and the mathematics font is used for non-terminals and meta-variables (i.e., mathematical variables that stand for code identifiers). E.g., we write `let` x `= 3 +` e `in` t where x stands for a identifier (say `x`), e for an expression and t for a term.

2.3.2 Maps

We consider *maps* which are total functions from finite domains. Additionally, we impose that both the domain and range of maps are a subset of a discrete countable universe – typically variables, terms, booleans, regular expressions, or pairs or sets thereof.

[4]This description is sufficient for our level of abstraction. In other contexts, more elements are constitutive of the state: the value held by registers, the buffers of network cards, the signals travelling on wires, the content of various caches, etc.

Extensional and intensional definitions We write $[x_1 \mapsto y_1, \dots, x_n \mapsto y_n]$, where the x_i are pairwise distinct, for the map that associates each x_i to y_i. The map $[\,]$ is the function with an empty domain.

Borrowing notation from set theory, we write $[x \mapsto f(x) \mid x \in X]$, where X is finite, for the map with domain X that maps any $x \in X$ to $f(x)$. By usual abuse of notation, we omit the guard when it is clear by context and write $[x \mapsto f(x)]$. (Note that f is not necessarily a map: it can be a function with an infinite domain. However, f has to be defined on the finite set X.)

Domain and range We write $domain(f)$ for the domain of the function f. We write $range(f)$ for the range of the function f.

Updates We write $f \lhd [x \mapsto y]$, where f is a map, for the map of domain $domain(f) \cup \{x\}$ which maps x to y and $\forall z \in domain(f) \setminus \{x\}$ maps z to $f(z)$. That is, $f \lhd [x \mapsto y]$ is f with x remapped to y.

We write $f \lhd g$ for $f \lhd [x_1 \mapsto g(x_1)] \lhd \cdots \lhd [x_n \mapsto g(x_n)]$ where $domain(g) = \{x_1, \dots, x_n\}$. Note that $domain(f \lhd g) = domain(f) \cup domain(g)$.

Composition and substitution We write $f \circ g$ for the composition of f and g. That is, $(f \circ g)(x) = f(g(x))$.

We use this notation mainly for substitution of variables: if f represents knowledge about some term and g represents a variable substitution scheme then $f \circ g$ represents the knowledge about the term after the substitution. With that use in mind, we write $\theta^x_y(f)$ for the map $f \circ [x \mapsto y]$. Additionally, we write $\theta^{\vec{x}}_{\vec{y}}(f)$ where $\vec{x} = (x_1, \dots, x_n)$ and $\vec{y} = (y_1, \dots, y_n)$, for the map $f \circ [x_i \mapsto y_i]$.

Note that, as is customary with substitution of variable, we implicitly lift the substitution to apply to structured elements. Specifically, we use $\theta^x_y(f)$ when f is a map of pairs of $(variable \times path)$ (*paths* are defined in Chapter 4). In this case, $\theta^x_y(f)$ is shorthand for $f \circ [(x, p) \mapsto (y, p) \mid p]$.

Domain restrictions We write $f \mid X$ where X is a set, for the function of domain $domain(f) \cap X$ which $\forall x \in domain(f) \cap X$ maps x to $f(x)$. That is, $f \mid X$ is f restricted to the domain X. Or, more formally, $f \mid X = [x \mapsto f(x) \mid x \in domain(f) \cap X]$.

We write $f \setminus X$ for the function of domain $domain(f) \setminus X$ which $\forall x \in domain(f) \setminus X$ maps x to $f(x)$. That is $f \setminus X$ is f restricted to the complement of X. In other words, $f \setminus X = [x \mapsto f(x) \mid x \in domain(f) \setminus X]$.

Practical considerations Note that maps are easily representable in computer programs – because they are finite relations between elements of two discrete countable universes. More interestingly, all the operations detailed above are computable. Specifically, given maps f and g and a finite set X, all represented in a computer program, it is possible to algorithmically compute the representation of $f \lhd g$, $f \circ g$, $f \mid X$ and $f \setminus X$. Additionally, provided X is finite, if f is computable then it is possible to compute the representation of $[x \mapsto f(x) \mid x \in X]$.

This is important as it lets us use maps to define our analyses and be assured there are possible implementations for them.

$$\begin{tikzcd} & \top & \\ 0 \arrow[ur] & & 1 \arrow[ul] \end{tikzcd}$$

(a) Uncertainty semi-lattice

$$\begin{tikzcd} 1 \\ \top \arrow[u] \\ 0 \arrow[u] \end{tikzcd}$$

(b) Truthness lattice

Figure 2.1: 3VL lattice structures

t	$\neg t$
0	1
$\top$	$\top$
1	0

(a) Not

$\vee$	0	$\top$	1
0	0	$\top$	1
$\top$	$\top$	$\top$	1
1	1	1	1

(b) Or

$\wedge$	0	$\top$	1
0	0	0	0
$\top$	0	$\top$	$\top$
1	0	$\top$	1

(c) And

$\sqcup$	0	$\top$	1
0	0	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$
1	$\top$	$\top$	1

(d) Merge

Figure 2.2: 3VL operators

2.4 Three-Value logic

Three-Value logic (3VL) is an algebra for logic with three values of truthness: true, unknown and false.

2.4.1 Orderings and lattices

More formally, 3VL is an algebra over the set $\{0, \top, 1\}$ (pronounced respectively "false", "maybe" and "true"). There are two distinct natural orders for 3VL: $0 < \top < 1$ and $0, 1 \sqsubset \top$. The former is the truthness order: 0 is less true than $\top$ which is less true than 1. The latter is the uncertainty order: 0 and 1 are less uncertain than $\top$. These orders form lattice structures, which are presented in Figure 2.1.

2.4.2 Operators

We define extended versions of the boolean operators in Figure 2.2. Note that, $\neg$ ("not"), $\vee$ ("or"), and $\wedge$ ("and") are mere extensions of the boolean counterparts where $\top$ is treated as per its place in the truthness-order. Specifically, $\vee$ is the upper-bound in the truthness lattice and $\wedge$ the lower-bound. The third operator, $\sqcup$ ("merge"), is an upper bound in the uncertainty semi-lattice.

The merge operator is used in static analyses to handle join points.

2.4.3 3VL sets

Traditional sets can be described as functions from their universe to the set of booleans $\{0, 1\}$. More precisely, there is a well-known, trivial correspondence between $\mathcal{P}(X)$ (the set of subsets of X) and 2^X (the space of functions from X into $\{0, 1\}$, these are known as the *characteristic functions* of their corresponding subset). Operations on these sets are lifted from operations on $\{0, 1\}$ by point-wise application of the characteristic functions. E.g., $\cup$ is a point-wise application of $\vee$: $(X \cup Y)(z) = X(z) \vee Y(z)$.

Similarly, 3VL sets over X can be described as functions from X to $\{0, \top, 1\}$; we write this space as 3^X. When X is finite and discrete – which is always the case thereafter, – the function representation is a map. When dealing with 3VL sets, we use both the set view and the underlying map view.

Operator lifting

Just like with traditional sets, we can lift 3VL logic operators into 3VL-set operators. More formally, consider a binary operator $\star : 3^{(3\times 3)}$ and two 3VL sets $f_1, f_2 \in 3^Z$, we write $f_1 \star f_2$ for the map $[z \mapsto f_1(z) \star f_n(z) \mid z \in Z]$. We define lifted operator on 3VL sets below. Note how, despite usage for sets, we avoid using $\cup$ and $\cap$ for the lifting of $\vee$ and $\wedge$.

$$\begin{array}{rrcl} & \multicolumn{3}{l}{\forall Z, \forall X, Y \in 3^Z} \\ \text{Union:} & X \vee Y & = & [z \mapsto X(z) \vee Y(z) \mid z \in Z] \\ \text{Intersection:} & X \wedge Y & = & [z \mapsto X(z) \wedge Y(z) \mid z \in Z] \\ \text{Merge:} & X \sqcup Y & = & [z \mapsto X(z) \sqcup Y(z) \mid z \in Z] \\ \text{Negation:} & \neg X & = & [z \mapsto \neg X(z) \mid z \in Z] \end{array}$$

Additionally, we define the two following partial orders on 3VL sets.

$$\begin{array}{rrcl} & \multicolumn{3}{l}{\forall Z, \forall X, Y \in 3^Z} \\ \text{Subset}_{<}\text{:} & X \leq Y & \iff & \forall z \in Z, X(z) \leq Y(z) \\ \text{Subset}_{\sqsubset}\text{:} & X \sqsubseteq Y & \iff & \forall z \in Z, X(z) \sqsubseteq Y(z) \end{array}$$

Extended lifted operators for fixpoint initialisation

Operator lifting, as described above, requires the domain of maps to coincide. However, during data-flow analyses, we often need to apply these lifted operators to maps with distinct domains. Specifically, when initialising a fixpoint computation, we use the empty map [ ] as the initial value for some maps.

We extend the lifted operators as follows: the empty map [ ] is neutral for all the lifted operators defined above (union, intersection, merge). Additionally, the empty map is neutral for the update ($\lhd$) operator. That is, $[\,] \lhd m = m \lhd [\,] = m$, $[\,] \vee m = m \vee [\,] = m$, etc.

2.4.4 3VL relations

In traditional boolean logic, a relation over values of Z can be defined as a set of pairs of elements of Z. That is, a (traditional) relation over Z is just an element of $\mathcal{P}(Z \times Z)$ or $2^{Z \times Z}$.

Similarly, we define 3VL relations as members of $3^{(Z\times Z)}$. For a relation $R \in 3^{(Z\times Z)}$, the following closure operators (analogous to their traditional counterparts) are defined.

Transitive $Tr(R) = \bigvee_{i\in\mathbb{N}} R_i$
where $R_0 = R$ and $R_{n+1}(a,b) = \bigvee_{c\in Z}(R_n(a,c) \wedge R_n(c,b))$

Symmetric $Sym(R)(a,b) = R(a,b) \vee R(b,a)$

Reflexive $\mathit{Refl}(R)(a,b) = R \lhd [(a,a) \mapsto 1 \mid a \in domain(R)]$

$$
\begin{array}{rcll}
\mathit{value} & ::= & \mathit{literal} & \\
& | & \mathit{variable} & \\
\mathit{expr} & ::= & \mathit{value} & \\
& | & \mathit{value}\ \mathit{op}\ \mathit{value} & \text{(operators)} \\
& | & \mathit{funname}\ \texttt{(}\mathit{variable}, \ldots, \mathit{variable}\texttt{)} & \text{(function call)} \\
\mathit{stm} & ::= & \mathit{variable}\ \texttt{<-}\ \mathit{value} & \text{(assignment)} \\
\mathit{term} & ::= & \texttt{return}\ \mathit{value} & \\
& | & \texttt{let}\ \mathit{variable}\ \texttt{=}\ \mathit{expr}\ \texttt{in}\ \mathit{term} & \text{(binding)} \\
& | & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{term}\ \texttt{else}\ \mathit{term} & \text{(conditional)} \\
& | & \mathit{stm}\texttt{;}\ \mathit{term} & \text{(sequence)} \\
& | & \texttt{while}\ \mathit{expr}\ \texttt{do}\ \mathit{term} & \text{(loop)}
\end{array}
$$

Figure 2.3: A simple ANF grammar

2.4.5 Practical considerations

For any finite set Z, the 3VL sets and relations over Z can be represented in a computer's memory. More specifically, the operators and orders, lifted from 3VL, are computable and so are the closure operators. This is straightforward except for the transitive closure because it is defined as a union over $\mathbb{N}$. However, note that $Z^{(3\times3)}$ with the $\leq$ order forms a finite lattice[5] in which R_i is monotonic. Thus, the union over $\mathbb{N}$ is in fact a well-defined fixpoint operation. (We discuss fixpoints in more details in Section 2.6.3.)

2.5 Intermediate representations, µL

We present here three IRS: administrative-normal form (ANF), continuation-passing style (CPS) and µL (pronounced "micro language" or "micro L"). ANF and CPS are well known IRS presented here for didactic purposes. On the other hand, µL is the IR we use in Chapter 5. It is an hybrid between ANF and CPS.

2.5.1 Administrative normal form

ANF is an IR whose grammar limits the nesting of terms. In particular, function arguments and branching conditions must be either simple values (i.e., literals) or variables. Additionally, the right-hand side of a binding cannot contain another binding[6]. These constraints are enforced by the grammar which has separate categories for *terms* (compound expressions involving control-flow), *expressions* (simple expressions not involving control-flow), and *values* (literals and variables).

As a result, ANF is a strongly hierarchical system: terms, expressions, statements, and values are distinct classes of construct. A simple ANF grammar is presented in Figure 2.3. Notice the hierarchy of strict inclusion: terms (term) can appear neither in expressions (expr) nor in statements (stm), none of which can appear in values (value). Interestingly, under eager semantics, the evaluation order of terms in ANF is explicit. More about ANF can be found in the work of Chakravarty, Keller and Zadarnowski [10].

[5]Remember that Z is finite.

[6]This makes ANF a scoped version of three-address code.

goto-based IRS	CPS
label	function definition
block	function body
return address	function (continuation) variable
goto / jump	function call
conditional jump	conditional function call
return	function (continuation) call

Figure 2.4: Correspondence between goto and CPS

$$
\begin{array}{rcll}
\mathit{expr} & ::= & \mathit{literal} & \\
 & \mid & \mathit{variable} & \\
 & \mid & \mathit{expr}\ \mathit{op}\ \mathit{expr} & \\
 & \mid & \lambda\ \mathit{variable}\ \ldots\ \mathit{variable}\ .\ \mathit{term} & \text{(lambda)} \\
\mathit{term} & ::= & \texttt{let}\ \mathit{variable}\ \texttt{=}\ \mathit{expr}\ \texttt{in}\ \mathit{term} & \text{(binding)} \\
 & \mid & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{call}\ \texttt{else}\ \mathit{call} & \text{(conditional call)} \\
 & \mid & \mathit{call} & \text{(instead of return and jump)} \\
\mathit{call} & ::= & \mathit{expr}\texttt{(}\mathit{expr}\texttt{,}\ \ldots\texttt{,}\ \mathit{expr}\texttt{,}\ \mathit{expr}\texttt{)} & \text{(call with continuation)}
\end{array}
$$

Figure 2.5: A simple CPS grammar

2.5.2 Continuation-passing style

CPS is a functional IR in which there is only one mechanism for control flow: function call. Not even returns are allowed: instead of `return`, there is a function call at the end of the body of every function. This call is to a *continuation*: a function passed (hence the name CPS) by the caller to the callee in addition to the other arguments. Figure 2.4 shows the correspondence between constructs of CPS and goto-based IRS[7].

CPS programs rely on a minimal set of constructs, which makes them easier to analyse: all blocks are functions, all jumps are function calls. A simple CPS grammar is presented in Figure 2.5. Note that, in the *call* construct, one of the arguments is a continuation[8] (either a lambda or a variable bound to a lambda).

Note that in CPS it is not possible to return a value from callee to caller. Specifically, it is not possible to use the `let` construct to bind the result of a function call. This is enforced by the grammar that only allows binding to expressions, not terms. Instead of writing `let`x`=`f`(`y`,`z`)` `in`t, in CPS the result of a function is used within t as follows: f`(`y`,`z`,`$\lambda\, x\ .\ t$`)`.

2.5.3 μL

μL is both a core-calculus we use to expose the ideas and principles underlying ASAP (Chapter 5) and an IR we use for our prototype (Chapter 7). Thus, μL was designed for both clarity of exposition and simplicity of implementation. It inherits a clear-cut hierarchy of constructs from ANF. It inherits the use of functions for all intra-procedure jumps (loops, confluence points) from CPS. Note however that it uses a standard `return` approach to function returns.

[7]A complete translation from CPS to single-static assignment and back is presented by Kelsey in [22].

[8]In some CPS IRS, functions take additional continuations to encode additional control flow mechanisms such as raising exceptions.

```
   value,  ::= literal
   pattern  |  variable
            |  discriminantname variable                   (sum variant)
            |  { fieldname=variable; ... ;
                   fieldname=variable; }                          (record)

      expr ::= value
            |  op(variable, ..., variable)    (arithmetic operator)
            |  funname(variable, ..., variable)    (function call)

      term ::= ⊙let variable : α = expr in term                  (binder)
            |  ⊙match variable with                      (match, branch)
                 [ ⊙pattern → term
                 | ...
                 | ⊙pattern → term
                 ]⊙
            |  ⊙return variable⊙                                  (return)

       def ::= fun funname(variable : α, ..., variable : α) : α =
                 term

   program ::= def ... def
```

Figure 2.6: Grammar of μL

2.5.4 Grammar

The grammar of μL is presented in Figure 2.6. The circle markers (⊙) indicate program points – more about program points is said in Section 2.6.

We use the meta-variables x, y and z to range over *variable*, D to range over *discriminantname*, F to range over *fieldname*, f and g to range over *funname*, and α to range over type names (see Section 2.5.7).

The *literal* category allows the description of non-pointer, word-sized values. These values are allocated on the stack or unboxed in records – details about memory representation of values in μL can be found in Section 2.5.8. Note that, at this step of the compilation process, the distinction between different flavours of non-pointer, word-sized values (integers, booleans, etc.) has been erased.

The discriminant construct allows injections into (when used as a value) and projections from (when used as a pattern) sums. Similarly, the record construct is used for both construction (when used as a value) and destruction (when used as a pattern) of records.

The *term* category provides most of the control flow constructs: sequences (`let`), branching (`match`) and return (`return`). Expressions – inside `let`-bindings – provide function calls.

2.5.5 Assumptions

Because μL is an IR, we can make a number of assumptions about programs without loss of generality. Indeed, μL programs are compiled down from an unspecified source language. This source language can have many more features than μL. As detailed in Chapter 7, our prototype includes a small front-end alleviating some of the limitations of μL. Below is a list of assumptions about programs in μL. Each is presented with a method to compile a richer (less assumptive) language into μL.

- We assume five distinct syntactic categories of identifiers: *variable* for values, *funname* for functions, *discriminantname* for sum discriminants, *fieldname* for record fields, and *typename* for types. The distinction need not exist in the source program; it is easy for the parser to classify an identifier based on the context in which it appears in the program.

- We assume the program is monomorphically type annotated. The grammar (see Section 2.5.4) enforces that function parameters and `let`-bound variables are annotated with a type identifier (α). Once again, it is not necessary for the source program to be type annotated, the compiler can perform type inference early on in the pipeline. However, this assumption does require the source program's memory safety to be expressible in μL's type system.

- We do not provide any construct for polymorphism. Whole program monomorphisation, whilst possible, is not a widely deployed technique. We ignore polymorphism for now, and introduce it in Chapter 6.2.

- We only consider programs consisting of a single compilation unit. In other words, we only consider analyses and transformation passes that act on the whole program – even though the source program might be split in modules, namespaces and even files. Allowing split programs degrades the precision of our inter-procedural analyses as detailed in Chapter 5.

- We do not provide grammar constructs for nested functions. This limitation can be alleviated through λ-lifting [21] (the technique we choose in our prototype detailed in Chapter 7).

- We assume values are immutable. Even though it is possible to compile code that makes use of a destructive assignment operator into immutable form (see Haskell's state monad [27]), we take a more direct approach: we simply ignore all form of mutation for now, and introduce an explicit mutation operator in Chapter 6.1.

- Function definitions are recursive but not mutually so. This is a didactic limitation: with this assumption it is simpler to explain our analyses' fixpoint operations.

  Note that it is possible to compile a set of mutually recursive functions into simply recursive functions using a trampoline. A trampoline is used as follows. Consider a set of mutually recursive functions $f_1, \ldots, f_n$. An additional function f_0 is synthesised such that it dispatches calls to the appropriate f_i based on its first argument. Instead of calling each others, the function simply returns a value indicating which of the f_i to call next.

  Trampolines increase the number of function calls (each call is indirected through f_0), rely on branching (f_0 dispatches to the f_i during execution), and, depending on the backend, can require allocations to box arguments.

- We assume first-order programs: higher-level functions are not allowed. The defunctionalisation [14] algorithms transform higher-level programs into equivalent first-order ones.

2.5.6 Syntactic sugar

For brevity we use syntactic sugar in all of our examples and omit type annotations. Figure 2.7 presents the additional constructs for terms with their desugared equivalent in the vanilla grammar. Note that $\textcircled{1}$, $\textcircled{2}$, $\textcircled{3}$ and $\textcircled{4}$ are program points. Also note that our prototype's front-end, applies these specific transformations, amongst others.

$$[\![{}^{\textcircled{1}} e ;\ {}^{\textcircled{2}} t]\!]_{\text{sugar}} = {}^{\textcircled{1}}\texttt{let}\ x : \mathit{unit} = e\ \texttt{in}\ [\![{}^{\textcircled{2}}t]\!]_{\text{sugar}} \quad (x \text{ is fresh})$$

$$\left[\!\!\left[\begin{array}{l} {}^{\textcircled{1}}\texttt{if}\ x\ \texttt{then} \\ \qquad {}^{\textcircled{2}}t_t \\ \texttt{else} \\ \qquad {}^{\textcircled{3}}t_f \\ \textcircled{4} \end{array}\right]\!\!\right]_{\text{sugar}} = \begin{array}{l} {}^{\textcircled{1}}\texttt{match}\ x\ \texttt{with} \\ \quad \texttt{[1} \rightarrow [\![{}^{\textcircled{2}}t_t]\!]_{\text{sugar}} \\ \quad \texttt{| 0} \rightarrow [\![{}^{\textcircled{3}}t_f]\!]_{\text{sugar}} \\ \quad \texttt{]}^{\textcircled{4}} \end{array}$$

$$[\![{}^{\textcircled{1}} t_1 ;\ {}^{\textcircled{2}} t_2]\!]_{\text{sugar}} = \begin{array}{l} {}^{\textcircled{1}} f\texttt{(}x_1\texttt{,} \ldots \texttt{,}x_n\texttt{)}\ \texttt{;}\ [\![{}^{\textcircled{2}}t_2]\!]_{\text{sugar}} \\ \left(\begin{array}{l} f \text{ is fresh} \\ x_i \text{ are the free variables of } t_1 \\ f \text{ is bound by } \texttt{fun}\ f\texttt{(}x_1\texttt{,} \ldots \texttt{,}x_n\texttt{)} = t_1 \end{array}\right) \end{array}$$

$$[\![{}^{\textcircled{1}} \texttt{\{\}}]\!]_{\text{sugar}} \text{ (as a term)} = {}^{\textcircled{1}}\texttt{let}\ x : \mathit{unit} = \texttt{\{\}}\ \texttt{in}\ {}^{\odot}\texttt{return}\ x^{\odot} \quad (x \text{ is fresh})$$

$$[\![{}^{\textcircled{1}} t]\!]_{\text{sugar}} = t \qquad \text{(otherwise)}$$

Figure 2.7: Syntactic sugar in µL

2.5.7 Types

We use the meta-variable α to range over the type names. The type names are bound to types τ in the global map Δ. This and the hierarchical grammar of types, detailed below, imposes that every type be named. Having an identifier for each type helps when handling paths in Chapter 4.

Types, defined in Figure 2.8, are either *word* (i.e., bit-sequences), *product* or *sum*. Products are composed of zero or more fields, each contains a value. Sums are unions disambiguated by *discriminantnames*. (Section 2.5.8 gives details about memory representation of values.)

The *unit* type is the empty product type $\{\}$ and the unit value is the empty record `{}`. We write Γ for the type environment; it maps value names (*variable*) to type names (*typename*).

$$\begin{array}{rll} \mathit{atom} ::= & \mathit{word} \ \mid \ \mathit{typename} & \\ \mathit{product} ::= & \{\, \mathit{fieldname} : \mathit{atom};\ \ldots;\ \mathit{fieldname} : \mathit{atom}\} & \\ \mathit{sum} ::= & \mathit{discriminantname}\ \mathit{atom}\ +\ \ldots\ +\ \mathit{discriminantname}\ \mathit{atom} & \\ \tau ::= & \mathit{word} \ \mid \ \mathit{product} \ \mid \ \mathit{sum} & \text{(type)} \\ \Delta : & \mathit{typename} \rightarrow \tau & \text{(environment of type names)} \\ \Gamma : & \mathit{variable} \rightarrow \mathit{typename} & \text{(type environment of variables)} \end{array}$$

Figure 2.8: The types of µL

2.5.8 Memory representation

Note that the types of µL do not necessarily correspond to the types of the source language. In particular it is possible for the source language to distinguish between integers, characters, and booleans even though there is only one type for word-sized values. Similarly, the source language can provide types for tuples which can be compiled into products of the form given in Figure 2.8.

Moreover, we assume that, in a product, fields of type $word$ are represented unboxed, inline in the memory block of the product. And conversely, fields carrying a value of a named type α are represented as a pointer to a separate memory block. Similarly for sums: $word$ items are stored inline and named types are pointers. This assumption can be made without loss of generality. Indeed, it is possible the source language offers inline/unboxed products which are eliminated by the time µL is used. Specifically, consider a higher-level type declaration such as `type`t`={`F_0`:{`F_1`:int;`F_2`:int}}`. To represent values of this type with an unboxed F_0 component, it is rewritten `{`F_{01}`: int;` F_{02}`: int}` and accesses of the form $x.F_0.F_1$ are replaced by $x.F_{01}$. (If pointers to unboxed components are allowed in the higher-level source, a more complex transformation is necessary.)

As a consequence to these restrictions, it is not possible, in a program, to hold a pointer to the middle of a block. Indeed, consider the program `let` x `=` $y.F$ `in` $^{\textcircled{\pi}}t$. At program point $\textcircled{\pi}$, the variable x, on the stack or in the registers, holds either a word (if the field F has type $word$) or a pointer to a memory block representing a value of type α (if the field F has type α).

Note that several optimisations are possible but not worth the added complexity: they are orthogonal to our considerations and lead to special cases. Specifically, sums with only one disjunct $D\ a$ can be treated directly like the atom a. Additionally, discriminants that carry the unit type $D\ unit$, where $\Delta(unit) = \{\}$, can be treated as $word$.

2.5.9 Example

Figure 2.9a gives an example where the types $pair$ (of pairs of words) and $list$ (of lists of pairs) are defined and used in a function ($makeExampleList$) that returns a value of type $list$. Note that the type of cons cells must be bound to a type identifier ($cons$) in order to define the type of lists.

Figure 2.9b shows the memory representation of the value returned by the function $makeExampleList$. Parts in grey are not represented in memory during execution: they are field identifiers, only their content is available during execution.

2.6 Data-Flow Analyses

A data-flow analysis is a method to gather information about the values handled by a program. Specifically, a data-flow analysis is a pass of the compiler that, given a program, generates a system of equations – sometimes referred to as a bag of constraints – relating information at one program point with information at another program point. The solution to this system of equations[9] is the information gathered by the analysis.

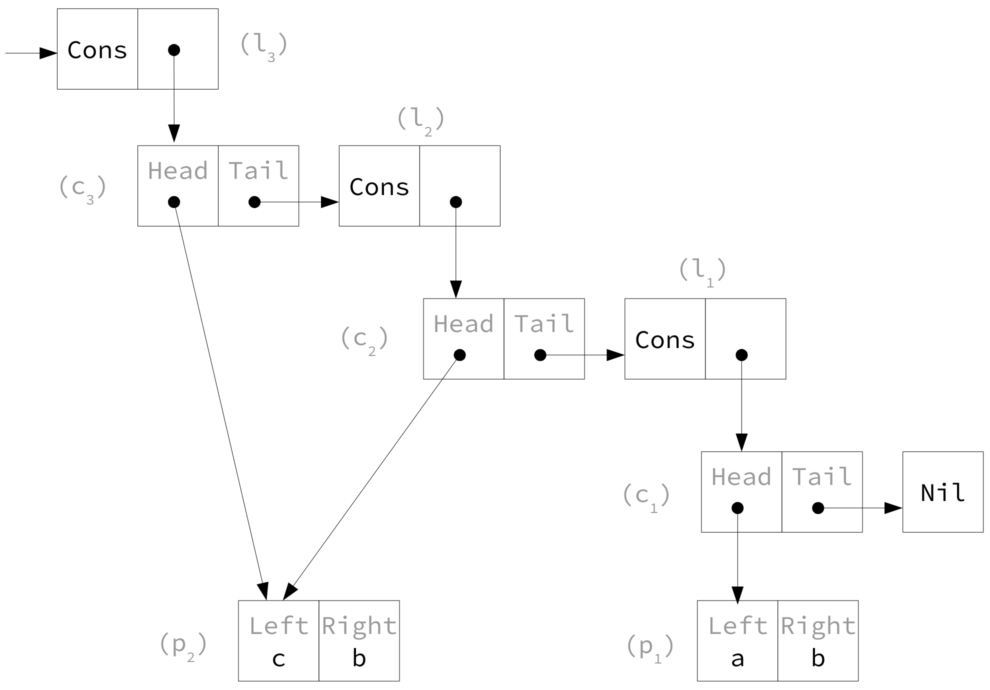
We present here a framework for data-flow analysis through an example (live-variable analysis) which we generalise on later. We use this framework in Chapter 5 to define the analyses of ASAP.

[9]Note that there is not always a single solution to the system of equations, nor is there always a best or preferred one. Thus, separate implementations of the same analysis might return different results.

```
$unit = \{\}$
$pair = \{Left : word;\ Right : word\}$
$cons = \{Head : pair;\ Tail : list\}$
$list = Cons\ cons + Nil\ unit$

$makeExampleList$($a$ : $word$, $b$ : $word$, $c$ : $word$) : $list$ =
    let $p_1$ : $pair$ = {$Left$: $a$; $Right$: $b$} in
    let $p_2$ : $pair$ = {$Left$: $c$; $Right$: $b$} in
    let $unit$ : $unit$ = {} in
    let $nil$ : $list$ = $Nil\ unit$ in
    let $c_1$ : $cons$ = {$Head$: $p_1$; $Tail$: $nil$} in
    let $l_1$ : $list$ = $Cons\ c_1$ in
    let $c_2$ : $cons$ = {$Head$: $p_2$; $Tail$: $l_1$} in
    let $l_2$ : $list$ = $Cons\ c_2$ in
    let $c_3$ : $cons$ = {$Head$: $p_2$; $Tail$: $l_2$} in
    let $l_3$ : $list$ = $Cons\ c_3$ in
    return($l_3$)
```

(a) Example of μL types and function



(b) Memory representation of the value returned by *makeExampleList*

Figure 2.9: Example of μL with memory representation

2.6.1 Example: simple live-variable analysis

To illustrate data-flow analyses we formally define live-variable analysis (LVA) on μL terms. LVA associates a set of variables to each program point. Variables in the set are said to be *live*, which indicates their value at the considered program point is useful for the rest of the program. (Note that because μL has no mutation operator, liveness is only influenced by definition and read points.)

Many presentations of LVA use sets of variables but we use maps (as per Section 2.3.2) of variables to the booleans $\{0, 1\}$. The domain of such maps is the set of variables in scope. If a variable maps to 1 it is live, to 0 it is not. In other words, our LVA associates, for any program point $\textcircled{\pi}$, a map $Live(\textcircled{\pi})$ such that $Live(\textcircled{\pi})(x)$ is the liveness of x at program point $\textcircled{\pi}$.

Additionally, LVA is commonly presented for imperative intermediate representations. In these presentations, the program is seen as a graph where vertices are program instructions and edges are possible transitions of the program counter. Sets of variables are attached to the entry and exit of each instruction vertex. In our presentation, because we use a scoped, functional IR, we associate information to each program point (defined in the grammar).

The data-flow equations are generated as per Figure 2.10a. These equations use the three $Transfer$ functions detailed in Figure 2.10b. Each $Transfer$ function operates on a distinct kind of term: expression bindings, pattern matchings and returns. We handle branching (in the `match` construct) with a lifted $\vee$ ("or") boolean operator. This lifted operator applies to maps ($Live(\textcircled{a_i})$) whose domains coincide because the set of variables in scope at each of these points is the same.

To solve the system of equations initial values are required. For each function $f(\dots) = t^{\textcircled{e}}$, the initial value is $Live(\textcircled{e}) = [y \mapsto 0 \mid y \text{ in scope at } \textcircled{e}]$.

2.6.2 Generalising: direction and approximation

We now take a step back and generalise from the liveness analysis example above.

Decorations The live-variable analysis generates a system of equations the solution of which is a map from variables to booleans for each program point. More precisely, it produces a function $L : \odot \to variable \to \{0, 1\}$. Other analyses would generate functions from program points to other mathematical constructs. We call such functions *decorations*. We avoid the word *annotations* because of its common usage in the context of user-provided information – e.g., type annotations, region annotations. However, just like annotations, decorations are pieces of information attached to different parts of a term. They decorate the term but do not change its execution behaviour. Formally, a decoration for a property is a function from program points to mathematical objects that express that property.

Direction In the example above, the data flow analysis is *backward*: using liveness information from program point at the end of the term, the liveness information at program points at the start of the term can be deduced. In other words, information flows backwards in the term.

Other analyses go in the opposite *direction*: they are *forward*. In forward analyses, information flows from the start to the end of a term. Generic equations (using analysis-specific $Transfer$ functions) for a forward analysis are presented in Figure 2.11.

Approximation: may- and must-analyses There is another interesting characteristic of LVA presented above: it uses the lifted $\vee$ operator to handle branching. As a result, a variable will be statically approximated to live even if it only happens to be actually live in some of the executions.

term	equations
$^{\textcircled{1}}\texttt{let}\ x = e\ \texttt{in}\ ^{\textcircled{2}}t$	$Live(\textcircled{1}) = Transfer_{\mathrm{e}}(x, e)(Live(\textcircled{2}))$
$^{\textcircled{a}}\texttt{match}\ x\ \texttt{with}$ $[\textcircled{a_1}p_1 \rightarrow \textcircled{b_1}t_1\textcircled{c_1}$ $\mid \ldots$ $\mid \textcircled{a_n}p_n \rightarrow \textcircled{b_n}t_n\textcircled{c_n}$ $]^{\textcircled{c}}$	$\forall i \leq n, Live(\textcircled{c_i}) = Live(\textcircled{c})$ $\forall i \leq n, Live(\textcircled{a_i}) = Transfer_{\mathrm{p}}(x, p_i)(Live(\textcircled{b_i}))$ $Live(\textcircled{a}) = \bigvee_{i \leq n} Live(\textcircled{a_i})$
$^{\textcircled{1}}\texttt{return}\ x^{\textcircled{2}}$	$Live(\textcircled{1}) = Transfer_{\mathrm{r}}(x)(Live(\textcircled{2}))$

(a) Equations generated for each term (equations for sub-terms t, t_1, ..., t_n are omitted)

$$
\begin{array}{rcl}
\multicolumn{3}{c}{Transfer_{\mathrm{e}} : variable \times expr \rightarrow 2^{variable} \rightarrow 2^{variable}} \\
Transfer_{\mathrm{e}}(x, l)(m) & = & m \setminus \{x\} \\
Transfer_{\mathrm{e}}(x, y)(m) & = & m \lhd [y \mapsto 1] \setminus \{x\} \\
Transfer_{\mathrm{e}}(x, D\ y)(m) & = & m \lhd [y \mapsto 1] \setminus \{x\} \\
Transfer_{\mathrm{e}}(x, \{F_1{=}y_1;\ \ldots F_n{=}y_n\})(m) & = & m \lhd [y_i \mapsto 1 \mid i \leq n] \setminus \{x\} \\
Transfer_{\mathrm{e}}(x, op\,(y_1, \ldots, y_n))(m) & = & m \lhd [y_i \mapsto 1 \mid i \leq n] \setminus \{x\} \\
Transfer_{\mathrm{e}}(x, f\,(y_1, \ldots, y_n))(m) & = & m \lhd [y_i \mapsto 1 \mid i \leq n] \setminus \{x\}
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{c}{Transfer_{\mathrm{p}} : variable \times pattern \rightarrow 2^{variable} \rightarrow 2^{variable}} \\
Transfer_{\mathrm{p}}(x, l)(m) & = & m \lhd [x \mapsto 1] \\
Transfer_{\mathrm{p}}(x, y)(m) & = & m \lhd [x \mapsto 1] \setminus \{y\} \\
Transfer_{\mathrm{p}}(x, D\ y)(m) & = & m \lhd [x \mapsto 1] \setminus \{y\} \\
Transfer_{\mathrm{p}}(x, \{F_1{=}y_1;\ \ldots F_n{=}y_n\})(m) & = & m \lhd [x \mapsto 1] \setminus \{y_i \mid i \leq n\}
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{l}{Transfer_{\mathrm{r}} : variable \rightarrow 2^{variable} \rightarrow 2^{variable}} \\
Transfer_{\mathrm{r}}(x)(m) & = & m \lhd [x \mapsto 1]
\end{array}
$$

(b) *Transfer* functions for the live-variable analysis

Figure 2.10: Live-variable analysis for μL terms

term	equations
$^{\textcircled{1}}\texttt{let}\ x = e\ \texttt{in}\ ^{\textcircled{2}}t$	$Live(\textcircled{2}) = Transfer_{\mathrm{e}}(x, e)(Live(\textcircled{1}))$
$^{\textcircled{a}}\texttt{match}\ x\ \texttt{with}$ $[\textcircled{a_1}p_1 \rightarrow \textcircled{b_1}t_{1}\textcircled{c_1}$ $\mid \ldots$ $\mid \textcircled{a_n}p_n \rightarrow \textcircled{b_n}t_{n}\textcircled{c_n}$ $]^{\textcircled{c}}$	$\forall i \leq n, Live(\textcircled{a_i}) = Live(\textcircled{a})$ $\forall i \leq n, Live(\textcircled{b_i}) = Transfer_{\mathrm{p}}(x, p_i)(Live(\textcircled{a_i}))$ $Live(\textcircled{c}) = \bigvee_{i \leq n} Live(\textcircled{c_i})$
$^{\textcircled{1}}\texttt{return}\ x^{\textcircled{2}}$	$Live(\textcircled{2}) = Transfer_{\mathrm{r}}(x)(Live(\textcircled{1}))$

Figure 2.11: Equations generated in a forward data-flow analysis (equations for sub-terms t, t_1, ..., t_n are omitted)

must	may	3VL
1	1	1
0	1	$\top$
0	0	0
1	0	inconsistent

Figure 2.12: Merging may and must information into 3VL

(Note that approximation is often necessary: many properties cannot be precisely decided statically.) In other words, liveness is over-approximated. As such, liveness is called a *may*-analysis: it can generate false positives but no false negative.

On the other hand, some analyses are *must*-analyses: they can generate false negatives but no false positive. Must-analyses under-approximate their property: they handle branching with the lifted $\wedge$ operator. As a result, a property is statically approximated to be true only if it happens to be actually true for all possible executions. Or, in other words, it is considered to be false even if it is only false in some of the possible executions.

Additionally, the set of possible executions is not, in general, decidable statically – because it depends on branching. Specifically, programs can contain infeasible paths: branches that are never reached but cannot be statically characterised as such. Thus, all static analyses approximate the set of possible executions. This adds a source of false-positives in may-analyses and false-negatives in must-analyses.

For different uses of the analysed information, either may- or must-analysis is appropriate. LVA is commonly used for register allocation in the back-end of compilers. Specifically, LVA is used to decide which virtual registers to coalesce. Coalescing two live virtual registers into the same physical register can change the execution behaviour of the program. On the other hand, failing to coalesce two virtual registers that are not simultaneously live merely slows down execution. Thus, liveness must be over-approximated for register coalescing to be safe.

Approximation: 3VL-analyses There is an alternative way to approximate static properties: three-value logic (3VL). In a 3VL-analysis, there are neither false-positives nor false-negatives, instead, when the analysis cannot decide whether something is true or false, it uses the $\top$ ("maybe") value. Running a 3VL-analysis is akin to running both may- and must-analyses simultaneously and merging the results as per the table of Figure 2.12. Note that the *inconsistent* combination shown in the table can actually appear during an analysis depending on its initialisation strategy. The inconsistent combination can carry all the way to the result of the analysis but only on parts of the code that are statically characterisable as dead.

A 3VL-analysis uses $\sqcup$ to handle branch points. Interestingly, $\sqcup$ behaves like $\wedge$ when 0 and $\top$ are amalgamated – as is the case in a must-analysis. And conversely, $\sqcup$ behaves like $\vee$ when 1 and $\top$ are amalgamated – as is the case in a may-analysis.

2.6.3 Example: advanced LVA

Notice, in the simple LVA example above, how functions calls are handled: the arguments are considered live in the caller (i.e., are mapped to 1 in the $\mathit{Transfer}_{\mathrm{e}}$ function) even if the callee ignores them. Similarly, the returned variable is considered live in the callee, even if the callers ignore it. We increase the precision of the liveness analysis for these specific constructs by considering the use of values across function calls.

$$\begin{aligned} \mathit{Transfer}_{\mathrm{e}}(x, f\,(y_1, \ldots, y_n))(m) &= m \lhd (\theta^{\vec{x}}_{\vec{y}}(f^{\downarrow})) \setminus \{x\} \\ \mathit{Transfer}_{\mathrm{r}}(x)(m) &= m \lhd (\theta^{x}_{ret}(f^{\uparrow})) \end{aligned}$$

Figure 2.13: Changes to the *Transfer* functions for advanced live-variable analysis

For each function $f\,(x_1, \ldots, x_n) = ^{\textcircled{b}} t_{body}$, we analyse the liveness of variables in t_{body}. We then define the map $f^{\downarrow}$ as the liveness decoration at program point $\textcircled{b}$ restricted to formal parameters:

$$f^{\downarrow} = \mathit{Live}(\textcircled{b}) \mid \{x_1, \ldots, x_n\}$$

The map $f^{\downarrow}$ is the *summary* of the function f.

We use $f^{\downarrow}$ in our $\mathit{Transfer}_{\mathrm{e}}$ function as per Figure 2.13. Specifically, for a function call $f\,(y_1, \ldots, y_n)$, the arguments (y_i) take the liveness of the corresponding formal parameters (x_i) at the entry point of f. (This is achieved by the renaming operator θ defined in Section 2.3.2.) Arguments that correspond to dead parameters of f do not become live as a result of the call.

Note that when the function is recursive, it is necessary to fixpoint $f^{\downarrow}$. To this end, we first assign $f^{\downarrow}$ an initial value $f^{\downarrow}_0$ (see below) used to compute a new summary $f^{\downarrow}_1$ used to compute a newer summary $f^{\downarrow}_2$ and so on until a fixpoint $f^{\downarrow}_n$ is reached (i.e., until $f^{\downarrow}_n = f^{\downarrow}_{n+1}$). This process is guaranteed to terminate because $2^{\{x_1,\ldots,x_n\}}$ forms a finite lattice in which $f^{\downarrow}_i$ is monotonic. The natural initial value is $f^{\downarrow}_0 = [x_i \mapsto 0 \mid i \leq n]$ (i.e., assuming at first that no parameter is used). Note that, starting with a different initial summary is correct, but leads to less precise solutions.

We further improve the liveness analysis by considering the context of calls. Specifically, for each term of the form `let` y = f`(...)` `in` $^{\textcircled{c}}t$, we collect the pair $(\textcircled{c}, y)$. We write C for the set of collected pairs; we define the map

$$f^{\uparrow} = \left[ret \mapsto \bigvee_{(\textcircled{c},y)\in C} \mathit{Live}(\textcircled{c})(y) \right]$$

The map $f^{\uparrow}$ is the *amalgamated call context* of the function f: it associates the reserved variable ret to 1 if any of the returned values is ever used. When analysing f, we handle `return` terms as per Figure 2.13. As a result, if x is live at any of the call points to f, then the returned variable is considered live at the return point inside the function f. Conversely, if x is not live at any of the call points to f, then the returned variable is not considered live at the return point inside the function f.

Note that the same caveat applies as with $f^{\downarrow}$ above: for recursive functions, we fixpoint $f^{\uparrow}$. The same technique is used.

2.6.4 Generalising: summary and amalgamated call-contexts

The improved LVA of Section 2.6.3 is *inter-procedural*: it takes into account information that flows between functions. Conversely, the basic LVA presented in Section 2.6.1 is *intra-procedural*: it analyses the body of each function in isolation.

Summary The first improvement for LVA presented above is the use of *function summaries*: $f^{\downarrow}$ is the summary of f. The following remarks apply.

First, distinct analyses have summaries of different forms. In the case of LVA, the summary of a function is a map from parameters to booleans.

Second, summaries are used by simple substitution (of parameter with arguments). As a result, every call to a given function f is handled similarly. We say, the summary is *context-insensitive* because it does not use the information available at the call point. We only use context-insensitive summaries in ASAP. Another approach is possible: a function can be summarised as a (computable) mathematical function from decorations to decorations.

Third, the order in which the summaries are computed was left implicit in the presentation above; we make it explicit now. Consider a function f which contains a call to a function g. To compute the summary of f, it is necessary to compute the decorations of the body of f, for which it is necessary to compute the summary of g. Thus, g must be summarised before f. As a result, summaries are computed in the topological order of the dependency graph of functions – we treat mutual recursion below. Note that the direction of the analysis (backward or forward) has no influence on this order. The direction of a data-flow analysis merely indicates the direction of information flow within a term, not across functions.

Fourth, consider a set of mutually recursive functions. The summary of each depends on the summary of at least another one, and by transitivity to all of them. Thus a fixpoint operation is necessary. This is similar to fixpointing the summary of a single recursive function, except that a summary is initialised for each function. A new summary is computed for each function using the initial values, and the process is repeated until all the summaries have reached a fixpoint. For brevity of exposition in this dissertation, we do not analyse mutually recursive functions (which µL does not support).

Fifth, the fixpoint operation described for liveness is simple. This simplicity follows from two facts: the solution space of LVA (maps from variables to booleans) forms a lattice, and the series of computed summaries ($f_i^{\downarrow}$) is monotonically increasing. These two facts are not a priori guaranteed for every analysis and the series of computed summaries $f_i^{\downarrow}$ may never reach a fixpoint. In such cases, an approximation need to be applied to the different iterations of the summary. *Widening* [13], an operation which coarsens the explored space into a finite lattice, is often chosen for this approximation. Widening can be seen as a lossy encoding of information: the result is less precise but of a more manageable size. (For more information about lattices and fixpoint termination, we refer the reader to the work of Khedker et al. in [23].)

Call context The second enhancement presented above is about *amalgamated call contexts* (or simply *call contexts*): $f^{\uparrow}$ is the amalgamated call context of f. It is the inverse of summaries: information flows from caller to callee. The following remarks apply.

First, distinct analyses have different forms of call context. For LVA, the call context only carries one bit of information: whether the return value is used by any of the callers. For other analyses, richer information can be represented.

Second, the order in which call contexts are computed is the reverse of summaries: the *reverse* topological order of the dependency graph of functions. This is for a simple reason: to compute the call context for a function, it is necessary to compute the decoration for all its callers, which require to compute the call context of all callers. As with summaries, the direction of the analysis (backward or forward) has no influence on this order.

Third, for recursive functions, the amalgamated call context is fixpointed. The same remarks as for summaries apply: fixpoints can require widening; mutually recursive functions are not treated here.

Fixpoint in 3VL The enhanced LVA illustrates how to fixpoint summaries and call contexts for a may-analysis. A must-analysis behaves similarly except for initialisation: the initial summary or call context is the most extreme over-approximation – instead of the most extreme under-approximation. That is, may-analyses initialise their fixpoint at 0 whilst must-analysis initialise their fixpoint at 1. Notice that the combination of 0 for a may- and 1 for a must-analysis, is inconsistent – see Figure 2.12. Instead of introducing a fourth value ($\perp$) for truthness, we use the empty map $[\,]$ as an initial value. This map is neutral for all the 3VL operators – as detailed in 2.4.3.

Note that $[\,]$ is used as the initial value for the summary and call context fixpoint operations; i.e., $f_0^{\downarrow} = f_0^{\uparrow} = [\,]$. This is different from the value used intra-procedurally at the entry point of a function's body (for a forward analysis) or return point (for a backwards analysis). This entry map is given by the amalgamated call context. In the case of the main function, which has no call context, the entry map is defined on a per-analysis basis.

2.6.5 Practical considerations

Consider equations that LVA can generate for a term. They involve: update ($\triangleleft$), domain restriction ($|$ and $\setminus$), composition ($\circ$) and operator lifting. Remember that, on maps, all these operations are computable: they can be handled algorithmically by a program.

Also, note that, in the presentation above, equations are all generated before they are all solved together. However, it is possible to solve the equations on the fly instead of acting in two distinct phases.

Chapter 3

Design space

We now explore the design space of existing memory management strategies. Note that we are not reviewing the design space of GCs, but the larger space of memory management strategies. Also note that we are mainly concerned with automatic strategies, but that we also briefly study manual strategies.

3.1 Memory Management Lexicon

As we already mentioned, memory management schemes are numerous: GCs of one kind or another, reference counters, and region and linear regimes. To explore the design space they reside in, we need to differentiate them by their behaviour and their properties – rather than by their implementation. To this end, we build a lexicon of memory management terms, some of which are already in use, others not.

3.1.1 Waste

For each state of a given execution of a given program, there are two distinct categories of memory blocks: those that will be dereferenced in further states of the execution and those that will not. We qualify as *waste* blocks of the latter category – and by extensions the values these blocks represent. We use "waste" both as an adjective (as in "a waste value") and noun (as in "there is waste on the heap"). We use "non-waste" to describe blocks and values that are not waste. (We avoid the word *garbage* because of its association with GCs.)

Note that waste is undecidable: it depends on branching behaviour. More precisely, it is impossible to characterise waste automatically (i.e., by a program), generally (i.e., for any state and any program), and exactly (i.e., to characterise all the waste and nothing more). This undecidability of waste is central to several of the definitions below and we explore it more thoroughly in Section 3.1.4.

Note that, given a finite trace and a given state in that trace, waste can be automatically, precisely characterised. However, this characterisation is a posteriori.

Reachability

Reachability is a property of memory cells in the heap which is distinct form the notion of waste/non-waste. However, because GCs use reachability as a proxy for non-waste, it is important to our purpose. More generally, a useful property of reachability is that is under-approximates waste.

A memory cell on the heap is *reachable* when the program can access it by following pointers in its roots. The *roots* form the set of values from which the memory graph can be accessed by a program. The roots include all values on the stack – which is always accessible from the program registers'. They also include external roots: values allocated outside of the heap – typically during a foreign function call. They also include global values.

Note that reachability depends on the operations that a programming language allows. E.g., in a language such as C with unlimited pointer arithmetic, a program may reach any cell of the heap. On the other hand, in a language such as Haskell where pointers are not available to the programmer, the only reachable cells are the one that represent part of a value in scope. In other words, the features of a programming language determines the way in which a program may access heap cells. This in turn determines which heap cells can a program legally reach. This, finally, determines whether the reachability is a useful and practical proxy for non-waste.

3.1.2 Strategy

A *memory management strategy* – or simply a *strategy* when context is clear – is a method or set of methods used to deallocate waste.

Note that we focus on values allocated on the heap rather than the stack. The latter category is managed by the stack discipline associated with function calls and returns – an issue out of our scope.

3.1.3 Correctness criteria

The two correctness criteria of memory management strategies are

Safe A memory management strategy is *safe* if, by design and construction, it cannot induce runtime errors. Such memory-management-induced runtime errors happen when non-waste (i.e., values that will be accessed in later states of the execution) is deallocated. This creates dangling pointers that are eventually accessed (by definition of non-waste), and causes a bug. Memory-management-related runtime errors also happen when a memory block is deallocated twice.

E.g., GCs are safe because, by design, they never create dangling pointers.

A strategy that is not safe is *unsafe*.

Complete A *memory leak* – or simply *leak* when context is clear – is a permanent accumulation of waste: memory blocks representing values which are not useful to the program accumulate indefinitely on the heap. Memory leaks can be caused by program patterns (e.g., if a program allocates data that is never safe to deallocate) or by a memory management strategy. In the latter case, the strategy does not deallocate data even though it is provably safe to do so: the leak is caused by the strategy rather than the program pattern. A strategy which can cause leaks we call *incomplete*, a strategy which cannot we call *complete*.

For programs that do not terminate (such as daemons or servers), leaks are always problematic. Indeed, the leaks fill the heap with more and more waste until inevitably the program runs out of memory.

For short-lived programs on the other hand, leaks are less of a serious issue: when the process exits, its memory is reclaimed by the operating system it runs in. We consider incomplete memory management strategies to be incorrect, even though they are usable for short-lived programs.

3.1.4 Waste undecidability

As mentioned in Section 3.1.1, in general, algorithmically and precisely characterising waste is not possible. More specifically, precisely characterising waste requires, in general, predicting branching which is beyond the reach of algorithms. Several solutions to this issue exist; we detail them now.

Approximating A strategy can forego precision and *approximate* waste in an algorithmically decidable way. E.g., GCs approximate waste by unreachability: they deallocate blocks that cannot be accessed by following pointers from the values stored on the stack.

In approximating strategies, waste must be under-approximated. I.e., it is never acceptable to deallocate non-waste values (because it causes bugs and crashes) whilst, contrariwise, delaying deallocation of waste is acceptable. However, note that under-approximating too much is a potential source of memory leaks (as is the case for conservative GCs).

Incorrect An incorrect strategy is one that can lead to leaks (i.e., an incomplete strategy; e.g., conservative GCs) or runtime errors (e.g., manual memory management *à la* C). We focus mostly on correct strategies and only consider incorrect ones as comparison points.

Restrictive A strategy can forego generality and *restrict* the source language to make waste easier to decide. E.g., linear type restrictions make waste characterisation trivial.

Some language restrictions can be based on purposeful limitations of the grammar – e.g., not providing language constructs for higher-order function. Other language restrictions are enforced by static type-like analyses.

Restrictions carried out by programmers (e.g., who choose not to use certain features of the language or patterns of code) can be considered both restrictive and manual. E.g., deciding not to use `malloc`.

Manual A strategy can forego algorithmicity and treat programmers as oracles that decide waste. Such strategies are *manual*, a classification we come back to in Section 3.1.6.

Note that strategies can combine several aspects of the solutions presented above. E.g., region-based memory management restricts aliasing across regions and approximates waste using region scope – the restriction guarantees the safety of the approximation.

3.1.5 Timely

Time elapses[1] between the instant a value becomes waste and the instant it is reclaimed by the memory management strategy. A memory management strategy is said to be *timely* when this delay is short. We call *untimely* the strategies that are not timely. Note that there are degrees of timeliness and untimeliness; thus, it is not useful to say a strategy “is timely” but it can be interesting to say it “is more timely than” a competing approach.

Timeliness is related to Röjemo and Runciman’s notion of Drag [34] but generalises to other approaches than GCs. More specifically, they introduce the word “dormant” to refer to reachable waste. Dormant blocks are the false-positives of GC: they are kept on the heap because they are reachable even though they should not because they are useless. In the words of the authors: “[dormant cells are] retained in heap memory though not actually playing a useful role in computation.” They also introduce the word “drag” to refer to the lifespan of dormant cells.

[1]Or more precisely, state transitions occur.

Timeliness affects the amount of memory a program needs – to store temporary values necessary for the program to progress. The more untimely a strategy, the more waste cells are kept on the heap, the more memory the program needs, the more likely the program is to run out of memory. From that respect, incompleteness is merely an extreme form of untimeliness. Specifically, it can have similar consequences: failure to deallocate values in a timely fashion increases memory requirements.

Timeliness of approximation In approximating approaches, there is a delay between a value becoming waste and it being characterisable as waste. Additionally, there is potentially delay between a value being characterisable (i.e., approximated) as waste and its memory being deallocated. However, only the former source of delay matters for timeliness. Indeed, the memory cells that are kept because of the latter source of delay can be reclaimed whenever more space is needed. As such, they do not impact the memory requirement of a program. E.g., for GCs, reclaimable memory blocks are kept until a collection cycle is triggered; this source of delay is simply an optimisation that batches memory management operations together to avoid repeated scanning.

Relatively timely Approximating strategies cannot be perfectly timely. Indeed, only when the waste is characterised as such can it be deallocated. Thus, the timeliness of an approximating strategy is related to the precision of the approximation. E.g., GCs' timeliness is directly tied to reachability. We say that a strategy is timely relative to a property (e.g., reachability, liveness, scope, etc.) if any untimeliness of the strategy corresponds exactly to the imprecision of the analysis.

Unlike approximating strategies, restrictive strategies such as linear-type systems can have perfect timeliness.

3.1.6 Actuators

Memory management can be effected by three distinct *actuators*: the programmer, the compiler, and the runtime. The programmer actuates memory management when the code they write explicitly handles memory deallocations. The compiler actuates memory management when it transforms code that does not handle memory deallocations into code that does. The runtime actuates memory management when it is linked to a program in order to handle its memory deallocations. This definition of the runtime actuator includes some libraries (such as Boost [1]) which are linked to the program and handle memory deallocations.

This characterisation by actuator subsumes two major, classic dichotomies: automatic-vs.-manual and static-vs.-dynamic.

Automatic: strategies that do not involve the programmer.

Manual: strategies that mostly involve the programmer.

Static: strategies that do not involve the runtime.

Dynamic: strategies that mostly involve the runtime.

Figure 3.1 summarises this subsumption.

Note that "automatic" and "manual" are the extremes of a spectrum: different strategies require different degrees of programmer involvement. The same remark applies to static and dynamic: strategies require varying degrees of runtime involvement. In fact, most approaches in-

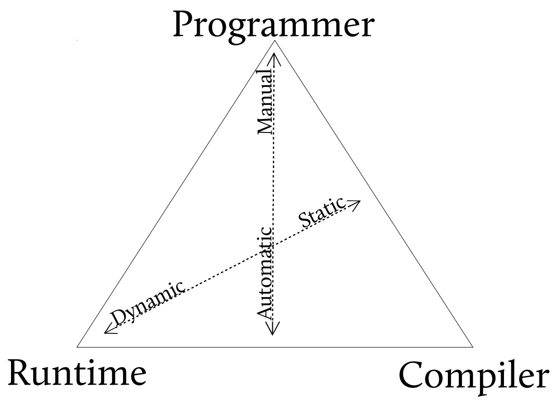


Figure 3.1: The actuator triangle subsuming classic dichotomies

volve several actuators. Even C, considered manual and static, relies on primitives provided by the runtime: `free` and `malloc`.

The actuator view is used in Section 3.4 in the design phase of a new strategy to distribute responsibility based on behavioural requirements.

3.1.7 Synchronous

We say a strategy is *synchronous* if the code that manages the memory runs at program points that are completely predictable ahead of the execution. E.g., linear types are synchronous: deallocations happen after last use; region systems are also synchronous: deallocations happen on end of scope.

A strategy that is not synchronous we call *asynchronous*. E.g., stop-the-world GCs are asynchronous: deallocations happen at allocation points (which are predictable) but only when the heap is full (which is not predictable).

3.2 Review of existing strategies

We now review existing strategies using our new-found vocabulary.

3.2.1 Manual memory management *à la* C

Note that C's approach to memory management is not automatic. Even though automatic strategies are our focus, it is still interesting to study, if only for its prevalence in the system-programming community.

As mentioned previously, C relies on a tiny fragment of runtime (essentially `free` and `malloc`) and a lot of work from the programmer. The runtime offers portability – across operating systems and architectures. It also provides a few features to ease memory management such as prefixing blocks with size information so they can be freed and word-aligning all blocks.

C-like strategies are neither restrictive nor approximating. They side-step this trade-off by relying on the programmer (rather than an algorithm) to characterise waste. Note that they offer no guarantees of correctness.

Programmers, or groups thereof, often set project- or company-wide policies or guidelines for a coherent manual strategy for managing memory. These guidelines are not enforced by the compiler but can be guarded by code reviews and tests. E.g., a project can use manual reference counting, where programmers (try to remember to) update references and deallocate values when appropriate.

Depending on the tooling available more advanced manual strategies are available. For example, David Gay Rob Ennals and Eric Brewer developed [17] to dynamically checked the safety of manual deallocation in existing code. Other tools include CPP which we explore in Section 3.3.1.

3.2.2 GC

Garbage collectors (GCs) are a family of strategies, all of which approximate waste by unreachability. Amongst this family of strategies are many variants we detail now.

Stop-the-world A stop-the-world GC is one to which the main program (the mutator) occasionally yields all control until, after the GC has completed a full collection, it is given back. Because it approximates waste by unreachability, GCs can exhibit poor timeliness. Specifically, many useless values can be reachable – as measured by Röjemo and Runciman [34]. Stop-the-world GCs are simple enough that safe, complete implementations abound.

In a language with a stop-the-world GC, the compiler translates constructors into special allocation instructions: calls to a routine in the runtime. This routine checks available space and either returns allocated space or, if none is available, transfers control to the GC per se to reclaim memory. Thus the runtime is the main agent of management with the compiler as a minor agent introducing runtime calls for each allocation.

Incremental, generational, concurrent, real-time There are multiple variations of the stop-the-world design above that aim to offer slightly different trade-offs in terms of frequency of pauses, length of pauses and amount of work the GC carries. Note that these approaches change the delay between a value becoming unreachable (i.e., approximated as waste) and deallocated, but do not affect timeliness: waste is still approximated by unreachability.

In an incremental GC, deallocation work is done in chunks instead of all in one go. As such, it yields control back and forth between mutator and GC more often. The result is shorter and more frequent pauses.

One specific type of incremental GC is called generational. In a generational GC, the heap is split in two: the minor and major heap. The minor heap is used for allocations and scanned as soon as it is full. Memory blocks that survive a minor heap collection are moved to the major heap. The major heap is only subject to collection when it becomes full.

The performance boost of generational GCs (compared to stop-the-world) relies on an empirically observed pattern that most values are short-lived. Under such conditions, long-lived objects get quickly moved to the major heap and are only occasionally scanned, thus reducing the workload of the GC.

In a concurrent GC, the scanning for unreachable values happens concurrently with the mutator. There can still be pauses when the collection happens, but they are shorter.

In a real-time GC, there are guaranteed bounds for pauses. These bounds allow programmers to give guarantees about the execution time of their own code.

Note that timeliness of generational GC is the same as that of stop-the-world GC (because the waste approximation, unreachability, is unchanged) even though the actual deallocations happen at different times. Deallocations of some values in the minor heap can be delayed because of references from the major heap. Even though these deallocations happen later (only after the major heap is collected), these values are still collectible. Indeed, were the program to run out of memory entirely, a major collection could be triggered that would reclaim the values.

In the case of a concurrent GC, the deallocations can be delayed for another reason. Specifically, the scanning starts earlier than the deallocation. As a result, when the program runs out of memory and a collection cycle is started, some of the reachability information is based on an outdated memory layout. In other words, the concurrent GC can start scanning roots that are later deleted by the mutator. The memory reachable from these roots is deallocated later (in terms of progress by the mutator) than with a stop-the-world GC.

Note that all these variations explore only a limited part of the design space of memory management strategies. Indeed, they all rely heavily on the runtime, they all use the same approximation of waste (unreachability), they all are non-restrictive. Also note that they are all safe and complete – that is, they all have safe and complete implementations.

Reference counters With a reference counter (RC), some of the responsibility is shifted away from the runtime to the compiler which inserts instructions after each mutation (as well as allocation). These instructions increment or decrement the reference counter associated with some values when the mutations add or remove references. The runtime is still used for the deallocations, especially when they cascade.

There are several effects on the run-time behaviour of a program. Chief amongst them is the reduction of delay: values are deallocated as soon as they become unreachable – cycles are dealt with below. Note however that the timeliness – limited by waste approximation – is the same as with the GCs described above.

Interestingly, naive RC is not able to detect cycles which causes leaks in contexts where cycles are allowed. These leaks are often fixed by appending a more standard GC [12] (i.e., one that relies almost entirely on the runtime). Some implementations require programmers to deal with cycles by breaking them explicitly (i.e., manually) once they are not needed. Some other implementations, such as the one proposed by Axford [5], require the programmers to use weak pointers [16]: pointers that can be dereferenced but are ignored for the purpose of detecting waste. Alternatively, some more clever RC implementations, such as the one proposed by Paz et al. [7], are able to detect cycles directly .

Liveness-assisted GC Even though it is not widely deployed, it is interesting to study the work of Asati, Sanyal, Karkare and Mycroft on liveness-assisted GC (LAGC) [4]. In their approach to memory management, the compiler performs usage analysis for values in the heap – similar to heap reference analysis [24]. When the compiler detects that a reference to a value (or to part of a value, say the left branch of a tree) is not used after a given program point, it inserts an instruction at that program point to set the reference to a null-pointer. This is safe because the compiler statically determined that the reference is not used afterwards. Then the program is attached to a GC.

The compiler transformation described above makes the execution-time approximation (unreachability) more precise. Indeed, values that can be statically characterised as waste are made unreachable, which in turn makes them deallocatable by the GC. As a result, timeliness is increased significantly which reduces the memory requirement of programs. A side-effect is to reduce the workload of the GC.

Note that it is common for GCs to use liveness information to reduce the amount of scanning. However, the rarer LAGC uses liveness of values in the heap which improves the granularity. Indeed, LAGC is able to detect that, say, only the left branch of a tree is used later, or, say, only the first element of a list is used later. This allows objects to be partially collected when portions of them are statically provably dead.

Using our vocabulary we say that LAGC shifts some of the weight from the runtime to the compiler. LAGC is a safe and complete strategy that increases the timeliness of GC by making more waste unreachable. The timeliness of this approach is tied to the precision of the liveness analysis: statically-decidably dead values are deallocated earlier.

3.2.3 Linear type systems

Linear type systems are a family of restrictions that make the characterisation of waste trivial. More specifically, they are type-like analyses that restrict value usage. Unlike most type systems, the usage restriction is not about distinguishing categories of data (integer, characters, booleans, etc.). Instead the restriction is such that values can only be used once.

After the one (final) use of a value, it becomes waste and can be safely deallocated. This statement is a vague and wide-spread over-simplification. We detail what "use" means in the statement above and what parts (if any) of the value are deallocated. Note that the details below are only one of the possible deallocation strategies for linear programs; variations exist.

- When a value is used as an argument to a function call (e.g., x in $f\,(x)$), the management of the memory representing that value is handed over to the callee. In other words, at the call point, the memory of the argument is untouched.
- When a value is used as an argument to the special function `ignore` it is deallocated. In that respect, linear type systems involve programmers in the same way C-like approaches do: they require explicit deallocations.
- When a value is used in a constructor (e.g., x in $\{F = x\}$ or in $D\,x$), it is copied into the newly allocated block. More specifically, the original value (held by x) is shallow-copied into the new value and its memory is untouched.
- When a value is used in an arithmetic or logic operator (e.g., x in $x + y$), the value is a word allocated on the stack which needs no management[2].
- When a value is matched against (e.g., x in `match`x`with` $[\{F = y;\ F' = z\}$ `-> ...` $]$), a pointer to each of the bound components of the value (y and z in the example) are placed on the stack[3] and the memory block for the value is deallocated. Thus, under linear types, destructors (`match`) are true complements of constructors in that they deallocate the memory allocated by the constructor.
- The type system forbids values to not be used before they fall out of scope. Programmers must use the special function `ignore` to circumvent these cases.

Linear type systems are safe, complete, restrictive strategies with optimal timeliness. Note, however, that it is sometimes necessary to add `ignore` instructions to satisfy the type system – as

[2]In the rare case where a programming language does not allocate words on the stack, it should be deallocated after the operator finishes.

[3]Word-sized inlined components are simply copied on the stack.

illustrated in Figure 1.2. These meaningless accesses, artificially extend the portion of the program the value is "useful" for.

The variants of linear type systems listed in Chapter 1, trade-off some timeliness for expressivity. In all variants, the compiler is the principal responsible actuator. The programmer is also involved: (re)writing the code within the bounds of linear-type systems.

3.2.4 Region-based memory management

Region regimes are a family of strategies in which values are allocated within a *region*. A type-like system tracks the region of each value and prevents values from escaping to an outer region. That is, references from outer regions into inner regions are prohibited. Whole regions are deallocated as soon as they fall out of scope.

Several implementations are possible. They can be incredibly efficient, especially if the runtime provides leverageable primitives to the compiler. E.g., Tofte et al. describe [38] an implementation where regions are divided in two kinds at compile time: finite and infinite regions. The former are represented at runtime by a fixed number of words on the stack. The latter as a linked list of memory pages: most allocations within the region increment an allocation pointer. When a page is full, a new page is requested from the operating system and appended to the list. Then, allocation proceeds within this new page. Deallocations are also fast: because they happen on a per region basis, it is sufficient to simply release each page in the list to the operating system.

Regions can either be specified by programmer annotations or inferred by the compiler. With the former method, programmers are directly involved in the memory management all along the development. One can easily believe that with the latter method, programmers are more oblivious to memory management. However, this is not so. Indeed, small changes in the code can drastically affect memory efficiency and timeliness – to the point of delaying some deallocations until the program exits. Patterns causing this untimeliness, referred to as region-unfriendly code, happen when values alias in ways that forces regions to be merged together. One of these patterns is shown in Figure 3.2a. In this example, the two lists passed to `sortedMerge` are entangled: their cells must be allocated in the same region. Even though `l2`'s elements are waste after `someComputation l3` returns, they are kept (with the elements of `l1`) until the program exits. As a result, programmers are indirectly involved with the memory management: they have to optimise the program into a region-friendly form to avoid poor performance. Rewriting programs for region-friendliness requires understanding of the region regime and assistance by profiling tools as Tofte and Talpin discuss [39]. Figure 3.2b shows how the code can be fixed to avoid entanglement: by manually introducing explicit copy instructions.

Thus, region regimes involve all three actuators: programmer, runtime, and compiler all contribute some effort to the memory management. Region regimes are both restrictive and approximative: programs are restricted (no outer-to-inner references) to ease the approximation (region is out-of-scope) of waste. Regions are safe – because of the restrictions on references –, complete – because regions are deallocated when they fall out of scope – and synchronous – because regions fall out of scope at known program points. Their timeliness depends on the programmer's savviness.

3.3 Review of existing programming languages

We now review a few programming languages and comment on their approach to memory management using the lexicon above.

```
let rec sortedMerge l1 l2 =
  match (l1,l2) with
  | [], l | l, [] -> l
  | x::xs, y::ys ->
    if x < y then
      x::sortedMerge xs l2
    else
      y::sortedMerge l1 ys
;;
let f () =
  let l1 = mkSortedList () in
  let l2 = mkSortedList () in
  let l3 = sortedMerge l1 l2 in
  someComputation l3;
  let n = moreComputation () in
  exit (nth n l1)
;;
```

(a) Pathological code (in OCaml)

```
let rec sortedMerge l1 l2 =
  match (l1,l2) with
  | [], l | l, [] -> copy l
  | x::xs, y::ys ->
    if x < y then
      (copy x)::sortedMerge xs l2
    else
      (copy y)::sortedMerge l1 ys
;;
let f () =
  let l1 = mkSortedList () in
  let l2 = mkSortedList () in
  let l3 = sortedMerge l1 l2 in
  someComputation l3;
  let n = moreComputation () in
  exit (nth n l1)
;;
```

(b) Manually fixed code (in OCaml)

Figure 3.2: Region-unfriendly code: `l1` and `l2` are entangled by `sortedMerge`

3.3.1 C

The programming language C uses the memory management detailed in Section 3.2.1. It is a manual strategy where programmers are given a few runtime functions (essentially `free` and `malloc`) and the responsibility to handle memory on their own.

Note that, amongst C programmers, best practices have emerged so that individual programmers are not left to their own devices. One of the informal guidelines to memory management in C is to pass pointers for values the caller is responsible for and pass values for values the callee is responsible for. This rule standardises the way functions share responsibility and simplifies the programmers' task of managing memory.

There are more advanced techniques of managing memory in C. One of them relies on CPP, the C pre-processor. Using CPP, it is possible for programmers to extend the compiler in limited ways – only syntactic transformations, or *macros*, are available. Macros let the programmers shift some of the work of memory management to the compiler. For example, different macros can be used for the different tasks of reference counting: increment and decrement counters, compare them to zero and deallocate values.

Several projects written in C use a garbage collector such as the Boehm GC [3]. Note that in C it is not generally possible to distinguish a pointer from an integer. Thus, GCs for C are of the conservative kind (they treat every word of memory as a potential pointer) which can cause leaks.

3.3.2 C++

The programming language C++ provides abstractions on to of C. Some of these abstractions relate to memory management.

The programmer can set custom initialisers and destructors for objects. These initialisers and destructors are called automatically when the value is created or deleted. This can happen in different ways.

By default, values associated to variables that are local to a function, are automatically deallocated before the function returns. More generally, the value of a variable with a limited scope is automatically deallocated at the end of scope. When the scope is a syntactic construct (e.g., the variables local to a function) the deallocation point is known. When the scope is a dynamic property (e.g., the members of an object are needed until the object is deallocated), the deallocation point is unknown.

If needed, the programmer can manually use the methods `new` and `delete`. This is akin to C's `malloc` and `free`. However, C++'s `new` and `delete` automatically take care of initialising or deleting the references held by the created/deleted object. E.g., when deleting a collection, the destructor (not the programmer) is responsible for calling the members' destructors.

3.3.3 ML, Haskell, Java, Go, Javascript, Lisp, Ruby, Python, etc.

These programming languages (and many other) use a GC of some kind. We do not list each implementation and the corresponding GC variant. Section 3.2.2 has more details about GCs.

3.3.4 Swift, Objective-C

Both Swift and Objective-C use RC to manage memory. Compilers for these languages introduce instructions to increment, decrement, and test reference counters on every block when necessary. Note, however, that no effort is made to detect cycles. Instead, programmers are required to use weak pointers.

3.3.5 Rust

Rust's approach to memory management relies on the concept of *ownership*. Programmers are required to provide annotations to indicate how ownership of arguments transfers from caller to callee. The compiler performs a type-like analysis to guarantee the ownership regime is not breached.

When passing a value as is, the ownership is *transferred* to the callee – it is also said the value is *passed*, *moved* or *consumed*. In this case, the caller is prevented, by the type-like analysis, from accessing the value after the call. The callee is now responsible for managing the memory of the value. The compiler checks the programmer respects the ownership rules and introduces deallocation instructions accordingly. Thus, transferring ownership treats values as a linear type system would: once a value is used (here as an argument in a call) it becomes unavailable.

On the other hand, when passing a value, the programmer can prefix its identifier with `&`[4], in which case the value is *lent* to the callee – or, more idiomatically, the value is *borrowed* by the callee. The value is still accessible to the caller who is responsible for deallocating it. The Rust book[5] states the following rule: "any borrow must last for a scope no greater than that of the owner." The "scope" is similar to a region and the rule states that values cannot escape their region. Thus, borrowed references treat values as a region regime would.

Rust offers distinct ways for programmers to bend these rules when necessary. Firstly, implementing the `Copy` trait for a type allows the caller to implicitly copy values it transfers ownership of. That is, if a type is given the `Copy` trait, it is possible to pass values of this type as a function argument and use it afterwards.

[4]We do not distinguish `&` (read-only borrow) and `&mut` (read-write borrow) here. Indeed, for the purpose of memory management, they are equivalent.

[5]`https://doc.rust-lang.org/stable/book/references-and-borrowing.html#the-rules`

Secondly, Rust provides, as part of the standard library, primitives to wrap values in a reference-counted block. Using the module Rc, programmers can bypass the ownership rules and rely on RC for memory management. In this case, it is the responsibility of the programmer to break cycles using weak pointers.

Thirdly, Rust lets programmers write unsafe blocks of code, explicitly annotated as such. In an unsafe block, the ownership rules are essentially ignored: the programmer is in charge.

3.3.6 Mercury

The Mercury programming language is a logic language (in the style of Prolog, which was used to bootstrap the Mercury compiler). Unlike most logic programming languages, Mercury has a strong type system used both to help programmers avoid bugs and to help the compiler optimise code.

Mercury also stands out from other programming languages for its memory management strategy: the compiler transforms the code to re-use dead memory blocks. A detailed description can be found in [29]. This compiler transformation reduces the workload of the garbage collector: more re-use means fewer allocations means fewer collections. In other words, Mercury shifts some of the responsibility from the runtime to the compiler.

It is relevant to our work for an additional reason: the Mercury compiler chooses which blocks to re-use based on a series of analyses; these analyses are similar to the ones carried out by ASAP (Chapter 5). This similarity hints to the fact that ASAP could also introduce memory re-use in the programs it processes.

3.3.7 Mezzo

Mezzo [33] is a programming language with a type system for managing state. Whilst this type system leaves allocations and deallocations entirely abstracted from the programmer, it offers a sound basis for manual re-use of memory. Not only are mutations allowed, it is also possible to safely store, in a single variable, values of different types at different points of the program. This lets the programmer re-use memory blocks even if the dead and fresh values are of different types. It also lets the programmer nullify pointers when the type system can verify it is safe to do so. Indeed, by destructively updating a variable (or a component of a value) to the unit value, the programmer removes a reference to the original value held by the variable. This is similar to the transformation LAGC performs.

The type system in Mezzo has other uses (such as guaranteeing safety in concurrent programs that use mutation). However, as we are interested in memory management, we focus on the re-use and pointer-nullifying aspects.

In Mezzo, the compiler checks, through a type-like analysis, that the mutations inserted by the programmer (whether for re-use or otherwise) are safe. Then, the memory is managed by a GC (the Mezzo compiler uses OCaml as a backend).

3.3.8 Cyclone

The Cyclone programming language ([20]) emulates the style of C whilst providing safety guarantees. The similarities with C – imperative features, pointer arithmetic, unions and structs, etc. – allows the many C programmers to quickly learn Cyclone.

In order to guarantee the safety against such errors as read-after-free common in C, Cyclone provides memory management features. Specifically, Cyclone provides manual deallocation for

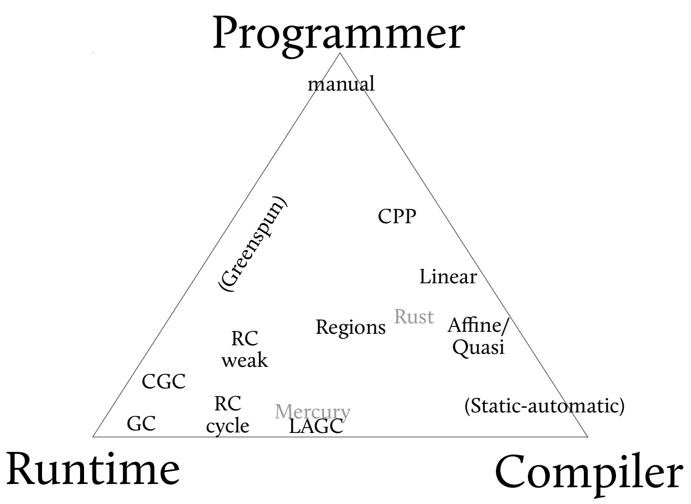


Figure 3.3: The actuator triangle with existing approaches plotted

unique pointers. These pointers follow linear-like restrictions which guarantees their manual deallocation can always be checked by the compiler for safety. Cyclone also provides region-based memory management as well as opt-out conservative garbage collection.

3.4 Gaps in space

Considering the design space as a whole and each existing strategy within it, we notice a few gaps. We make the design space, and, negatively, the gaps in the space, more visual in Figure 3.3: the memory management strategies are plotted in the actuator triangle (introduced in Figure 3.1). The closer a strategy is to an actuator corner, the more the actuator is involved in the strategy. Strategies involving mostly a single actuator (e.g., GC) are tucked away in the corresponding corner. Strategies involving two actuators (e.g., LAGC) are close to an edge. Strategies involving all three actuators are placed somewhere in the middle.

The “manual” entry encompasses all the manual strategies, including those where the programmer manually performs RC. The “RC weak” entry corresponds to RC where the programmer is responsible for breaking cycles using weak references whilst the “RC cycle” entry is for approaches where the runtime detects and gets rid of cycles. The “LAGC” entry is for liveness-assisted GC and “CGC” for conservative GC. The “CPP” entry refers to the use of a pre-processor to assist the programmer in managing the memory as mentioned in Section 3.3.1. The “Linear” entry is for linear strict types whilst the “Affine/Quasi” entry is for affine and quasi-linear type variants. We also included two noteworthy languages, Rust and Mercury. The gaps are in brackets: “(Greenspun)” and “(static-automatic)”. We detail them below.

Note that the placement of strategies in the actuator triangle is diagrammatic (showing the relative importance of actuators) rather than quantitative. Moreover, each of the strategies named therein represents, in fact, a family of approaches to memory management and should, therefore, cover an area of the diagram. For readability on a static, two-dimensional medium, this is omitted.

Also note that this representation only displays one variable of the design space: the distribution of work amongst the actuators. We present other interesting characteristics in Figure 3.4.

3.4.1 Greenspun

Philip Greenspun coined a saying known as Greenspun's tenth rule[6]:

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp.

The famous sentence refers to the habit of C programmers to write more and more runtime code as a project grows, in an attempt to standardise and automate memory management as well as other features lacking in C. This runtime code, derisively described by Greenspun, quickly grows to include a GC of its own. It is similar to the CPP-based approach when programmers rely on the runtime instead of the compiler.

3.4.2 A promising gap: static-automatic

There is an interesting gap in the compiler corner of the diagram, just under linear type systems and right of LAGC and Mercury. Thus it is safe to assume there are other, as yet undiscovered strategies in that part of the diagram. Because of its position in the design space, we name this gap *static-automatic*.

There are two reasons why the static-automatic gap is interesting to explore. First, strategies in this part of the design space do not involve the programmer. As such, we can assume most of them are correct-by-design. (Note how unsafe strategies – C, CPP, Greenspun – involve the (error-prone) human actuator. The exception to this rule is CGC; interestingly the Boehm collector, a widely deployed conservative GC, was designed for backwards compatibility with the manual strategy of C.) Second, strategies in this part of the design space do not involve the runtime. As such, we can assume most of them do not tamper with the representation of values in memory. (Note that the strategies that tamper with value representation – Regions, RC,GC, LAGC – tend to involve the runtime more.)

With this in mind, it becomes obvious that, in the context of safe system programming, the static-automatic gap is worth exploring. Indeed, as noted previously, control over memory representation and correctness are important to system programmers.

Several questions can be immediately put forward. What approximation of waste can be used? (Remember that automatic, non-restrictive strategies must approximate waste.) What degree of timeliness can be achieved? How much work can be concentrated in the compiler, how much needs to be distributed to the other actuators?

The rest of this dissertation is about ASAP: a memory management strategy that fits in the static-automatic gap. After detailing the inner working of ASAP, we argue that it is a viable lead to develop the memory management strategy of a safe, system programming language.

[6] `http://philip.greenspun.com/research/`

Strategy	Agnostic(👍)	Correct(👍)	Restrictive(👎)	Synchronous	Approximative	Re-use(👍)
À la C	✓			✓	N/A	✓(a)
GC		✓			✓(b)	
LAGC		✓			✓(c)	
CGC	✓				✓(d)	
linear	✓	✓	✓	✓	(e)	
region	✓	✓(f)	✓	✓	✓(g)	
Rust	✓	✓	✓	✓	✓(h)	
Mercury		✓			✓(i)	✓(j)
Mezzo		✓	(k)		✓(l)	✓(m)

(a) manual re-use
(b) unreachability
(c) unreachability (statically provably dead edges are removed)
(d) unreachability (all words are treated as edges)
(e) usefulness of values is artificially extended by `ignore`
(f) altough entangled regions are similar to leaks
(g) scope of region
(h) liveness (aided by ownership annotations)
(i) unreachability
(j) automatic re-use
(k) manual optimisations are accepted only if the type checker can prove they are safe
(l) unreachability (some dead edges can be removed by the programmer)
(m) manual re-use

Figure 3.4: Summary of strategies and their characteristics

Chapter 4

Paths

In order to fill the static-automatic gap (presented in Chapter 3), we must design a memory management strategy as a compiler pass. This task requires static analysis of heap properties. When analysing properties of the heap, it is necessary to have an efficient compile-time abstraction of heap objects. Indeed, the heap can take an infinite number of configurations: any number of objects can be pointing to each other in any number of ways. For the purpose of analysis, an abstraction with a bounded-size representation is more useful than an exact representation with unbounded size.

Thus, *paths*: bounded-size[1] representations of heap structures. In our analyses (Chapter 5), paths are used at compile-time to approximate sets of sequences of actual, execution-time accesses.

We first formalise paths and define some operations necessary for our analyses. We then show how to generate, at compile-time for any given path, code which scans, at execution-time, through the heap along this path. Because this code is generated with full knowledge of types, it is able to scan values even without runtime types.

Comparison with similar work

Paths are similar to the notion of access graphs from the work of Khedker, Sanyal and Karkare [24]. They essentially fill the same role: describing the shape of objects on the heap; but they were designed for different aims and thus fill distinct roles.

Specifically – and as detailed below – paths are designed to be used for multiple analyses. As a result they are more expressive and are equipped with a more complete set of operations – including a widening operation to ensure termination of analyses that use a fixpoint. On the other hand, access graphs in [24] are designed to compute liveness of heap references – i.e., detect which pointers to a heap object are used by the program later on. As a result, access graphs are more specialised.

The formalisation of paths (using a regular-expression-like form with a Kleene star (*), alternative ($+$) and sequence ($\cdot$)), the operations (partial order, prefix closure, widening, etc.), as well as the code generation (through $SCAN$) are my own.

4.1 Formalisation

A *path* is a regular expression over field and discriminant identifiers. They are used to describe sets of sequences of memory accesses. E.g., in Figure 4.1 the path first represents an access to

[1]As detailed in Section 4.3.5, for a value of type α, our analyses handles paths of size up to $O(2^s)$ where s is the size of the type definition for α.

$$\begin{array}{rcl}
\mathit{unit} & = & \{\} \\
\mathit{pair} & = & \{\mathit{Left} : \mathit{word};\ \mathit{Right} : \mathit{word}\} \\
\mathit{cons} & = & \{\mathit{Head} : \mathit{pair};\ \mathit{Tail} : \mathit{list}\} \\
\mathit{list} & = & \mathit{Cons\ cons} + \mathit{Nil\ unit} \\
\\
\mathit{first} & = & \mathit{Cons} \cdot \mathit{Head} \\
\mathit{elems} & = & (\mathit{Cons} \cdot \mathit{Tail})^* \cdot \mathit{Cons} \cdot \mathit{Head} \\
\mathit{spine} & = & (\mathit{Cons} \cdot \mathit{Tail})^* \cdot \mathit{Cons} \\
\mathit{nil} & = & (\mathit{Cons} \cdot \mathit{Tail})^* \cdot \mathit{Nil}
\end{array}$$

Figure 4.1: Example of paths for the *list* type

$$\begin{array}{rcll}
\mathit{path} & ::= & \epsilon & \\
& | & \mathit{fieldname} & \\
& | & \mathit{discriminantname} & \\
& | & \mathit{path} \cdot \mathit{path} & \text{(sequence)} \\
& | & \mathit{path} + \mathit{path} & \text{(alternative)} \\
& | & \mathit{path}^* & \text{(repetition)} \\
\mathit{zone} & ::= & (\mathit{variable}, \mathit{path}) &
\end{array}$$

Figure 4.2: The grammar of paths

the first element of a list; the path *nil* represents a recursive descent in a list (denoted by the repetition *) to the nil block. More details follow.

4.1.1 Grammar

The grammar of paths and zones is defined in Figure 4.2. A field identifier ($F \in \mathit{fieldname}$) indicates a dereference to the corresponding field. A discriminant identifier ($D \in \mathit{discriminantname}$) corresponds to a dynamic check to determine what element of a sum is being considered – more below.

Note that the formal definition of paths, as well as each of the auxiliary definitions below, depends on the type system for the considered language. Specifically, paths include identifiers from *fieldname* and *discriminantname*. We only present paths for the values of µL because they are the ones we use in ASAP. However, we posit the path abstraction can be adapted to other languages with other type systems.

4.1.2 Type compatibility

The operator $\alpha.p$, where α is a type name and p is a path, is defined in Figure 4.3a where the global map $\Delta : \mathit{typename} \rightarrow \tau$ associates type names with their definitions. The result of $\alpha.p$ is the type of the values represented by memory blocks reached from values of type α following sequences of dereference recognised by p. E.g., $\mathit{list}.\mathit{Cons}$ is the type cons, $\mathit{list}.\mathit{Cons} \cdot \mathit{Tail}$ is the type list, and $\mathit{list}.(\mathit{Cons} \cdot \mathit{Tail})^*$ is also the type list.

Note that the operator is partial: not all paths apply to all types. Specifically, $\alpha.p$ is defined only when p recognises sequences of dereferences that can be applied on values of type α. E.g., a path describing a descent into a tree does not apply to the type of lists.

$$\begin{array}{rcll}
\_.\_ & : & \mathit{typename} \times \mathit{path} \to \mathit{typename} & \\
\alpha.\epsilon & = & \alpha & \\
\alpha.F & = & \beta & \text{if } \Delta(\alpha) = \{\ldots ; F : \beta; \ldots\} \\
\alpha.D & = & \beta & \text{if } \Delta(\alpha) = \cdots + D\,\beta + \ldots \\
\alpha.(p_1 \cdot p_2) & = & (\alpha.p_1).p_2 & \\
\alpha.(p_1 + p_2) & = & \beta & \text{if } \alpha.p_1 = \alpha.p_2 = \beta \\
\alpha.p^* & = & \alpha & \text{if } \alpha.p = \alpha \\
\alpha.p & & \text{undefined otherwise} &
\end{array}$$

(a) Applying paths to types

$$\begin{array}{rcl}
\mathit{compatible}(p_1, p_2) & \iff & \forall \alpha, \alpha.p_1 = \alpha.p_2 \vee (\alpha.p_1 \text{ and } \alpha.p_2 \text{ undefined}) \\
\mathit{compatible}(P) & \iff & \forall p, q \in P, \mathit{compatible}(p, q)
\end{array}$$

(b) Type compatibility between pairs or sets of paths

Figure 4.3: Path application and type compatibility

$$\begin{array}{rcl}
\multicolumn{3}{c}{Z : \mathit{location} \times \mathit{typename} \times \mathit{path} \to \mathit{heap} \times \mathit{stack} \to 2^{\mathit{location}}} \\
Z(l, \alpha, \epsilon)(\eta, \sigma) & = & \{l\} \\
Z(l, \alpha, F)(\eta, \sigma) & = & \begin{cases} \emptyset & \text{if } \alpha.F = \mathit{word} \\ \{\pi_F(l)\} & \text{otherwise} \end{cases} \\
Z(l, \alpha, D)(\eta, \sigma) & = & \begin{cases} \emptyset & \text{if } \alpha.D = \mathit{word} \\ \emptyset & \text{if the variant at } l \text{ is not } D \\ \{\pi_D(l)\} & \text{otherwise} \end{cases} \\
Z(l, \alpha, p \cdot p')(\eta, \sigma) & = & \left\{ Z(l', \alpha', p')(\eta, \sigma) \;\middle|\; \begin{array}{l} \alpha' = \alpha.p, \\ l' \in Z(l, \alpha, p)(\eta, \sigma) \end{array} \right\} \\
Z(l, \alpha, p + p')(\eta, \sigma) & = & Z(l, \alpha, p)(\eta, \sigma) \cup Z(l, \alpha, p')(\eta, \sigma) \\
Z(l, \alpha, p^*)(\eta, \sigma) & = & \bigcup_{n \in \mathbb{N}} Z_n \\
& & \text{where } \left\{ \begin{array}{l} Z_0 = Z(l, \alpha, \epsilon)(\eta, \sigma) \\ Z_{n+1} = \{Z(l', \alpha, p)(\eta, \sigma) \mid l' \in Z_n\} \end{array} \right.
\end{array}$$

Figure 4.4: The set of locations $Z(l, \alpha, p)(\eta, \sigma)$

We say that two paths p_1 and p_2 are type-compatible when they apply to all types similarly. We write $\mathit{compatible}(p_1, p_2)$ for compatibility between p_1 and p_2 and $\mathit{compatible}(P)$ for pairwise compatibility of paths in the set P. Type-compatibility is defined in Figure 4.3b.

4.1.3 Zones during execution

Given a location l of a value of type α, a path p, a heap η and a stack σ, we write $Z(l, \alpha, p)(\eta, \sigma)$ for the set of memory blocks reachable from l, following sequences of dereferences recognised by the path p. It is formally defined in Figure 4.4. In the definition, we write π_F for the projection for the field F – remember that blocks are tuples of $\mathit{address} \cup \mathit{word}$ – and we also write π_D for the projection for the variant D: for a memory block at location l that represents the value $D\ x$, the location of x is $\pi_D(l)$.

Notice how some arguments of Z are available at compile-time but others are only available at execution-time. We implement a staged function that corresponds to Z in Section 4.4.

$$\begin{array}{rcl}
p \preceq p' & \Leftarrow & p = p' \\
\epsilon \preceq p^* & \Leftarrow & \mathit{compatible}(p, p \cdot p) \\
p_1 + \cdots + p_n \preceq p'_1 + \cdots + p'_m & \Leftarrow & \forall i \leq n, \exists j \leq m, p_i \preceq p'_j \\
p^* \preceq p'^* & \Leftarrow & p \preceq p' \\
p \cdot q \preceq p^* & \Leftarrow & q \preceq p^* \\
p_1 \cdot p_2 \preceq p'_1 \cdot p'_2 & \Leftarrow & p_1 \preceq p'_1 \text{ and } p_2 \preceq p'_2
\end{array}$$

Figure 4.5: Axioms of path comparison

We overload Z and often simply write $Z(x,p)(\eta,\sigma)$ for the more complete form $Z(\sigma(x),\Gamma(x),p)(\eta,\sigma)$ – remember that Γ maps variables to their type. This overloading is more than purely æsthetic: in the shorter form we use the compile-time identifier for the scanned value (i.e., the variable x) rather than the execution-time location (l). Thus, we can talk at compile-time about the set of memory blocks in a zone (x,p) and defer the location resolution ($\sigma(x)$) to the execution. Specifically, it allows us to use $Z(z)$ (where z is a zone) to describe, during analyses, properties of the heap.

4.2 Examples

Consider a variable x of type *list* and the paths *first*, *elems* and *spine* defined in Figure 4.1.

The set $Z(x, \mathit{first})(\eta,\sigma)$ is empty if the list x is empty, or it contains the first element of the list x otherwise. Also note that $\mathit{list}.\mathit{first} = \mathit{pair}$, which coincides with the fact that the first element of the list is of type *pair*.

The set $Z(x, \mathit{elems})(\eta,\sigma)$ contains all the elements of the list x. Once again, the type $\mathit{list}.\mathit{elems} = \mathit{pair}$ coincides with the type of the elements of the list.

Finally, the set $Z(x, \mathit{spine})(\eta,\sigma)$ contains all the cons cells of the list x.

4.3 Use for analysis

In order to carry out the heap analyses of ASAP in Chapter 5, we need to provide a few operations on paths. Specifically, our analyses rely on fixpoints in the 3VL-sets of zones ($3^{(\mathit{variable} \times \mathit{path})}$).

A common pattern during the fixpoint operations is the accumulation of values such as $\{(x,\epsilon), (x,p), (x,p{\cdot}p), (x,p{\cdot}p{\cdot}p), \ldots\}$. We define a widening operator *Widen* that collapses these accumulations into $\{(x,p^*)\}$. These accumulations appear when analysing a recursive function that descends into a value (of a recursive type) along the path p. E.g., a function that computes the length of a list will cause the accumulation $\{(x,\epsilon), (x, \mathit{Cons} \cdot \mathit{Tail}), (x, \mathit{Cons} \cdot \mathit{Tail} \cdot \mathit{Cons} \cdot \mathit{Tail}), \ldots\}$ and is collapsed into $\{(x, (\mathit{Cons} \cdot \mathit{Tail})^*)\}$.

Before we can define the widening operator, we need several helper functions which we define now.

4.3.1 Partial order

We define the partial order $\preceq$ on paths in Figure 4.5. By design, the definition is such that $p \preceq p' \Rightarrow \forall x, \eta, \sigma, Z(x,p)(\eta,\sigma) \subseteq Z(x,p')(\eta,\sigma)$.

Note that Figure 4.5 hints at the implementation of the comparison: it reads as pseudo-Prolog where the left-hand side is destructed and the (smaller) right-hand side is tested recursively.

$$\begin{array}{rcl}
\mathit{Prefix} : \mathit{path} & \to & 2^{\mathit{path}} \\
\mathit{Prefix}(\epsilon) & = & \{\epsilon\} \\
\mathit{Prefix}(F) & = & \{F, \epsilon\} \\
\mathit{Prefix}(D) & = & \{D, \epsilon\} \\
\mathit{Prefix}(p \cdot q) & = & \mathit{Prefix}(p) \cup \{p \cdot q' \mid q' \in \mathit{Prefix}(q)\} \\
\mathit{Prefix}(p + q) & = & \mathit{Prefix}(p) \cup \mathit{Prefix}(q) \\
\mathit{Prefix}(p^*) & = & \mathit{Prefix}(p) \cup \{p^* \cdot p' \mid p' \in \mathit{Prefix}(p)\}
\end{array}$$

Figure 4.6: Prefix of path

Also note that in the sequence case ($p_1 \cdot p_2 \preceq p_1' \cdot p_2'$) it is only ever useful to consider cases where p_1 and p_1' are type compatible because the order $\preceq$ is only defined on type compatible pairs of paths. This consideration is useful when comparing long sequences, to decide where to split the paths.

4.3.2 Prefix closure

We define the function *Prefix* as follows: $\forall p \in \mathit{path}$, $\mathit{Prefix}(p)$ is a set of paths such that $\bigcup_{q \in \mathit{Prefix}(p)} L(q)$ is the language which contains all the prefixes of words of $L(p)$. The function is defined in Figure 4.6.

We often use $\mathit{Prefix}(p)$ (which is technically a set of paths) where a path is expected – e.g., we write $p \cdot \mathit{Prefix}(q)$ or $[(x, \mathit{Prefix}(p)) \mapsto \top]$. The intuition behind this abuse of notation is that the set of paths is used instead of the alternation of its elements – i.e., $p \cdot \mathit{Prefix}(q)$ stands for $p \cdot \Sigma \mathit{Prefix}(q)$ where $\Sigma\{p_1, \ldots, p_n\} = p_1 + \ldots + p_n$.

Note, however, that the elements of $\mathit{Prefix}(p)$ are not type-compatible in general – e.g., the prefix for the path selecting the elements of a list will select blocks that represent the elements of the list, but also blocks that represent the cons cells of the list. Given incompatible types $\{p_1, \ldots, p_n\}$, the application $\alpha.p_1 + \ldots + p_n$ is undefined for any type name α. Thus we give an arguably less intuitive but better-behaved interpretation of the set-of-paths-as-path notation: we write $p \cdot \mathit{Prefix}(q)$ for the notationally correct $\{p \cdot q' \mid q' \in \mathit{Prefix}(q)\}$ and we write $[(x, \mathit{Prefix}(p)) \mapsto \top]$ for $[(x, p') \mapsto \top \mid p' \in \mathit{Prefix}(p)]$. In other words, we use a set of paths as a path where a guard in an intensional definition would be more correct.

4.3.3 Wild path set

We write $\mathit{Wild}(\alpha)$ for the *wild* path set of the type α. The wild path set of α is a set of paths that, collectively, explores the whole memory occupied by a value of type α. More formally, we require that $\forall \eta, \sigma, \bigcup_{p \in \mathit{Wild}(\Gamma(x))} Z(x, p)(\eta, \sigma)$ comprises all the memory locations that are reachable from x. (Note, that this property does not define a unique wild path set for each type. However, we show below how to compute a particular solution, which we call *the* wild path set.)

Note that, just as with prefix closure, the different paths of a wild path set are not, in general, type-compatible. Indeed, the memory blocks that constitute a value, say a list, represent values of different types, say cons cells and elements. As a result, we use the same notation as with prefix sets: we write $[(x, \mathit{Wild}(\Gamma(x))) \mapsto \top]$ for $[(x, p) \mapsto \top \mid p \in \mathit{Wild}(\Gamma(x))]$.

Example: trees of lists of pairs We first show the wild path set for a simple but non-trivial example: a type including several tiers and recursions. The type *tree*, along with the other inter-

$$
\begin{aligned}
&\mathit{unit} = \{\}\\
&\mathit{pair} = \{\mathit{Left} : \ \mathit{word};\ \mathit{Right} : \ \mathit{word}\}\\[1em]
&\mathit{cons} = \{\mathit{Head} : \ \mathit{pair};\ \mathit{Tail} : \ \mathit{list}\}\\
&\mathit{list} = \mathit{Cons}\ \mathit{cons} + \mathit{Nil}\ \mathit{unit}\\[1em]
&\mathit{node} = \{\mathit{LBr} : \ \mathit{tree};\ \mathit{Val} : \ \mathit{list};\ \mathit{RBr} : \ \mathit{tree}\}\\
&\mathit{tree} = \mathit{Node}\ \mathit{node} + \mathit{Leaf}\ \mathit{unit}
\end{aligned}
$$

(a) Definition of the *tree* type

$$\begin{tikzcd}
\mathit{tree} \arrow[r, "\mathit{Node}"] \arrow[dr, "\mathit{Leaf}"'] & \mathit{node} \arrow[l, bend right, "{\mathit{LBr},\mathit{RBr}}"'] \arrow[r, "\mathit{Val}"] & \mathit{list} \arrow[r, "\mathit{Cons}"] \arrow[dl, "\mathit{Nil}"] & \mathit{cons} \arrow[l, bend right, "\mathit{Tail}"'] \arrow[r, "\mathit{Head}"] & \mathit{pair} \arrow[r, bend left, "\mathit{Left}"] \arrow[r, bend right, "\mathit{Right}"'] & \mathit{word} \\
& \mathit{unit} & & & &
\end{tikzcd}$$

(b) Graph representation of the *tree* type

$$
\begin{array}{rcl}
\mathit{Wild}(\mathit{pair}) & = & \{\epsilon, \mathit{Left}, \mathit{Right}, \mathit{Left} + \mathit{Right}\}\\[1em]
\mathit{Wild}(\mathit{list}) & = & \mathit{suffixes} \cup \mathit{descending}\\
\text{where } \mathit{suffixes} & = & \{\epsilon, \mathit{Nil}, \mathit{Cons}, \mathit{Cons} \cdot \mathit{Tail}\}\\
 & & \quad \cup\{\mathit{Cons} \cdot \mathit{Head} \cdot p \mid p \in \mathit{Wild}(\mathit{pair})\}\\
\text{and } \mathit{descending} & = & \{(\mathit{Cons} \cdot \mathit{Tail})^* \cdot s \mid s \in \mathit{suffixes}\}\\[1em]
\mathit{Wild}(\mathit{tree}) & = & \mathit{suffixes} \cup \mathit{descending}\\
\text{where } \mathit{suffixes} & = & \{\epsilon, \mathit{Leaf}, \mathit{Node}, \mathit{Node} \cdot \mathit{RBr}, \mathit{Node} \cdot \mathit{LBr},\\
 & & \quad \mathit{Node} \cdot (\mathit{RBr} + \mathit{LBr})\}\\
 & & \quad \cup\{\mathit{Node} \cdot \mathit{Val} \cdot l \mid l \in \mathit{Wild}(\mathit{list})\}\\
\text{and } \mathit{descending} & = & \left\{ \begin{array}{l} (\mathit{Node} \cdot \mathit{LBr})^* \cdot s,\\ (\mathit{Node} \cdot \mathit{RBr})^* \cdot s,\\ (\mathit{Node} \cdot \mathit{LBr} + \mathit{Node} \cdot \mathit{RBr})^* \cdot s \end{array} \middle| \ s \in \mathit{suffixes} \right\}
\end{array}
$$

(c) Wild path set for the *tree* type

Figure 4.7: Example: type *tree* with its graph representation and wild path set

mediate types it relies on, is defined in Figure 4.7a. The type is given the visual representation of a graph in Figure 4.7b to help form intuition. Finally, the wild path set for the type is given in Figure 4.7c.

Computation To compute the wild path set of a type τ named α, we start by computing two sets of paths, R and S, based on the type definition.

The paths of R represent type recursion: they lead from a value of type α back to a value of type α without visiting intermediate values of type α. Formally,

$$\forall r \in R, \alpha.r = \alpha \wedge \forall r' \in \mathit{Prefix}(r) \setminus \{r\}, \alpha.r' \neq \alpha$$

They are computed by searching for (recursive) occurrences of α in τ.

The paths of S are the non-recursive suffixes; they are computed as follows where $\alpha_i \setminus \alpha$ is α_i where any reference to α is removed. More specifically, the type $\alpha_i \setminus \alpha$ is obtained by pruning the definition of α_i of all the leaves α.

$$S = \begin{cases} \{F_i \cdot w \mid i \leq n, w \in \mathit{Wild}(\alpha_i \setminus \alpha)\} & \text{if } \tau = \{F_1 : \alpha_1; \ldots; F_n : \alpha_n\} \\ \{D_i \cdot w \mid i \leq n, w \in \mathit{Wild}(\alpha_i \setminus \alpha)\} & \text{if } \tau = D_i\, \alpha_i + \cdots + D_n\, \alpha_n \\ \{\} & \text{if } \tau = \mathit{word} \end{cases}$$

Using these two sets R and S we define the wild path set as follows where $\Sigma\{p_1, \ldots, p_n\} = p_1 + \ldots + p_n$.

$$\mathit{Prefix}(\{(\Sigma R')^* \cdot (\Sigma S') \mid R' \subseteq R, S' \subseteq S\})$$

4.3.4 Widening

The widening operator transforms a set of type-compatible paths into a single path. It is used during ASAP analyses; specifically, it is used during the fixpoint computations to ensure termination.

The Widen operation takes a set of type-compatible paths and produces a single path. It is such that $\forall i, p_i \preceq \mathit{Widen}(\{p_1, \ldots, p_n\})$ – which implies that $\forall v, i, \eta, \sigma$, we have $Z(v, p_i)(\eta, \sigma) \subseteq Z(v, \mathit{widen}(\{p_1, \ldots, p_n\}))(\eta, \sigma)$. That is, the widening of a set of paths is a singular path that subsumes them all. Note however, that widening is not defined for arbitrary sets of paths: they must be type compatible.

Given a set of zones $\{(x, p_1), \ldots, (x, p_n)\}$, the path $\mathit{Widen}(\{p_1, \ldots, p_n\})$ is the smallest element p_0 of the wild path set $\mathit{Wild}(\Gamma(x))$ such that $\forall i, p_i \preceq p_0$. Note that widening is always used for sets of zones that pertain to a single variable (here x) which makes it possible to compute the appropriate wild path set (here $\mathit{Wild}(\Gamma(x))$).

Note that our widening operator erases more information than is strictly necessary for termination. E.g., consider the set of zones $e = \{(x, \epsilon), (x, p \cdot p), (x, p \cdot p \cdot p \cdot p), \ldots\}$ where the sub-path p always appears an even number of times; widening loses the parity information: $\mathit{Widen}(e) = \{(x, p^*)\}$.

4.3.5 Size bound

Our analysis always uses paths of bounded size; we compute the bound now. There are two cases to consider. First, in non-recursive functions, ASAP's analyses happens in one pass over the body of the function. During the analysis, both constructors and destructors introduce paths that are bigger than previously known paths. However, the size increment on the paths is bounded by a constant (see later the specific *Transfer* functions). Thus, in a non-recursive function, both the number and the length of paths in the analysis result grow linearly with the size of the function.

Second, in recursive functions, ASAP's analyses use the widening operator between each iteration of the fixpoint computation. Note that the result of a widening is a path that is a member of the wild path set for the associated type. Paths in the wild set are bounded in size. Specifically, they are at most as long as the concatenation of their longest recursive branch and their longest non-recursive branch and they are at most as wide as the number of type-compatible segments.

4.4 Use for $SCAN$ning

Paths contain enough information to generate code that scans through a given part of the heap. We write $SCAN(x, \alpha, p, PRE, FIN)$ for the scanning code that starts at the variable x of type α

and explores every memory block reachable through dereferences described by the path p. During the exploration, this scanning code uses FIN on all the memory blocks of $Z(x,p)(\eta,\sigma)$. The scanning code also uses PRE on every memory block it visits – i.e., on all the blocks of $Z(x, \mathit{Prefix}(p))(\eta,\sigma)$. The mnemonics for these compile-time functions is that FIN stands for "final" and PRE for "prefix". Note that PRE is also used on the blocks of $Z(x,p)(\eta,\sigma)$ because $p \in \mathit{Prefix}(p)$.

We use capitalised identifiers for $SCAN$, PRE and FIN because they are compiler functions, not program functions. They are like macros; they do not appear at run time but, instead, are transformed by the compiler into low-level code. The function $SCAN$ recurses at compile time over bounded-size values: types and paths. The result of that function is code (in µL) that, at execution time, recurses on potentially unbounded-size (but always finite) values. In other words, the function $SCAN$ is the algorithmic counterpart to the mathematical definition of Z in Figure 4.4. It is a *staged function* implementation of $Z(l,\alpha,p)(\eta,\sigma)$ where l, α and p are compile-time arguments and η and σ are execution-time arguments. Details about staging can be sought in Taha's introduction to multi-stage programming [36].

Note that values of the word type are treated differently – which is also the case in the definition of Z. These values are stored inline inside memory blocks or directly on the stack. Memory management is about the allocation and deallocation of blocks in the heap, not their content. Thus, the tests for types in the definition of $SCAN$ below simply skips inline word values. Remember that these tests are not performed during execution; instead they inform a compile-time decision about code generation. Notice, specifically, how the tests appear in the definition but not the example of $SCAN$.

4.4.1 Example

Figure 4.8 shows the synthesised code for the $\mathit{elems} = (\mathit{Cons} \cdot \mathit{Tail})^* \cdot \mathit{Cons} \cdot \mathit{Head}$ path of a value of type list. This scanning code explores a whole list; it calls FIN on the elements of the list and PRE on every block it traverses. The code includes hand-written comments and syntactic sugar for the purpose of readability. The delimiters $\langle$ and $\rangle$ denote (staging) code quotations whilst the construct $\sim(e)$ denotes anti-quotations – as per MetaOCaml syntax [36].

The $SCAN$ning code consists of two important functions. The first one is $\mathit{descend}$ which corresponds to the repeated portion of the path ($(\mathit{Cons} \cdot \mathit{Tail})^*$). On each cons cell of the list it calls the second important function: work. This second one resolves the Head component of the cons cell to get the corresponding element of the list. On each element of the list, it calls the function FIN.

4.4.2 Formal definition

The $SCAN$ function is formally defined in Figure 4.9. The function takes a variable (x), a type name (α), a path (p) and two compile-time functions (PRE and FIN). Its domain-codomain signature is:

$$SCAN : \begin{pmatrix} \mathit{variable} \times \mathit{typename} \\ \times \mathit{path} \\ \times(\mathit{variable} \times \mathit{typename} \to \mu\mathrm{L}) \\ \times(\mathit{variable} \times \mathit{typename} \to \mu\mathrm{L}) \end{pmatrix} \to \mu\mathrm{L}$$

At compile-time $SCAN$ mechanically produces code that, when executed, scans the value held by x of type α along the path p. Note that we require PRE to be idempotent – typically "set a mark bit to 1" or "record that address in this set." The definition uses MetaOCaml [36] syntax to

```
SCAN(x, list, ((Cons · Tail)* · Cons · Head), PRE, FIN) =
    ⟨descend(x)⟩
```

where the function $descend$ is generated as

```
fun descend(x)=
    (*this function implements (Cons · Tail)* *)
    ~(PRE(x, list));
    match x with
        [ Cons x' ->
            ~(PRE(x', list.Cons));
            match x' with {Tail=x''} ->
                ~(PRE(x'', list.Cons · Tail));
                (*this recursion implements the repetition*)
                descend(x'')
        | _ -> {} (* Nil: don't do anything *)
        ];
    (*continue with the element*)
    work(x)
```

and the function $work$ is generated as

```
fun work(x)=
    (*this function implements Cons · Head *)
    match x with
        [ Cons x' ->
            ~(PRE(x', list.Cons · Head));
            match x' with {Head=x''} ->
                ~(PRE(x'', list.Cons · Tail));
                (* accepting state: use FIN(al) action *)
                ~(FIN(x'', list.Cons · Tail))
        | _ -> {} (* Nil: don't do anything *)
        ]
```

Figure 4.8: Example of synthesised scanning code

distinguish compile-time and execution-time elements. Specifically, the notation $\langle$ and $\rangle$ is used to quote code and $\sim(e)$ to escape e.

The $SCAN$ function matches over the path and generates appropriate code. For the trivial path ϵ: use the PRE and FIN functions except for values of type $word$. For the simple field path F: use PRE on the current block, then, if the path leads to a block rather than a $word$, dereference and continue. For the simple discriminant path D: use PRE on the current block, then, if the path leads to a block rather than a $word$ and if the block matches D, dereference and continue. For the sequence path $p \cdot p'$: explore p but replace FIN by a continuation ($CONT$) that explores p'. For the alternative path $p + p'$: explore p, then explore p'. For the repetition path p^*: use a loop[2] to descend through p and call FIN on each of the blocks.

[2]Remember than in µL loops are expressed as in CPS: with a recursive function.

Note the similarity with the formal definition of Z in Figure 4.4. The main difference is the inclusion of PRE.

4.4.3 Optimisation of $SCAN$ning code

Note that Figure 4.9 above defines a simple, general definition of $SCAN$. The code generated by this simple version is not optimised. We now give an overview of different possible improvements.

First, consider the code generated for the repetition operator $SCAN(x, \alpha, p^*, PRE, FIN)$: a recursive function. Note that the recursive calls (the ones generated by $SCAN(x, \alpha.p, p, PRE, LOOP)$) are not in tail position (i.e., not immediately before `return`) which prevents tail-call optimisation – also known as tail-call elimination. It is tempting to swap the two lines of the definition of $loop$ to enable tail-call optimisation. However, note that there might be multiple sequential recursive calls – e.g., if the path p is a disjunction $p_1 + p_2$ – only one of which would be in tail position. Also note that, even with tail-call-friendly paths, swapping the two lines is not always safe. Indeed, consider the case where FIN is used to deallocate the memory: the block representing x would be deallocated before it is used.

Second, consider that we only require PRE to be idempotent; we made no further hypotheses about FIN and PRE. In Chapter 5, we see that $SCAN$ receives specific instantiations of FIN and PRE. Specifically, either FIN or PRE is a no-op. Thus, the code generated by $SCAN$ can be simplified.

Third, consider paths that are disjunction of discriminant names such as $D_1 + \cdots + D_n$. As presented above, $SCAN$ generates a sequence of n matches, at most one of which will succeed. Additionally, the generated code uses PRE n times. Instead, the function can generate a single match, at most one branch of which will be executed, as follows.

```
⟨ $\sim(PRE(x, \alpha))$;
  match $x$ with
      [ $D_1\ x'$ -> $\sim(PRE(x', \alpha.D))$; $\sim(FIN(x', \alpha.D))$
      | ...
      | $D_n\ x'$ -> $\sim(PRE(x', \alpha.D))$; $\sim(FIN(x', \alpha.D))$
      | _ -> {} /* do nothing */
      ]                                                              ⟩
```

```
$SCAN\begin{pmatrix} x, \alpha, \epsilon, \\ PRE, FIN \end{pmatrix}$ = if $\Delta(\alpha) \neq word$ then
        $\langle \sim(PRE(x,\alpha));\ \sim(FIN(x,\alpha)) \rangle$
    else
        $\langle$ {} $\rangle$
$SCAN\begin{pmatrix} x, \alpha, F, \\ PRE, FIN \end{pmatrix}$ = if $\alpha.F \neq word$ then
        $\Big\langle$
          $\sim(PRE(x,\alpha))$;
          match $x$ with
              [ {$F$=$y$} ->
                  $\sim(PRE(y,\alpha.F))$;
                  $\sim(FIN(y,\alpha.F))$
              ]
        $\Big\rangle$
    else
        $\langle \sim(PRE(x,\alpha));\ \sim(FIN(x,\alpha)) \rangle$
$SCAN\begin{pmatrix} x, \alpha, D, \\ PRE, FIN \end{pmatrix}$ = if $\alpha.D \neq word$ then
        $\Big\langle$
          $\sim(PRE(x,\alpha))$;
          match $x$ with
              [ $D\ y$ ->
                  $\sim(PRE(y,\alpha.D))$;
                  $\sim(FIN(y,\alpha.D))$
              | _ -> {} (* do nothing *)
              ]
        $\Big\rangle$
    else
        $\langle \sim(PRE(x,\alpha));\ \sim(FIN(x,\alpha)) \rangle$
$SCAN\begin{pmatrix} x, \alpha, p \cdot p', \\ PRE, FIN \end{pmatrix}$ = $\langle \sim(SCAN(x,\alpha,p,PRE,CONT)) \rangle$
        where $CONT$ is defined by
            $CONT(y,\alpha') = SCAN(y,\alpha',p',PRE,FIN)$
$SCAN\begin{pmatrix} x, \alpha, p + p', \\ PRE, FIN \end{pmatrix}$ = $\Big\langle$ $\sim(SCAN(x,\alpha,p,PRE,FIN))$;
                                      $\sim(SCAN(x,\alpha,p',PRE,FIN))$ $\Big\rangle$
$SCAN\begin{pmatrix} x, \alpha, p^*, \\ PRE, FIN \end{pmatrix}$ = $\langle loop(x) \rangle$
        where $loop$ is a fresh function defined by
            fun $loop(x)$ =
                $\Big\langle$ $\sim(SCAN(x,\alpha.p,p,PRE,LOOP))$;
                   $\sim(SCAN(x,\alpha,\epsilon,PRE,FIN))$ $\Big\rangle$
        and $LOOP$ is a staged function defined by
            $LOOP(x,\alpha) = \langle loop(x) \rangle$
```

Figure 4.9: Definition of the *SCAN* macro.

Chapter 5

Asap

We now present as-static-as-possible (asap), a novel memory-management strategy. Asap fits in the static-automatic gap highlighted in Chapter 3: it is mostly compiler-driven.

5.1 Properties

Asap is an automatic memory-management strategy in which most of the responsibility is borne by the compiler. Asap lets programmers opt-in to a limited role on a per-type basis. More specifically, programmers can decide, for each type of their program, whether or not to customise the memory representation and management of its values. Programmers that customise the memory representation of values of a given type cannot control nor observe deallocations: they merely specify a layout and provide marking-like and freeing primitives that asap uses. More details are given in Section 5.6.

Asap is safe and complete. Its waste approximation is timely relative to liveness analysis – thus its timeliness is comparable to liveness-assisted gc, more timely than classic gcs. More precisely, asap has the following characteristics:

Safe Asap never causes programs to dereference dangling pointers. Note however, that dangling pointers are created by asap, but only in places that analyses have proven safe. This is similar to liveness-assisted gc which introduces null (instead of dangling) pointers in statically provably safe places.

Complete All the values that become useless to the program are eventually deallocated.

Timely relative to *Access* analysis Waste is approximated as precisely as *Access* (which is essentially liveness for heap-allocated objects, see Section 5.4.3) is computed. Note that *Access* is more precise than unreachability; thus asap is timelier than gcs.

Compiler-driven The compiler is the main actuator for the strategy. Note however, that the programmer can be involved on an opt-in basis. The programmer, even when involved, is unaware of any individual deallocation; their role is limited to, at most, providing custom marking-like and freeing memory management primitives.

Agnostic with regards to memory representation The programmer or the compiler can customise the memory representation of any type of value. When they decide to do so, they are required to provide a few functions for handling values of the customised type as explained in Section 5.6.

Approximating and non-restrictive Waste is approximated and the language is not restricted in any way. Note that it is possible to customise the approximation of waste in different ways as detailed in Section 5.7.

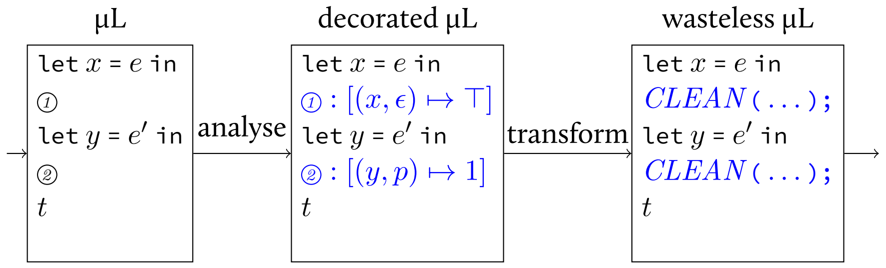


Figure 5.1: Overview of ASAP analysis and transformation

Cache-efficient ASAP avoids scanning the whole heap and tends to scan objects that have been used recently. Thus, we posit that ASAP is more cache friendly than GCs. We explore this property in Section 5.8.

As was highlighted previously, no existing approach to memory management offers this combination of benefits: agnosticism to memory representation, non-restrictiveness, and correctness.

5.2 Overview

ASAP works by a combination of static analysis and code transformation. Specifically, ASAP is a phase in the middle-end of a compiler. It receives the program in the μL IR (presented in Chapter 2), analyses it (using the framework defined in Chapter 2) and transforms it before passing it along to the next compiler phase. This is summarised in Figure 5.1.

Analyses ASAP first analyses properties of the code being compiled. More specifically, it performs data-flow analyses for three distinct properties. These properties concern value access (akin to liveness analysis but for values on the heap, similar to [24, 4]) and heap structure (akin to alias analysis but for values on the heap, similar to [24]).

Transformation Using the statically inferred information, ASAP then inserts memory-management instructions directly into the code. These instructions are generated using the $SCAN$ compile-time function defined in Chapter 4. The instructions will, during execution, deallocate the memory blocks[1] representing values that are not needed by the program. When ASAP has enough information to statically decide what to deallocate it inserts simple code that performs the necessary operations. When ASAP does not have enough information to statically decide what to deallocate, it emits instructions that, during execution, first scan part of the heap and then decide what memory can be deallocated.

Result Interestingly, the deallocation instructions are expressed in the same IR as is received from the front-end. That is, both application code and deallocation code are expressed in the same language and compiled the same way. As a result no special compilation techniques are needed in the back-end.

Also note, the transformation performed by ASAP is local: it merely adds instructions, never deleting, rearranging or tampering with code it receives. Optimisations can be performed separately by the back-end of the compiler.

[1]These deallocations do not directly affect the values stored in words on the stack or inline within record.

Additionally, µL does not possess any constructs related to low-level memory access. Thus, the deallocation code emitted by ASAP cannot rely on specifics of memory representation such as offsets and padding. As a result, ASAP is agnostic to memory representation *by construction*.

5.3 µL

We briefly revisit µL. Specifically, for each of the assumptions listed in Chapter 2, we explain how they impact ASAP.

Several assumptions are benign: distinct syntactic categories for identifiers, mandatory type annotations, and lack of nested functions. Whilst these are limits of µL, they do not constrain the source language.

One assumption is of minor importance.

Single compilation unit Programs consist of one standalone, self-contained unit; modules and name spaces are not supported. This forces ASAP to run somewhere in the compiler pipeline where the whole program is available. Alternatively, allowing multiple compilation units is possible but limits the precision of inter-procedural analysis when control-flow crosses the units borders: indeed, conservative (i.e., weak) assumptions can be made for cross-unit calls. Another alternative is to allow programmers to specify the intra-compilation-unit components of the analysis – in a fashion similar to type interfaces which specify the intra-compilation-unit for type checking.

The other assumptions are more serious. These are either difficult to compile down to µL or their compilation introduces patterns that induce a severe loss of precision in ASAP's analysis. (Note that the reduction of precision in ASAP merely causes degradation of performance at execution. Specifically, more time is spent dynamically checking for aliasing.)

No polymorphism We could compile polymorphic code to the monomorphic µL using whole program monomorphisation. However, this is not a widely deployed technique and we instead address polymorphism directly in Chapter 6.

No mutual recursion We could compile code with mutually recursive functions using a trampoline. However, trampolines impair performance as discussed in Chapter 2.

Moreover, the trampolines parameters carry information from all of the mutually recursive functions. As such, it creates a choke point in the aliasing graph: many values alias through this single variable. As a result, trampolines can limit the precision of ASAP's analyses.

Note however, that the lack of mutually recursive functions is due to the didactic nature of µL: having mutually recursive functions makes the definition and implementation of the fixpoints in ASAP's analyses more complicated. There are no fundamental limitations that prevent ASAP's analysis from being applied to mutually recursive functions.

No mutation We could compile away mutations using a state monad – the method Haskell uses [27]. However, this is potentially harmful for the precision of ASAP's analyses. Indeed, just like trampolines, this technique introduces a choke point in the aliasing graph: the state variable. We address the lack of mutability in Chapter 6 by enhancing µL and adapting ASAP.

No higher-order We could use defunctionalisation [14] to transform higher-level programs to only use first-class function. However, such techniques are not widely deployed. We posit

Analysis	decorations
$Shape$	$\odot \rightarrow 3^{zone \times zone}$
$Share$	$\odot \rightarrow 3^{zone}$
$Access$	$\odot \rightarrow 3^{zone}$

Figure 5.2: Decorations collected by asap's analyses

it is possible to integrate control-flow analysis in asap to handle higher-order functions. However, doing so is left as future work.

5.4 Analyses

Asap performs three data-flow analyses:

Shape A generalisation of alias analysis, $Shape$ analysis is concerned with the way in which the representation of two values might share some memory blocks. The $Shape$ analysis can determine, say, if two lists definitely share some elements, or whether they might have a common suffix (i.e., share some part of their spines). This information is gathered in the form of decorations which are 3vl relations over $zone$ where $Shape(z, z')$ is the 3vl-certainty that the zones z and z' overlap – a formal definition is given in Section 5.4.2.

Share This analysis is concerned with internal sharing: whether there is sharing within a given data-structure. The $Share$ analysis detects, say, if multiple elements of a list might be represented with the same memory block. The $Share$ decorations are 3vl sets of $zone$ where $Share(z)$ is the 3vl-certainty that the zone z contains internal sharing – a formal definition is given in Section 5.4.2. Note that $Shape$ and $Share$ properties are interdependent and analysed simultaneously.

Access A generalisation of liveness, $Access$ analysis is concerned with future uses of values. More specifically, it detects what parts of which values are going to be used later in the program. The $Access$ analysis detects for, say, a list, whether its elements or its spine or both are definitely accessed later in the program. The $Access$ decorations are 3vl sets of $zone$ where $Access(z)$ is the 3vl-certainty that some blocks of the zone z are accessed later in the program – a formal definition is given in Section 5.4.3.

The result of these analyses are used to generate deallocation instructions as detailed in Section 5.5.

5.4.1 Data-flow analysis framework

We use the data-flow analysis framework presented in Chapter 2. That is, we generate equations that relate information at one program point to another, using the schema of Figure 2.10a or Figure 2.11 (depending on the direction of the analysis). We define the $Transfer$ functions for the $Shape$ and $Share$ analyses in Section 5.4.2 and for the $Access$ analysis in Section 5.4.3.

Nature of the decorations All of asap's analyses collect information in the form of 3vl sets and relations. Figure 5.2 presents the domain and range of the decorations.

As is standard in 3vl analyses, information from different program branches is merged using $\sqcup$.

Function summaries The analyses of ASAP use summaries (and call contexts, see below) for context-insensitive inter-procedural aspects of the analysis. The summaries are used to handle function calls as detailed in the definitions of the *Transfer* functions below. Summaries are computed, as explained in Chapter 2.

Note that to ensure the fixpointing of summaries always terminates, we use the widening function for paths defined in Chapter 4. Remember that the fixpointed summaries are initialised to the empty map [ ] which is neutral for all the 3VL set and relation operators.

For example, consider the function $\mathit{nth}(l\colon \mathit{list},\ n\colon \mathit{word})\colon \mathit{pair}$ which takes a list of pairs (l) and a number (n) and returns the n-th pair of the list l. Its summary for the *Access* analysis includes $[(l, \mathit{elems}) \mapsto 1]$ which indicate that the function accesses the elements of the list l. On the other hand, the function $\mathit{length}(l\colon \mathit{list})\colon \mathit{word}$ which computes the length of a list (l) has the *Access* summary $[(l, \mathit{spine}) \mapsto 1, (l, \mathit{elems}) \mapsto 0]$ which indicates that only the spine of the list is accessed during a call.

Call contexts We compute the amalgamated call contexts (as presented in Chapter 2) for all three analyses of ASAP. The call context for the *Shape* and *Share* properties informs the callee about the 3VL-certainty of sharing within its parameters. The call context for the *Access* property informs the callee about the 3VL-certainty of whether the parameters and return value are used after it returns.

For example, consider the function $\mathit{cons}(p\colon \mathit{pair}, l\colon \mathit{list})\colon \mathit{list}$ which takes a pair (p) and a list of pairs (l) and returns a new list where the head is p and the tail is l. The amalgamated for the *Shape* analysis indicates whether the arguments p and l ever alias at call points. Specifically, the value $\mathit{cons}^{\uparrow}((p, \epsilon), (l, \mathit{elems}))$ is 1 if, at every call, the memory block representing the pair p is already stored in the list l. On the other hand, the value is 0 if p never appears in l at any call points.

5.4.2 *Shape* and *Share*

ASAP performs a forward data-flow analysis to collect a static description of the heap. It analyses the program to determine two distinct but related properties: *Shape* and *Share*.

Shape analysis (similar to Sagiv et al. [35]) characterises the way memory is shared between distinct values on the heap. It can be thought of as a generalisation of alias analysis concerned with heap structure rather than stack variables.

Consider, for example, the term below. The *Shape* decoration indicate that, at program point $\textcircled{\pi}$, the tails of xs and ys are always represented by the same memory block. Specifically, $\mathit{Shape}(\textcircled{\pi})((xs, \mathit{Cons} \cdot \mathit{Tail}), (ys, \mathit{Cons} \cdot \mathit{Tail})) = 1$.

```
let xs = Cons {Head = ... ; Tail =zs} in
let ys = Cons {Head = ... ; Tail =zs} in
ⓟt
```

On the other hand, the *Share* analysis is concerned with sharing within a zone: whether some blocks of a zone can be accessed through two distinct paths. For example, if two elements of a list l are represented with the same block, there is sharing within the zone (l, elems).

Formal definition of Shape

The property $\mathit{Shape}(\textcircled{\pi})(z, z')$ statically approximates whether the set of memory blocks described by the zones z and z' intersect at program point $\textcircled{\pi}$ in none (0), some ($\top$) or all (1) of the executions. The formal definition is given in Figure 5.3 where *State*, formally defined in Chapter 2, associates

$$Shape(\textcircled{\pi})(z, z') = \begin{cases} 1 & \text{if } \forall(\eta, \sigma) \in State(\textcircled{\pi}), Z(z)(\eta, \sigma) \cap Z(z')(\eta, \sigma) \neq \emptyset \\ 0 & \text{if } \forall(\eta, \sigma) \in State(\textcircled{\pi}), Z(z)(\eta, \sigma) \cap Z(z')(\eta, \sigma) = \emptyset \\ \top & \text{otherwise} \end{cases}$$

Figure 5.3: Formal definition of *Shape* using zone intersection

to each program point $\textcircled{\pi}$ the set of states the program might visit at $\textcircled{\pi}$. Remember that, for our level of abstraction, states are specified as triplets $(\textcircled{\pi}, \eta, \sigma)$ where $\textcircled{\pi}$ is a program point, η is a heap, and σ is a stack.

Note that *Shape* is not decidable because *State* is not either. However, weaker approximations are decidable and useful. Specifically, we accept approximations of *Shape* where $\top$ is used where 0 or 1 holds.

Computation of *Shape*

An approximation of *Shape* is computed using the data flow analysis framework of Chapter 2 and the *Transfer* functions from Figure 5.4.

Note that, by nature, *Shape* is a symmetric and reflexive relation. Additionally, *Shape* is suffix closed: if two lists share a cons cell, they share all the subsequent cons cells as well. Furthermore, when a zone (x, p_x) aliases with (w, p'_w) and a zone (y, p_y) aliases with (w, p''_w), then it is possible (x, p_x) and (y, p_y) alias provided there is sharing within the representation of the value w. In Figure 5.4a, the auxiliary function *Close* integrates these aspect of the *Shape* property into the analyses. To understand the role of *shr* consider the following example: if x and y are each an element of a list w, then they might alias if w has internal sharing. Specifically, the paths might be $p_x = Cons \cdot Head$ (i.e., x is the first element of the list), $p_y = Cons \cdot Tail \cdot Cons \cdot Head$ (i.e., y is the second element of the list), and $p_w = (Cons \cdot Tail)^* \cdot Cons \cdot Head$ (i.e., (w, p_w) is the set of elements of the list).

Consider the computation of the *Shape* property for the entry-point, main function ($main(x_1, \ldots, x_n)$): no call context is available. Thus, the initial value is:

$$Close\left(\left[\left(\begin{array}{c}(x_i,\ Wild(\Gamma(x_i))), \\ (x_j,\ Wild(\Gamma(x_j)))\end{array}\right) \mapsto 0 \;\middle|\; i, j \leq n\right]\right)$$

In other words, asap assumes that, when the program starts, there is no aliasing between the parameters of the main function.

Formal definition of *Share*

The value of $Share(\textcircled{\pi})(x, p)$ is the certainty that p recognises two (or more) distinct sequences of dereference that, starting from the address of x, lead to the same memory cell at program point $\textcircled{\pi}$ for any execution. That is, $Share(\textcircled{\pi})(z)$ indicates that at program point $\textcircled{\pi}$, $Z(z)$ has internal sharing in none (0), some ($\top$) or all (1) of the executions.

As with *Shape*, this property is not decidable. As with *Shape* we accept approximations that are weaker (i.e., where $\top$ can appear in place of either 0 or 1).

$$
\begin{aligned}
&\mathit{Close} : 3^{\mathit{zone}\times\mathit{zone}} \to 3^{\mathit{zone}\times\mathit{zone}}\\
&\mathit{Close}(m) = \mathit{Suffix}(\mathit{Refl}(\mathit{Sym}(m \vee \mathit{shr}(m))))\\
&\mathit{shr} : 3^{\mathit{zone}\times\mathit{zone}} \to 3^{\mathit{zone}\times\mathit{zone}}\\
&\mathit{shr}(m) = \left[\begin{pmatrix}(x,p_x),\\(y,p_y)\end{pmatrix} \mapsto \left(\bigvee_{\substack{w\in \mathit{variable}\\ p_w\in \mathit{Wild}(\Gamma(w))\\ p'_w,p''_w\preceq p_w}} \begin{pmatrix} & \mathit{Share}(\text{Ⓟ})((w,p_w))\\ \wedge & m((x,p_x),(w,p'_w))\\ \wedge & m((y,p_y),(w,p''_w)) \end{pmatrix} \right) \;\middle|\; \begin{pmatrix}(x,p_x),\\(y,p_y)\end{pmatrix} \in \mathit{domain}(m) \right]
\end{aligned}
$$

(a) Auxiliary functions for the *Shape* analysis

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\mathit{Transfer}_{\mathrm{e}} : \mathit{variable} \times \mathit{expr} \to 3^{\mathit{zone}\times\mathit{zone}} \to 3^{\mathit{zone}\times\mathit{zone}}}\\
\mathit{Transfer}_{\mathrm{e}}(x,l)(m) & = & m \qquad \text{where } l \text{ is a literal}\\
\mathit{Transfer}_{\mathrm{e}}(x,y)(m) & = & \mathit{Close}(m \lhd [((x,\epsilon),(y,\epsilon)) \mapsto 1])\\
\mathit{Transfer}_{\mathrm{e}}(x,D\ y)(m) & = & \mathit{Close}(m \lhd [((x,D),(y,\epsilon)) \mapsto 1])\\
\mathit{Transfer}_{\mathrm{e}}(x,\{F_1\text{=}y_1;\ldots;F_n\text{=}y_n\})(m) & = & \mathit{Close}(m \lhd [((x,F_i),(y_i,\epsilon)) \mapsto 1 \mid i \leq n])\\
\mathit{Transfer}_{\mathrm{e}}(x,\mathit{op}(y_1,\ldots,y_n))(m) & = & m\\
\mathit{Transfer}_{\mathrm{e}}(x,f(y_1,\ldots,y_n))(m) & = & \mathit{Close}(m \lhd \theta^{\vec{y}}_{\vec{z}}(\theta^{x}_{\mathit{ret}}(f^{\downarrow})))
\end{array}
$$

(b) $\mathit{Transfer}_{\mathrm{e}}$ function for the *Shape* analysis

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\mathit{Transfer}_{\mathrm{p}} : \mathit{variable} \times \mathit{pattern} \to 3^{\mathit{zone}\times\mathit{zone}} \to 3^{\mathit{zone}\times\mathit{zone}}}\\
\mathit{Transfer}_{\mathrm{p}}(x,l)(m) & = & m \qquad \text{where } l \text{ is a literal}\\
\mathit{Transfer}_{\mathrm{p}}(x,y)(m) & = & \mathit{Close}(m \lhd [((y,\epsilon),(x,\epsilon) \mapsto 1]))\\
\mathit{Transfer}_{\mathrm{p}}(x,D\ y)(m) & = & \mathit{Close}(m \lhd [((y,\epsilon),(x,D)) \mapsto 1])\\
\mathit{Transfer}_{\mathrm{p}}(x,\{F_1\text{=}y_1;\ldots;F_n\text{=}y_n\})(m) & = & \mathit{Close}(m \lhd [((y_i,\epsilon),(x,F_i)) \mapsto 1 \mid i \leq n])
\end{array}
$$

(c) $\mathit{Transfer}_{\mathrm{p}}$ function for the *Shape* analysis

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\mathit{Transfer}_{\mathrm{r}} : \mathit{variable} \to 3^{\mathit{zone}\times\mathit{zone}} \to 3^{\mathit{zone}\times\mathit{zone}}}\\
\mathit{Transfer}_{\mathrm{r}}(x)(m) & = & \mathit{Close}(m \lhd [((\mathit{ret},\epsilon),(x,\epsilon)) \mapsto 1])
\end{array}
$$

(d) $\mathit{Transfer}_{\mathrm{r}}$ function for the *Shape* analysis

Figure 5.4: The *Transfer* functions for *Shape* analysis

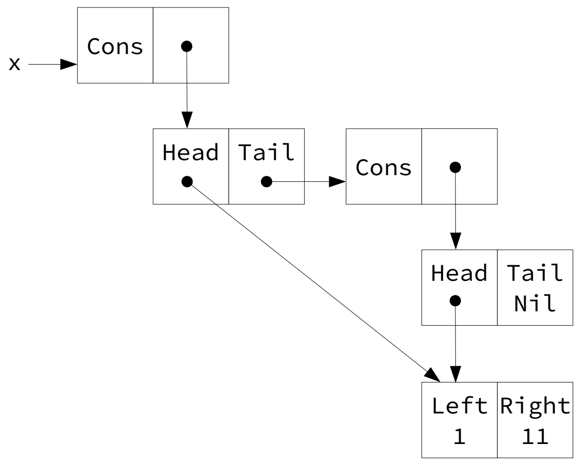


Figure 5.5: Simple data-structure with sharing

Example of Sharing

We illustrate sharing in Figure 5.5: the zone $(x, (\mathit{Cons} \cdot \mathit{Tail})^* \cdot \mathit{Cons} \cdot \mathit{Head})$ has sharing. Note that, zones with sharing are suffix closed; that is, if a zone (x, p) has sharing, then $\forall p'$, the zone $(x, p \cdot p')$ has sharing – provided p' is appropriate for the type $\Gamma(x).p$.

Computation of *Share*

Asap computes an approximation of *Share* with the *Transfer* functions defined in Figure 5.6. Note that, in order to analyse sharing, the value of *Shape* and *Share* at the previous program point (noted ⓟ here) is necessary – these appear as the terms *Shape*(ⓟ) and *Share*(ⓟ) respectively in the *Transfer* functions. Additionally, the *Share* property is suffix closed which is reflected in the local *Close* function.

We now discuss the $\mathit{Transfer}_{\mathrm{e}}$ function – defined in Figure 5.6b. Specifically, we discuss the *carry* and *fresh* components in the record-binding case:

$$\mathit{Transfer}_{\mathrm{e}}(x, \{F_1 = y_1 ; \ \ldots ; F_n = y_n\})(m) = \mathit{Close}(m \lhd (\mathit{carry} \vee \mathit{fresh}))$$

The transfer adds, to the known map m, information from two maps (*carry* and *fresh*) that we separated for clarity. The *carry* map carries information about sharing inside any of the y_i. More specifically, it carries sharing information from any zone of the form (y_i, p) to the corresponding $(x, F_i \cdot p)$. The *fresh* map integrates information about aliasing that is created when two of the fields receive values that alias. Consider the special case when two of the variables, say y_i and y_j are the same[2]. In this case, the zone $(x, (F_i + F_j))$ is given sharing information equal to *Shape*(ⓟ)$((y_i, \epsilon), (y_j, \epsilon))$. This *Shape* value is 1 because *Shape* is reflexively closed and $y_i = y_j$.

Note that the *fresh* map contains paths of the form $(x, (F_i \cdot p + F_j \cdot p'))$ which may not appear in the wild path set of x. In this case, it is safe to discard the zone altogether during fixpoints. Indeed, the property *Shape* keeps track of these forms of aliasing in these cases: the value of *Shape*(ⓟ)$((x, F_i \cdot p), (x, F_j \cdot p'))$ carries the information that can be discarded from *Share*(ⓟ).

[2]Remember that y_i and y_j are meta-variables that stand for variables in μL. Thus, it is possible y_i and y_j stand for the same variable, say y.

$$\begin{aligned}
\mathit{Close} &: 3^{zone \times zone} \to 3^{zone \times zone} \\
\mathit{Close}(m) &= \mathit{Suffix}(m)
\end{aligned}$$

(a) Auxiliary function for the *Share* analysis

$$\mathit{Transfer}_{\mathrm{e}} : \mathit{variable} \times \mathit{expr} \to 3^{zone} \to 3^{zone}$$

$$\begin{aligned}
\mathit{Transfer}_{\mathrm{e}}(x, l)(m) &= m \qquad \text{where } l \text{ is a literal} \\
\mathit{Transfer}_{\mathrm{e}}(x, y)(m) &= \mathit{Close}(m \lhd [(x,p) \mapsto \mathit{Share}(\textcircled{\pi})(y,p)]) \\
\mathit{Transfer}_{\mathrm{e}}(x, D\ y)(m) &= \mathit{Close}(m \lhd [(x, D \cdot p), (x,p) \mapsto \mathit{Share}(\textcircled{\pi})(y,p)]) \\
\mathit{Transfer}_{\mathrm{e}}(x, \{F_1{=}y_1;\ \ldots;F_n{=}y_n\})(m) &= \mathit{Close}(m \lhd (\mathit{carry} \vee \mathit{fresh})) \\
\text{where } \mathit{carry} &= \left[(x, F_i \cdot p) \mapsto \mathit{Share}(\textcircled{\pi})(y_i, p) \;\middle|\; \begin{array}{c} i \le n \\ p.\Gamma(y_i) \text{ defined} \end{array} \right] \\
\text{and } \mathit{fresh} &= \left[\begin{array}{l} (x, (F_i \cdot p + F_j \cdot p')) \\ \quad \mapsto \mathit{Shape}(\textcircled{\pi})((y_i, p), (y_j, p')) \end{array} \;\middle|\; \begin{array}{c} i, j \le n \\ p.\Gamma(y_i) \text{ defined} \\ p'.\Gamma(y_j) \text{ defined} \end{array} \right] \\
\mathit{Transfer}_{\mathrm{e}}(x, op(y_1, \ldots, y_n))(m) &= m \\
\mathit{Transfer}_{\mathrm{e}}(x, f(y_1, \ldots, y_n))(m) &= \mathit{Close}(m \lhd \theta^{\vec{y}}_{\vec{z}}(\theta^{x}_{ret}(f^{\downarrow})))
\end{aligned}$$

(b) $\mathit{Transfer}_{\mathrm{e}}$ function for the *Share* analysis

$$\mathit{Transfer}_{\mathrm{p}} : \mathit{variable} \times \mathit{pattern} \to 3^{zone} \to 3^{zone}$$

$$\begin{aligned}
\mathit{Transfer}_{\mathrm{p}}(x, l)(m) &= m \qquad \text{where } l \text{ is a literal} \\
\mathit{Transfer}_{\mathrm{p}}(x, y)(m) &= \mathit{Close}(m \lhd [(y,p) \mapsto \mathit{Share}(\textcircled{\pi})(x,p)]) \\
\mathit{Transfer}_{\mathrm{p}}(x, D\ y)(m) &= \mathit{Close}(m \lhd [(y,p) \mapsto \mathit{Share}(\textcircled{\pi})(x, D \cdot p)]) \\
\mathit{Transfer}_{\mathrm{p}}(x, \{F_1{=}y_1;\ \ldots;F_n{=}y_n\})(m) &= \mathit{Close}(m \lhd [(y_i,p) \mapsto \mathit{Share}(\textcircled{\pi})(x, F_i \cdot p) \mid i \le n])
\end{aligned}$$

(c) $\mathit{Transfer}_{\mathrm{p}}$ function for the *Share* analysis

$$\begin{aligned}
\mathit{Transfer}_{\mathrm{r}} &: \mathit{variable} \to 3^{zone} \to 3^{zone} \\
\mathit{Transfer}_{\mathrm{r}}(x)(m) &= \mathit{Close}(m \lhd [(ret,p) \mapsto \mathit{Share}(x,p)])
\end{aligned}$$

(d) $\mathit{Transfer}_{\mathrm{r}}$ function for the *Share* analysis

Figure 5.6: The *Transfer* functions for *Share* analysis

This discarding is necessary to fixpoint summaries and call contexts because the operation is based on the wild path set.

Finally, note that none of the constructs of µL invalidate *Shape* nor *Share* properties. As a result, the *Transfer* functions never explicitly introduce 0 in the decorations – they might introduce 0 implicitly through the auxiliary *Close* functions. Additionally, the scope in µL extends to the end of the term (because of the ANF-like absence of nesting). As a result, there is no need for domain restrictions in the update functions of the (forwards) *Share* and *Shape* analyses.

5.4.3 *Access*

Access analysis characterises parts of the heap that are still necessary to the computation. It can be thought of as a generalisation of liveness concerned with heap structures rather than stack variables. Unlike liveness, however, access is not concerned with the distinction between read and write: all uses matter when trying to avoid use-after-free. Access expresses facts such as: the spine of a list is still definitely useful whilst its elements are definitely not – e.g., if the list is only ever passed to `length`-like functions. This property has been used to improve GCs [4, 24].

Consider the term below in which x is an integer and l is a list. At point ③, the $Access$ property for the zone $(l, spine)$ is 0: the list is not accessed at all. At point ②, the $Access$ property for the zone $(l, spine)$ is 1: the spine of the list is accessed when its length is computed. At point ①, the $Access$ property for the zone $(l, spine)$ is $\top$: it is accessed in some but not all of the executions.

```
①if x>0 then
    ②let y = length(l) in
    return y
else
    ③return x
```

Formal definition of $Access$

The value $Access(\text{Ⓟ})(z)$ specifies whether some memory blocks of the zone z are accessed in every (1), some ($\top$), or none (0) of the possible states reachable from Ⓟ.

As with $Shape$ and $Share$ analyses, it is not possible to decide $Access$. However, we are satisfied with an approximation where $\top$ replaces some 0 and 1 entries.

Computation of the $Access$ **property**

The $Access$ analysis generates equations using the $Transfer$ functions defined in Figure 5.7.

It too uses an auxiliary function $Close$ (defined in Figure 5.7a). This auxiliary function keeps track of *implied* accesses: accesses due to aliasing with an explicitly accessed value. Implied accesses can be approximated using the aliasing information available in the $Shape$ decorations.

Note that, because Access is a backwards analysis, we need to be careful with scope. Specifically, we need to remove all references to a variable when reaching the point before its binding. This is achieved in the $Transfer$ function through the use of the $\setminus$ operator.

Also note the treatment of the `return` construct. The amalgamated call context is used for the map m; it contains information about the use of parameter and return value by the caller after the callee returns. Apart from the variables explicitly mentioned in m, all the rest of the variables in scope are not used after the return. In order to mirror that fact, the $Transfer$ function sets all zones to 0 and uses the update operator ($\lhd$) to overwrite this default map with information from the amalgamated call context. Also note that the $Shape$ property keeps track of the aliasing between ret and x. As a result, the $Access$ information from the amalgamated call is transferred onto x by the $Close$ function.

5.5 Transformation

After running the analyses defined above, ASAP enters its next phase: transforming the program. Specifically, ASAP uses the $Shape$, $Share$ and $Access$ decorations to generate and insert memory-management code at appropriate program points. This memory management code scans through parts of the heap, decides which blocks should be deallocated and deallocates them. When the statically inferred information is sufficiently precise, ASAP does away with the scanning part and emits code that deallocates values unconditionally.

$$\begin{array}{rcl}
\mathit{Close} & : & 3^{zone} \to 3^{zone} \\
\mathit{Close}(m) & = & m \vee \mathit{imply}(m) \\
\mathit{imply} & : & 3^{zone} \to 3^{zone} \\
\mathit{imply}(m) & = & [z \mapsto (\bigvee_{z'}(m(z') \wedge \mathit{Shape}(\textcircled{\pi})(z', z)))]
\end{array}$$

(a) Auxiliary function for the *Access* analysis

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathit{Transfer}_{\mathrm{e}} : \mathit{variable} \times \mathit{expr} \to 3^{zone} \to 3^{zone}} \\
\mathit{Transfer}_{\mathrm{e}}(x, l)(m) & = & m \setminus \{x\} \qquad \text{where } l \text{ is a literal} \\
\mathit{Transfer}_{\mathrm{e}}(x, y)(m) & = & \mathit{Close}(m \lhd [(y, \epsilon) \mapsto 1]) \setminus \{x\} \\
\mathit{Transfer}_{\mathrm{e}}(x, \mathit{Tag}\ y)(m) & = & \mathit{Close}(m \lhd [(y, \epsilon) \mapsto 1]) \setminus \{x\} \\
\mathit{Transfer}_{\mathrm{e}}(x, \{F_1\texttt{=}y_1\texttt{;} \ldots \texttt{;}F_n\texttt{=}y_n\})(m) & = & \mathit{Close}(m \lhd [(y_i, \epsilon) \mapsto 1 \mid i \leq n]) \setminus \{x\} \\
\mathit{Transfer}_{\mathrm{e}}(x, op\texttt{(}y_1\texttt{,} \ldots \texttt{,}y_n\texttt{)})(m) & = & \mathit{Close}(m \lhd [(y_i, \epsilon) \mapsto 1 \mid i \leq n]) \setminus \{x\} \\
\mathit{Transfer}_{\mathrm{e}}(x, f\texttt{(}y_1\texttt{,} \ldots \texttt{,}y_n\texttt{)})(m) & = & \mathit{Close}(m \lhd \theta^{\vec{y}}_{\vec{z}}(\theta^{x}_{ret}(f^{\downarrow}))) \setminus \{x\}
\end{array}$$

(b) $\mathit{Transfer}_{\mathrm{e}}$ for the *Access* analysis

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathit{Transfer}_{\mathrm{p}} : \mathit{variable} \times \mathit{pattern} \to 3^{zone} \to 3^{zone}} \\
\mathit{Transfer}_{\mathrm{p}}(x, l)(m) & = & m \qquad \text{where } l \text{ is a literal)} \\
\mathit{Transfer}_{\mathrm{p}}(x, y)(m) & = & \mathit{Close}(m \lhd [(x, \epsilon) \mapsto 1]) \setminus \{y\} \\
\mathit{Transfer}_{\mathrm{p}}(x, D\ y)(m) & = & \mathit{Close}(m \lhd [(x, D) \mapsto 1]) \setminus \{y\} \\
\mathit{Transfer}_{\mathrm{p}}(x, \{F_1\texttt{=}y_1\texttt{;} \ldots \texttt{;}F_n\texttt{=}y_n\})(m) & = & \mathit{Close}(m \lhd [(x, F_i) \mapsto 1 \mid i \leq n]) \setminus \{y_i \mid i \leq n\}
\end{array}$$

(c) $\mathit{Transfer}_{\mathrm{p}}$ for the *Access* analysis

$$\begin{array}{rcl}
\multicolumn{3}{c}{\mathit{Transfer}_{\mathrm{r}} : \mathit{variable} \to 3^{zone} \to 3^{zone}} \\
\mathit{Transfer}_{\mathrm{r}}(x)(m) & = & \mathit{Close}([(y,\ \mathit{Wild}(\Gamma(y))) \mapsto 0 \mid y \text{ in scope at } \textcircled{\pi}] \lhd m)
\end{array}$$

(d) $\mathit{Transfer}_{\mathrm{r}}$ for the *Access* analysis

Figure 5.7: The *Transfer* functions for Access analysis.

We present this transformation in steps. First, we give the general approach where ASAP emits inefficient code that scans through all the variables in scope at every program point. Second, we show how the part of the heap that is scanned can be reduced. Third, we detail how the actual scanning code is generated.

5.5.1 µL functions and compile-time functions

In the discussion below, we use the same conventions as in Chapter 4. Specifically: all upper-case variables (e.g., $CLEAN$) are reserved for compile-time functions, $\langle$ and $\rangle$ are used to quote code and $\sim(e)$ denotes anti-quotation.

The distinction between compile-time and execution-time values is important. It shows that the bulk of the work of ASAP is carried by the compiler. It also draws parallels between the workings of GCs and ASAP which are further explored in Chapter 8.

5.5.2 Pseudo-primitive CLEAN

We first assume there exists a compile-time function called CLEAN which expands into deallocation code. We use it as a primitive in an intermediate step in our transformation. It is compiled out in Section 5.5.4.

Purpose of CLEAN

Asap introduces code in the form $\sim(\mathit{CLEAN}(m_1, \ldots, m_p)(a_1, \ldots, a_q))$ where all m_i and a_i are zones. The m_i arguments are zones that describe the memory which needs to be kept and we call *matter set*[3] the set of m_i. The a_i arguments are zones that describe the memory which needs to be deallocated and we call *anti-matter set* the set of a_i. That is, $\sim(\mathit{CLEAN}(m_1, \ldots, m_q)(a_1, \ldots, a_p))$ deallocates the set of blocks $\bigcup_i Z(a_i) \setminus \bigcup_j Z(m_j)$.

The means by which CLEAN performs these deallocations are described in Section 5.5.4.

Usage of CLEAN

Calls to CLEAN are inserted into the terms of µL by the $[\![.]\!]_{\mathrm{CLEAN}}$ function defined below. The placement and arguments of these calls depend on the term and its decorations.

Bindings Terms with bindings are transformed as follows:

$$\left[\!\!\left[\begin{array}{l} ^{\textcircled{1}}\texttt{let } x \texttt{ = } e \texttt{ in} \\ ^{\textcircled{2}}t \end{array} \right]\!\!\right]_{\mathrm{CLEAN}} = \begin{array}{l} ^{\textcircled{1}}\texttt{let } x \texttt{ = } e \texttt{ in} \\ \sim(\mathit{CLEAN}(M)(A))\texttt{;}\ [\![^{\textcircled{2}}t]\!]_{\mathrm{CLEAN}} \end{array}$$

$$\begin{array}{l} A = \{z \mid \mathit{Access}(\textcircled{1})(z) \geq \top, \mathit{Access}(\textcircled{2})(z) = 0\} \\ M = \{z \mid \mathit{Access}(\textcircled{2})(z) \geq \top\} \end{array}$$

The anti-matter set, A, contains those zones that may be accessed during the evaluation of e but are definitely not accessed after that. The matter set, M, contains those zones that may be accessed after evaluating e.

Function calls need to be treated separately. Indeed, some of the memory passed as argument in a call may be deallocated by the callee. However, the part that is deallocated depends on the *amalgamated* call context which is in general weaker (i.e., greater for the uncertainty order $\sqsubseteq$) than the context for the single call considered. The difference between the two contexts must be deallocated after the call returns. To that end, bindings with function calls are transformed thus:

$$\left[\!\!\left[\begin{array}{l} ^{\textcircled{1}}\texttt{let } x \texttt{ = } f\texttt{(...)}\texttt{ in} \\ ^{\textcircled{2}}t \end{array} \right]\!\!\right]_{\mathrm{CLEAN}} = \begin{array}{l} ^{\textcircled{1}}\texttt{let } x \texttt{ = } f\texttt{(...)}\texttt{ in} \\ \sim(\mathit{CLEAN}(M)(A))\texttt{;}\ [\![^{\textcircled{2}}t]\!]_{\mathrm{CLEAN}} \end{array}$$

$$\begin{array}{l} A = \{z \mid \mathit{Access}(\textcircled{1})(z) \geq \top, \mathit{Access}(\textcircled{2})(z) = 0, \theta_{\vec{p}}^{\vec{a}}(f^{\uparrow})(z) \neq 0\} \\ M = \{z \mid \mathit{Access}(\textcircled{2})(z) \geq \top\} \end{array}$$

The difference with the general case for expressions is the additional guard in the definition of the anti-matter set A: $\theta_{\vec{p}}^{\vec{a}}(f^{\uparrow})(z) \neq 0$ (where $\vec{p}$ are the formal parameters and $\vec{a}$ are the actual arguments) which guarantees the zones of the anti-matter set have not been deallocated by the callee.

[3]The terms *matter* and *anti-matter* are borrowed from the work of Röjemo and Runciman [34].

$$\begin{array}{rcll}
\textit{matches}: & & & \\
\textit{variable} \times \textit{pattern} \times \textit{zone} & \to & \{0,1\} & \\
(x, y, z) & \mapsto & 1 & \text{(keep non-discriminant patterns)} \\
(x, l, z) & \mapsto & 1 & \text{(keep non-discriminant patterns)} \\
(x, \{\dots\}, (x, p) & \mapsto & 1 & \text{(keep non-discriminant patterns)} \\
(x, D\,y, (w, p)) \text{ where } w \neq x & \mapsto & 1 & \text{(keep zones of variables other than } x) \\
(x, D\,y, (x, (\dots + (D \cdot p) + \dots) \cdot p')) & \mapsto & 1 & \text{(keep compatible zones)} \\
(x, D\,y, (x, (\dots + (D \cdot p) + \dots)^* \cdot p')) & \mapsto & 1 & \text{(keep compatible zones)} \\
(x, p, z) \text{ otherwise} & \mapsto & 0 & \text{(do not keep incompatible zones)}
\end{array}$$

Figure 5.8: The *matches* helper function

Destructors A similar transformation affects terms which perform matching:

$$\left[\!\!\left[\begin{array}{l} ^{①}\texttt{match } x \texttt{ with} \\ \quad \texttt{[...} \\ \quad \texttt{| } p \texttt{ ->} {}^{②}t \\ \quad \texttt{| ...} \\ \quad \texttt{]} \end{array} \right]\!\!\right]_{\text{CLEAN}} = \begin{array}{l} ^{①}\texttt{match } x \texttt{ with} \\ \quad \texttt{[...} \\ \quad \texttt{| } p \texttt{ ->} \sim(\mathit{CLEAN}(M)(A)); \\ \qquad\qquad {}^{②}[\![t]\!]_{\text{CLEAN}} \\ \quad \texttt{| ...} \\ \quad \texttt{]} \end{array}$$

$$\begin{aligned} A &= \{z \mid \mathit{Access}(①)(z) \geq \top, \mathit{Access}(②)(z) = 0\} \\ M &= \{z \mid \mathit{Access}(②)(z) \geq \top, \mathit{matches}(x, p, z)\} \end{aligned}$$

Other branches (not represented here) must also be transformed in the same way. The matter and anti-matter sets are essentially the same as with binders; the only difference is the use of $\mathit{matches}(x, p, z)$ – formally defined in Figure 5.8 – which avoids zones that are inconsistent with the pattern p. Specifically, *matches* is false for triplets of the form $(x, D_p, (x, D_z))$ where the discriminant matched by the pattern (D_p) is different from the discriminant expected by the zone (D_z). In other words, *matches* integrates flow-sensitive information about the possible values that the pattern allows.

Return Terms of the form `return` x are not transformed:

$$[\![\texttt{return } x]\!]_{\text{CLEAN}} = \texttt{return } x$$

Indeed, consider the term ${}^{⓪}\texttt{return } x^{①}$. Specifically, consider the value of the *Access* property at program point ⓪: all zones map to 0 except for those mentioned in the amalgamated call context. Thus, all the zones except for the ones mentioned in the amalgamated call context have already been placed in an anti-matter set at a previous program point.

In other words, at return points, all the function's memory has already been managed.

5.5.3 Optimising matter and anti-matter sets

The matter and anti-matter sets, as presented above, grow with the number of variables in scope. Fortunately, both sets can be trimmed down using the *Shape* property. This is achieved by the $[\![\cdot]\!]_{\text{trim}}$ function defined below.

Specifically, we transform terms as follows:

$$\begin{aligned}&[\![\sim(\mathit{CLEAN}(M)(A))\,;\,{}^{①}t]\!]_{\text{trim}}\\&\qquad=\sim(\mathit{CLEAN}(M_1)(A_1))\,;\;\ldots\,;\sim(\mathit{CLEAN}(M_n)(A_n))\,;\,{}^{①}[\![t]\!]_{\text{trim}}\end{aligned}$$

where the A_i and M_i are such that:

$$\bigcup_i\left(\bigcup_{a\in A_i}Z(a)\setminus\bigcup_{m\in M_i}Z(m)\right)=\bigcup_{a\in A}Z(a)\setminus\bigcup_{m\in M}Z(m)$$

This equality ensures that the set of deallocated memory cells is identical before and after the transformation. The transformation happens in three consecutive steps which we describe now.

First, we remove from A those zones that also appear in M:

$$A'=\{(v,p_a)\in A\mid\neg\exists(v,p_m)\in M,p_a\preceq p_m\}$$

In other words, we remove from the anti-matter set, zones that are known to be matter. Second, A' is partitioned into connected components $A_1,\ldots,A_n$ of the transitive closure of $\mathit{Shape}(①)$:

$$(A_1,\ldots,A_n)=\mathit{ConnectedComponents}(A')$$

This splits the different zones of presumed waste into non-mutually aliasing sets. Third, a specialised matter set is tailored for each anti-matter component A_i:

$$M_i=\{z_m\in M\mid\exists z_a\in A_i,\mathit{Shape}(①)(z_a,z_m)\geq\top\}$$

In other words, we remove from the matter set, zones that are not threatened by the deallocations of the anti-matter set.

A final step eliminates trivial calls to CLEAN: if one of the anti-matter sets A_i is empty, the corresponding call to CLEAN is removed altogether.

5.5.4 Compiling CLEAN

In this phase of asap's transformation, the compile-time function CLEAN is expanded into µL code.

Gc-like primitives The compile-time function CLEAN expands into code that uses several primitives which we list below.

Note that the name of the primitives (below) were chosen to make CLEAN easier to grasp; specifically they were chosen to highlight the similarities with different functions of gcs. However, it is important to note that these primitives do not behave like the corresponding operations in gcs: they merely hold the same role in managing the memory. The most significant difference is that, when the code is generated, at each application of one of the primitives, the type of all of its argument is known. As a result, the primitives for, say, lists need not be the same as the primitives for, say, ethernet frames. Moreover, the primitives are selected during compilation: no runtime-type information is needed, no branching during execution is required, values of different types can have different memory representations. We explore later how this can be leveraged to let the programmer customise the memory representation of values.

The *cleaning primitives*, necessary to expand the compile-time function CLEAN, are: $\mathit{newMarks}$`()` which initialises an empty mark-set, mark`(`v`,`a`)` which records the

```
    (* Scan the matter set and mark the addresses in $matter$*)
(1) let $matter$ = $newMarks$() in
(2) $\sim(SCAN(u_1, \alpha_{u_1}, q_1, MARK_M, NOOP_M))$; ... ;
        $\sim(SCAN(u_n, \alpha_{u_n}, q_n, MARK_M, NOOP_M))$;
    (* Scan the anti-matter set and free it *)
(3) let $anti$ = $newMarks$() in
(4) $\sim(SCAN(v_1, \alpha_{v_1}, p_1, NOOP_A, FREE_A))$; ... ;
        $\sim(SCAN(v_m, \alpha_{v_n}, p_m, NOOP_A, FREE_A))$;
    (* clean up ASAP's own mess *)
(5) $freeMarks$($matter$);$freeMarks$($anti$);
    (* back to business *)
    $t$
```

where the compile-time functions $MARK_M$, $NOOP_M$, $NOOP_A$ and $FREE_A$ are defined as

```
(6) $MARK_M(v) = \langle mark(v, matter) \rangle$
    $NOOP_M(v, \alpha) = \langle \{\} \rangle$
    $NOOP_A(v) = \langle \{\} \rangle$
    $FREE_A(v, \alpha) =$
        ⟨ (7) if $marked(v, matter)$ then {} else
          (8) if $marked(v, anti)$ then {} else
          (9) $mark(v, anti)$; $free(v)$ ⟩
```

Figure 5.9: Expansion of $CLEAN$

address of v into the mark-set a, $marked(v, a)$ which tests whether the address of v is recorded in the mark-set a, $freeMarks(a)$ which deallocates the mark-set a, and finally $free(v)$ which deallocates the memory at the address of v. Note that $CLEAN$ is a compile-time function (hence, upper-case) whilst the primitive are meta-variables for µL identifiers (hence, lower case). We give all the details about these function in Section 5.6.

Compiling $CLEAN$ Using the function $SCAN$ defined in Chapter 4, it is possible to expand the pseudo-primitive $CLEAN$. The term

$$\sim(CLEAN((v_1, p_1),\ \ldots,\ (v_n, p_n))((u_1, q_1),\ \ldots,\ (u_m, q_m)));\ t$$

expands into the term presented in Figure 5.9. The expanded code operates in two steps during execution: first, gather all the addresses reachable through the matter set in the mark-set $matter$ – lines (1), (2), (6) – and, second, deallocates all the memory in the anti-matter set – lines (4), (9) – except for those marked in the mark-set $matter$ – line (7) – or already freed – lines (3), (8), (9).

Note that the mark-sets $matter$ and $anti$ are always in scope when they are passed to the cleaning primitives. These mark-sets, depending on the run-time implementation (see Section 5.6), may use some heap space. In this case, the space is deallocated using the $freeMarks$ primitive – line (5).

Also remember that memory cycles in µL are not possible because the language has no mutation construct; thus, the code presented here makes no effort to detect them. Issues related to cycles are explored in Chapter 6.1.

Immediate deallocations The generated code above can seem heavy-handed. However, remember that the anti-matter set A has been partitioned into smaller sets $A_1, \ldots, A_n$ and the matter set M specialised into smaller sets $M_1, \ldots, M_n$ – as described earlier. This leads to two major optimisation opportunities.

First, if there is no aliasing between one of the A_i and M, then the corresponding matter set M_i is empty. As a result, for the deallocation of A_i, ASAP specialises the code above by eliminating both the mark-set $matter$ and the test $marked\,(v\,, matter)$ from $FREE_A$. Second, if one of the anti-matter sets A_i has no internal sharing (i.e., the $Share$ property for each of its zone is 0 and the $Shape$ property between any pair of its zones is also 0) then ASAP eliminates the mark-set $anti$ and the test $marked\,(v\,, anti)$ from $FREE_A$. When both conditions are true, the memory can be deallocated without any dynamic check whatsoever: the mark sets $matter$ and $anti$ are ignored, $FREE_A$ is merely *free*. In this case, the deallocation is immediate.

For simplicity of exposition, we defined the function $SCAN$ to generate scanning code for the general case and exposed different optimisations that can be applied to it. However, generating optimised scanning code directly is not fundamentally harder.

5.6 Execution-time primitives and programmer's involvement

It can appear, despite the claims made about ASAP, that runtime code is actually needed. Indeed, the cleaning primitives ($newMarks$, $mark$, $marked$, *freeMarks*, and *free*) are used by ASAP's cleaning code. We explore this aspect of ASAP in depth now. Specifically, we argue that these primitives do not constitute runtime code.

5.6.1 Type specialisation

The five cleaning primitives are required by programs under ASAP: they must be provided for the execution to proceed. However, unlike the primitives of a GC, these functions are used in specialised code synthesised at compile-time. Thus, for every application of one of the primitives, the types of their arguments are known. As a result, each primitive can have distinct implementations: one for each type of the program.

More accurately, ASAP uses five *families* of primitives, each indexed by types. As a result, there is no need for a uniform memory representation of values: the padding of records can vary, the alignment policy within blocks can vary, the position of discriminants in sum blocks can vary, etc. This is different from GCs which rely on a uniform memory representation to be able to explore the memory graph.

5.6.2 Actuators

Before we give examples of possible cleaning primitives, we show how they can be provided by either of the three actuators: compiler, runtime, programmer.

Compiler The compiler can generate cleaning primitives. Indeed, the back-end of the compiler is aware of the memory representation of values. (This is necessary to emit code that loads and store values in fields of values.) In the case when this memory representation has spare bits – e.g., for padding –, the compiler can emit specialised primitives that leverage these bits by marking blocks like a GC would.

Note that the compiler cannot generate cleaning primitives for all values of a program. Consider specifically, system software that write messages on hardware components: their representation must follow specifications exactly.

Runtime The runtime code can include cleaning primitives. In this case, the memory representation cannot be customised: it has to fit the requirements of the runtime code. This option strips ASAP of a major benefit and makes for a moot alternative. However, as highlighted above, the cleaning primitives are parameterised by types. Thus, it is possible to use a generic set of cleaning primitives provided by the runtime for some types whilst using a specialised set for other types. We give more details below when exploring hybrid approaches.

Programmer The programmer can provide cleaning primitives. For any type defined in the program, the programmer can write five functions to be used by ASAP. This is most useful if the source language also allows the programmer to specify the layout of values. In that case, a type declaration comprises three components: a language description (which specifies how the programmer constructs, destructs and operates on the values of that type), a representation description (which specifies how the values of that type are laid out in memory) and a set of cleaning primitives (which specifies how ASAP handles the values of that type).

Note that, in some cases, the primitives should be written in a programming language featuring low-level memory access such as reading from and writing to specific offsets. We give examples in Section 5.6.3.

Hybrid As mentioned above, the cleaning primitives are parameterised by types. As a result, distinct actuators can provide the cleaning primitives of distinct types.

This is useful for systems programming. Indeed, as mentioned before, control over the memory representation of values is important in systems programming because it enables zero-copy interaction with hardware components. However, not all values are sent over the network or blitted onto the graphic memory. Consider the case of a priority queue of messages: the messages are meant to be sent out, but the priority queue is for internal bookkeeping only. Values of the former type should be represented as specified in the definition of the messaging protocol. Values of the latter type can be represented however is convenient for the local system. In this case, the programmer would declare a type with custom representation and cleaning primitives for the messages. For the queue, the programmer would declare a type and let the compiler decide on the appropriate representation.

What ASAP provides is an optional involvement of the programmer: for types that matter, the representation can be tuned and for types that do not, it can be ignored. By contrast, in C it is impossible to opt-out, in ML it is impossible to opt-in.

5.6.3 Example of cleaning primitives

We now describe examples of cleaning primitives that can be provided by either of the three actuators.

It is possible to implement the primitives in a GC-like fashion: relying on spare bits in the memory representation of some value. These bits are set in the *mark* function and read in the *marked* function. The functions *newMarks* and *freeMarks* are no-op. This approach relies on the availability of spare bits in the memory representation. In cases where the memory representation is not customised by the programmer, spare bits can be added to any memory block.

Another solution for implementing the primitives is to represent mark-sets as hash tables. These are allocated with $newMarks$ and used to record the visited address with $mark$. Checking for markedness with $marked$ is checking an address belongs to the hash table.

Similarly, there are several possible implementations for the *free* primitive. It can actually free the memory on the spot. Alternatively, it can simply place the address of its argument in a work queue which a separate thread consumes, freeing all the blocks it gets.

5.7 Alternative approximation of waste

The presentation of ASAP above approximates waste by non-$Access$, which is essentially a generalisation of liveness. In this respect, ASAP tries to be as timely as possible (see discussion of timeliness in Chapter 3).

It is possible to reduce the amount of scanning by sacrificing some timeliness. Consider the case of a term of the form:

$$\begin{array}{l}
\sim(CLEAN(M_1)(A_1));\\
\textcircled{1}\ \texttt{let}\ x = e_x\ \texttt{in}\\
\sim(CLEAN(M_2)(A_2));\\
\textcircled{2}\ \texttt{let}\ y = e_y\ \texttt{in}\\
t
\end{array}$$

Depending on the program source, it might happen that part of the heap is scanned at both program points ① and ②; this happens when a zone appears in both the matter set M_1 and the anti-matter set (A_2). In this case, it is possible to batch the deallocations together to avoid scanning the same zone twice. Batching the deallocations together produces the term:

$$\begin{array}{l}
\textcircled{1}\ \texttt{let}\ x = e_x\ \texttt{in}\\
\sim(CLEAN(M_2)(A_1 \cup A_2));\\
\textcircled{2}\ \texttt{let}\ y = e_y\ \texttt{in}\\
t
\end{array}$$

Taking this batching strategy to the extreme, we can push all the deallocations to the return points of each function. This extreme approximation of waste is based on scope: values are deallocated when the function returns and they fall out of scope. It is not possible to deallocate values later than this because, when a function returns, its local values are popped off the stack and their memory blocks become unreachable.

5.8 Cache friendliness

Notice that ASAP tries to minimise the amount of scanning that is done for each call to $CLEAN$: the sizes of the matter and the anti-matter sets are reduced using analysed information. As a result, ASAP deallocates often, a few values at a time.

In particular, ASAP tries to deallocate a value immediately after its last use. ASAP also tries to only scan immediately around the values it deallocates. Thus, ASAP's deallocation instructions exhibit both temporal locality (values that are accessed during deallocation were accessed recently) and spatial locality (values that are accessed during deallocation are often part of the same, bigger data structure). Locality is generally beneficial to cache efficiency [32].

Two factors can reduce locality. Fragmentation of the heap reduces spatial locality: blocks belonging to the same data structure can be stored far apart. Batching deallocations as detailed in Section 5.7 reduces temporal locality: values are deallocated further from their last use.

Additionally, remember that deallocation code is composed of specialised chunks, embedded directly into the program. As a result, the deallocation instructions are loaded into the instruction cache with the rest of the program instructions. This can have both negative and positive effects on the instruction cache. Specifically, with deallocation instructions embedded, some section of the program can become larger to the point where they do not fit in the instruction cache. On the other hand, because the deallocation instructions are loaded into the cache with the rest of the instruction, they do not trigger any additional instruction fetching.

Chapter 6

Language extensions

In this chapter, we explore two additions to the µL intermediate representation: mutability (in Section 6.1) and polymorphism (in Section 6.2). These extensions require modifications of the analyses and transformation of ASAP. The combination of mutability and polymorphism induces additional complexity which we tackle in Section 6.3.

6.1 Mutability

One feature µL lacks is mutability. Note that it is possible to transform a program with assignment into one without. A famous way to achieve this is with a state monad – e.g., the Haskell State monad [27]. However, there is a major concern with that approach under ASAP: the value holding the state would have deep aliasing with many other values. In other words, with a state monad, the state is a high-degree node in the *Shape* graph. Thus, using a state monad would slow down ASAP's compile-time analyses and could potentially reduce their precision which, in turn, increases the necessary amount of scanning and degrades the execution-time performance.

We take a more straightforward solution: adding a mutation operator to µL. We make this decision because of the effects of state monads mentioned above, but also to prove that both the analyses and transformations of ASAP can be extended to support more advanced IRs. Additionally, it offers the opportunity to deal with cycles in memory.

We detail the changes to the IR in Section 6.1.1 and the changes to the analyses in Section 6.1.2. Then we go over the repercussions this addition has on the synthesised collection code in Sections 6.1.3 and 6.1.4. Finally, we show in Section 6.1.5 that mutable values shared between functions require special attention.

6.1.1 µL+<-

The intermediate representation µL+<- is the same as µL with the addition of the mutation operator <-. More specifically, we add a category of constructs called *stm* (pronounced "statement") which contains an assignment operator: x.F <- y replaces the content of the field F of x by the value of y. Note that y is not deep-copied, instead the instruction introduces aliasing between x.F and y. The effects of the assignment operator are shown in Figure 6.1.

Statements can be used in terms through the newly introduced sequence operator ;. Note that ; is also syntactic sugar to sequence terms and expressions – e.g., in f() ; g().

The µL+<- grammar is presented in Figure 6.2. For contrast, elements inherited from µL are faded whilst newly introduced constructs are not. Note that, keeping with the ANF roots of µL, the mutation construct of µL+<- forces all the values involved to be named.

Also note that no direct variable update is possible. To be more precise, only fields of a record can be mutated; the form x <- y is not allowed. Whilst it would be possible to integrate direct variable update, it would result in a mutation of a stack value which is a lesser issue when tackling

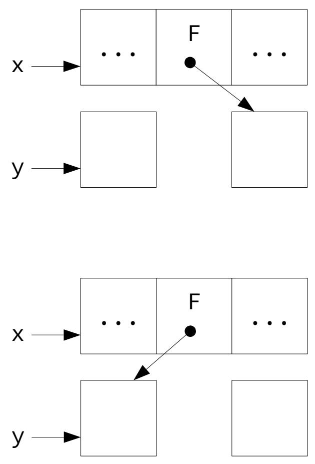


Figure 6.1: Before and after the x.F <- y mutation

the problem of memory management. Thus, adding direct variable mutation would only make the exposition longer but not provide any insight. We also neglect mutation of sum variants for the same reason and because it is an uncommon feature of programming languages[1].

There are no changes to the type grammar. Note that, at the source level, the type system might distinguish mutable and immutable fields but that distinction is not needed for the analyses and transformations of ASAP.

6.1.2 Changes to the analyses

The first change we focus on is that of the analyses.

Concerns

There are various concerns that arise with mutations. We list them now before defining the new analyses.

Cycles Mutations can produce cyclic data-structures which is impossible in vanilla µL. Cycles are accounted for by the *Share* property, defined in Chapter 5. Recall that $Share(\textcircled{\pi})(x, p)$ is the 3VL-certainty that p recognises two (or more) distinct sequences of dereference that, starting from the address of x, lead to the same memory cell at program point $\textcircled{\pi}$ for any execution.

When there is a cycle in memory, it means there is a variable x, a path p_1 and a non-trivial path $p_2 \neq \epsilon$ such that $Z(x, p_1) = Z(x, p_1 \cdot p_2)$. That is, there is a memory block (of $Z(x, p_1)$) from which a finite number of memory accesses (described by p_2) leads back to itself. As a result $Share(\textcircled{\pi})(x, p_1 \cdot (p_2{}^*)) \neq 0$.

Thus, we simply need to ensure that *Share* analysis correctly handles mutation. This leads to our second concern: how to handle mutation in analyses.

[1]In mezzo [33], both the content and the discriminant of a variant can be updated: the statement `tag of x <- Cons` is valid.

$$
\begin{array}{rll}
\mathit{value}, \mathit{pattern} ::= & \mathit{literal} & \\
| & \mathit{variable} & \\
| & \mathit{discriminantname}\ \mathit{variable} & \\
| & \{\ \mathit{fieldname}\texttt{=}\mathit{variable}\texttt{;}\ \ldots\texttt{;}\ \mathit{fieldname}\texttt{=}\mathit{variable}\texttt{;}\ \} & \\
\mathit{expr} ::= & \mathit{value} & \\
| & \mathit{op}\texttt{(}\mathit{variable}\texttt{,}\ \ldots\texttt{,}\ \mathit{variable}\texttt{)} & \\
| & \mathit{funname}\texttt{(}\mathit{variable}\texttt{,}\ \ldots\texttt{,}\ \mathit{variable}\texttt{)} & \\
\mathit{stm} ::= & \mathit{variable}\texttt{.}\mathit{fieldname}\ \texttt{<-}\ \mathit{variable} & \text{(mutation)} \\
\mathit{term} ::= & {}^{\odot}\texttt{let}\ \mathit{variable}\ \texttt{:}\ \alpha\ \texttt{=}\ \mathit{expr}\ \texttt{in}\ \mathit{term} & \\
| & {}^{\odot}\mathit{stm}\texttt{;}\ \mathit{term} & \text{(sequence)} \\
| & {}^{\odot}\texttt{match}\ \mathit{variable}\ \texttt{with} & \\
 & \quad \texttt{[}\ {}^{\odot}\mathit{pattern} \rightarrow \mathit{term} & \\
 & \quad \texttt{|}\ \ldots & \\
 & \quad \texttt{|}\ {}^{\odot}\mathit{pattern} \rightarrow \mathit{term} & \\
 & \quad \texttt{]}^{\odot} & \\
| & {}^{\odot}\texttt{return}\ \mathit{variable}^{\odot} & \\
\mathit{def} ::= & \mathit{funname}\texttt{(}\mathit{variable}\ \texttt{:}\ \alpha\texttt{,}\ \ldots\texttt{,}\ \mathit{variable}\ \texttt{:}\ \alpha\texttt{)}\ \texttt{:}\ \alpha\ \texttt{=}\ \mathit{term} & \\
\mathit{program} ::= & \mathit{def}\ \ldots\ \mathit{def} &
\end{array}
$$

Figure 6.2: Grammar of μL+<-

Additional *Transfer* **function** All three analyses (*Shape*, *Share* and *Access*) are defined by a direction and three *Transfer* functions (for expressions, patterns, and returns). Each of those analyses need an additional *Transfer* function for statements: $\mathit{Transfer}_s$. We define those below.

Invalidation in $\mathit{Transfer}_s$ Mutations are sometimes ominously referred to as *destructive* updates. As the name suggests, the operation can destroy some structures of the heap and invalidate some analysis results. Consequently, the $\mathit{Transfer}_s$ functions for ASAP's analyses can map some zones to 0.

Note that function summaries can also map some zones to 0. This can indicate that mutation happens in the memory of the parameters.

Framework

The analysis framework is mostly unchanged. For terms of μL+<- that also belong to μL, it generates the same equations. For terms of the form ${}^{①}s\texttt{;}\ {}^{②}t$, it generates $P(②) = \mathit{Transfer}_s(s)(P(①))$ for a forward analysis or $P(①) = \mathit{Transfer}_s(s)(P(②))$ for a backward analysis.

The function $\mathit{Transfer}_s$ is defined per-analysis below. The *Share* analysis, which keeps track of cycles, is the one with the most significant changes.

Helper functions

We first define the helper function *through* in Figure 6.3. Given a program point $\textcircled{\pi}$, a zone z, a variable x, and a field F, $\mathit{through}(\textcircled{\pi}, z, x, F)$ is the 3VL-certainty that, when execution reaches $\textcircled{\pi}$, some sequences of memory accesses in the zone z go through the field F of the value held by x. In other words, if $\mathit{through}(\textcircled{\pi}, z, x, F) = 1$ scanning the zone z will dereference the field F of x in every possible execution. If $\mathit{through}(\textcircled{\pi}, z, x, F) = 0$, scanning the zone z will never dereference

$$
\begin{aligned}
\mathit{through}'(\textcircled{\pi}, z, x, F) &= \begin{cases} 1 & \text{if } z \text{ is } (x, (F \cdot p_1 + \cdots + F \cdot p_n) \cdot q) \\ 1 & \text{if } z \text{ is } (x, (F \cdot p_1 + \cdots + F \cdot p_n)^* \cdot q) \\ \top & \text{if } z \text{ is } (x, (p_1 + \cdots + p_n + F \cdot p_{n+1}) \cdot q) \\ \top & \text{if } z \text{ is } (x, (p_1 + \cdots + p_n + F \cdot p_{n+1})^* \cdot q) \\ 0 & \text{otherwise} \end{cases} \\
\mathit{through}(\textcircled{\pi}, z, x, F) &= \bigvee\nolimits_{z'} (\mathit{Shape}(\textcircled{\pi})(z, z') \wedge \mathit{through}'(\textcircled{\pi}, z, x, F))
\end{aligned}
$$

Figure 6.3: The *through* helper function

$$
\begin{aligned}
\mathit{Transfer}_{\mathrm{s}}(x\,.F \texttt{ <-} y) &= \mathit{Close}(m \wedge \neg \mathit{kill} \lhd \mathit{gen}) \\
\text{where } \mathit{kill} &= \left[(z, z') \mapsto \wedge \begin{array}{l} \mathit{through}(\textcircled{\pi}, z, x, F) \\ \neg \mathit{through}(\textcircled{\pi}, z', x, F) \end{array} \;\middle|\; (z, z') \in \mathit{domain}(m) \right] \\
\text{and } \mathit{gen} &= [((x, F), (y, \epsilon)) \mapsto 1]
\end{aligned}
$$

Figure 6.4: The $\mathit{Transfer}_{\mathrm{s}}$ functions for *Shape* analysis

the field F of x in any of the possible execution. If $\mathit{through}(\textcircled{\pi}, z, x, F) = \top$, scanning the zone z will dereference the field F of x in some but not all executions[2].

Shape and Share

The $\mathit{Transfer}_{\mathrm{s}}$ function for the *Shape* property is defined in Figure 6.4 where the auxiliary function *Close* is the same as for the vanilla analysis (Figure 5.4a). The definition uses the form $m \wedge \neg \mathit{kill} \lhd \mathit{gen}$ where m is the decoration at the preceding program point, *kill* indicates the pairs of zones that are separated by the mutation, and *gen* indicates the newly introduced aliasing. The *Close* function generates additional zones based on already computed aliasing information and the single zone $((x, F), (y, \epsilon))$ mentioned in the *gen*.

The $\mathit{Transfer}_{\mathrm{s}}$ function for the *Share* property is defined in Figure 6.5 where $\textcircled{\pi}$ is the preceding program point and the auxiliary function *Close* is the same as for the vanilla analysis (Figure 5.6a). The *kill* component invalidates information made stale by the assignment. On the other hand, the *gen* components integrate newly created sharing.

Access

The $\mathit{Transfer}_{\mathrm{s}}$ function for the *Access* property is defined in Figure 6.6 (the *Close* function is defined in Figure 5.7a.)

Consider the term $^{\textcircled{1}}x\,.F \texttt{ <- } y\,;^{\textcircled{2}}\, t$. Remember that $\mathit{Shape}(\textcircled{2})((x, F), (y, \epsilon)) = 1$. This allows the *Access* property to track implied accesses to y. Specifically, the function *Close* transfers information about explicit accesses to (x, F) into information about implied accesses to (y, ϵ); and vice versa.

[2]As usual, we are satisfied with safe approximations of *through* where $\top$ is used instead of 0 or 1. These approximations are necessary to circumvent the unpredictability of branching.

$$
\begin{array}{rcl}
\mathit{Transfer}_{\mathrm{s}}(x\,.\,F\ \texttt{<-}\,y)(m) & = & \mathit{Close}(m \wedge \neg \mathit{kill} \lhd \left(\begin{array}{l} \mathit{gen}_{\mathrm{star}} \\ \vee \mathit{gen}_{\mathrm{alt}} \\ \vee \mathit{gen}_{\mathrm{prefix}} \end{array}\right)) \\
\text{where } \mathit{kill} & = & [z \mapsto \mathit{through}(\textcircled{\pi}, z, x, F)] \\
\text{and } \mathit{gen}_{\mathrm{star}} & = & \left[\begin{array}{l} (x, (F \cdot p)^*) \mapsto \mathit{Shape}(\textcircled{\pi})((y,p),(x,\epsilon)) \\ (y, (p \cdot F)^*) \mapsto \mathit{Shape}(\textcircled{\pi})((y,p),(x,\epsilon)) \end{array}\right] \\
\text{and } \mathit{gen}_{\mathrm{alt}} & = & [(x, (F \cdot q + q')) \mapsto \mathit{Shape}(\textcircled{\pi})((y,q),(x,q')) \\
 & & \qquad\qquad\qquad\qquad \mid \neg \mathit{through}(\textcircled{\pi}, (x,q'), x, F)] \\
\text{and } \mathit{gen}_{\mathrm{prefix}} & = & [(x, F \cdot p) \mapsto \mathit{Share}(\textcircled{\pi})(y,p)]
\end{array}
$$

Figure 6.5: The $\mathit{Transfer}_{\mathrm{s}}$ functions for Share analysis

$$
\mathit{Transfer}_{\mathrm{s}}(x\,.\,F\ \texttt{<-}\,y)(m) \quad = \quad \mathit{Close}(m \lhd [(x,\epsilon) \mapsto 1])
$$

Figure 6.6: The $\mathit{Transfer}_{\mathrm{s}}$ function for Access analysis

6.1.3 Changes to the code generation

It is important to note that mutation can render some values unreachable. Indeed, the memory block that represented $x\,.\,F$ may become unreachable after the instruction $x\,.\,F$ <- y. As a result, it is necessary to deallocate such memory blocks *before* the mutation. Indeed, unlike GCs which can scan the whole memory and pick-out the unreachable values, ASAP embeds its deallocation code in the program directly. As such, unreachable blocks are also unreachable to ASAP.

Fortunately, the available Shape, Share and Access information is sufficient to emit code to perform this operation safely. Note that even in the case where $x\,.\,F$ aliases with y, ASAP can emit safe and complete deallocation code.

Specifically, we extend the function $[\![.]\!]_{\mathrm{CLEAN}}$ for sequence terms:

$$
\begin{array}{l}
[\![{}^{\textcircled{1}}x\,.\,F\ \texttt{<-}\ y;\ {}^{\textcircled{2}}t]\!]_{\mathrm{CLEAN}} = \sim\!(\mathit{CLEAN}(M)(A));\ {}^{\textcircled{1}}x\,.\,F\ \texttt{<-}\ y;\ [\![{}^{\textcircled{2}}t]\!]_{\mathrm{CLEAN}} \\
M = \{(x,p) \mid \mathit{Access}(\textcircled{2})(x,p) \geq \top, \neg \mathit{through}(\textcircled{\pi}, (x,p), x, F)\} \\
A = \{(x, F \cdot p) \mid \forall p \in \mathit{Wild}(\Gamma(x).F)\}
\end{array}
$$

Note that the matter set M includes all the future accesses to y. If y and $x\,.\,F$ alias in any way at that point in the program (i.e., before the assignment), the shared memory is protected by the dynamic tests that are generated for y.

Note that, in the case where y has already been characterised as useless (i.e., $\mathit{Access}(\textcircled{1})((y,\epsilon)) = 0$), its memory has been deallocated and y is a dangling pointer on the stack. In this case the address of the deallocated memory block is copied into $x\,.\,F$ and, even though $x\,.\,F$ becomes dangling, it does not cause any issues.

The optimisations and refinement presented in Chapter 5.5 also apply to these calls to CLEAN.

Remember in Chapter 5 the discussion about delaying deallocations: moving calls to CLEAN forward in the term can reduce the amount of scanning – but reduces timeliness. Mutations impose hard limits on the movement of calls to CLEAN. More specifically, it is not safe to move calls to CLEAN beyond a mutation. Indeed, delaying these calls would create unreachable values (as explained above). Note, however, that it is always possible to preserve names for mutated parts of the memory using the following transformation:

$$[\![x\texttt{.}F\ \texttt{<-}\ y\texttt{;}\ t]\!]_{\text{name}} \quad = \quad \texttt{match}\ x\ \texttt{with [\{}F\ \texttt{=}\ z\texttt{\} ->}\ x\texttt{.}F\ \texttt{<-}\ y\texttt{;}\ [\![t]\!]_{\text{name}}\texttt{]}$$
$$(z \text{ is fresh})$$

The transformed program includes a name (z) for $x\texttt{.}F$ which prevents it from becoming unreachable.

6.1.4 Changes to the generated code

Another change that is required affects the $SCAN$ compile-time function. Indeed, with the introduction of mutation, cycles can now appear in the memory graph. Exploring memory that has cycles is a different matter than exploring memory that has none.

We equip $SCAN$ with an additional argument: CYC. The CYC function is evaluated during compilation to produce code that detects cycles. When a cycle is detected, the generated scanning code is interrupted. Note that cycles are only ever a problem when exploring a path with repetition (*). Indeed, exploring any other path can be done in a bounded number of steps. As a result, the changes to $SCAN$ are not invasive. We present the refined definition of $SCAN$ in Figure 6.7.

To allow CYC to perform its task (detecting cycles to prevent endless cyclical exploration), the value of PRE must be adapted too. Specifically, PRE must unconditionally mark values in a mark-set that CYC reads from. The result of compiling $CLEAN$ is presented in Figure 6.8. Note that cycle detection can be removed when *Share* information shows that no cycles are present. This can be optimised on a per-$SCAN$ basis.

6.1.5 Changes to function calls

The text above describes how to handle mutation of data that aliases with other values in the same function. However, mutation of values that alias across functions introduces additional issues. We detail what patterns of aliasing are problematic and how to tackle them now.

Consider the program in Figure 6.9. According to the scheme above, during the execution of the function g, the memory block that represents the value x_1 is deallocated[3]. As a result, x_1 points to deallocated memory which is eventually dereferenced by the callee, causing a runtime error. This error arises because decorations of g's body do not mention the aliasing between x_1 and y. Note that g does not even have a handle on x_1 as it belongs in the caller's frame.

More abstract formulation of the problem

Not all functions calls are problematic. The problematic calls are those that make a destructive update to an argument which the caller uses, albeit by a different name, after the call. Figure 6.10 shows a general example of the situation. It reads as follows:

- The function g has two local variables w_1 and w_2.
- The values of these variables alias at the call point $\textcircled{\pi}$:

$$\mathit{Shape}(\textcircled{\pi})((w_1, p_1), (w_2, p_2 \cdot F \cdot p_2')) \geq \top$$

[3]Unless the expression bound to x_2 references $x\texttt{.}F$

```
$SCAN\begin{pmatrix} x, \alpha, \epsilon, \\ PRE, FIN, CYC \end{pmatrix}$ = if $\Delta(\alpha) \neq word$ then
        $\langle \sim(PRE(x,\alpha))$; $\sim(FIN(x,\alpha)) \rangle$
    else
        $\langle${}$\rangle$
$SCAN\begin{pmatrix} x, \alpha, F, \\ PRE, FIN, CYC \end{pmatrix}$ = if $\alpha.F \neq word$ then
        $\langle$ $\sim(PRE(x,\alpha))$;
          match $x$ with
              [ {$F$=$y$} ->
                  $\sim(PRE(y,\alpha.F))$;
                  $\sim(FIN(y,\alpha.F))$
              ] $\rangle$
    else
        $\langle \sim(PRE(x,\alpha))$; $\sim(FIN(x,\alpha)) \rangle$
$SCAN\begin{pmatrix} x, \alpha, D, \\ PRE, FIN, CYC \end{pmatrix}$ = if $\alpha.D \neq word$ then
        $\langle$ $\sim(PRE(x,\alpha))$;
          match $x$ with
              [ $D$ $y$ ->
                  $\sim(PRE(y,\alpha.D))$;
                  $\sim(FIN(y,\alpha.D))$
              | _ -> {} (* do nothing *)
              ] $\rangle$
    else
        $\langle \sim(PRE(x,\alpha))$; $\sim(FIN(x,\alpha)) \rangle$
$SCAN\begin{pmatrix} x, \alpha, p \cdot p', \\ PRE, FIN, CYC \end{pmatrix}$ = $\langle \sim(SCAN(x,\alpha,p,PRE,CONT,CYC)) \rangle$
    where $CONT$ is defined by
        $CONT(y,\alpha') = SCAN(y,\alpha',p',PRE,FIN,CYC)$
$SCAN\begin{pmatrix} x, \alpha, p + p', \\ PRE, FIN, CYC \end{pmatrix}$ = $\langle$ $\sim(SCAN(x,\alpha,p,PRE,FIN,CYC))$;
                                        $\sim(SCAN(x,\alpha,p',PRE,FIN,CYC))$ $\rangle$
$SCAN\begin{pmatrix} x, \alpha, p^*, \\ PRE, FIN, CYC \end{pmatrix}$ = $\langle loop(x) \rangle$
    where $loop$ is a fresh function defined by
        fun $loop(x)$=
            $\langle$ if $\sim(CYC(x,\alpha))$ then
                  {}
              else
                  $\sim(SCAN(x,\alpha.p,p,PRE,LOOP,CYC))$;
                  $\sim(SCAN(x,\alpha,\epsilon,PRE,FIN,CYC))$ $\rangle$
    and $LOOP$ is a staged function defined by
        $LOOP(x,\alpha) = \langle loop(x) \rangle$
```

Figure 6.7: Definition of $SCAN$ with cycle detection

```
let $matter$ = $newMarks$() in
let $cycle$ = $newMarks$() in
$SCAN$($u_1, \alpha_{u_1}, q_1, MARK_M, NOOP_M, CYC$);
$freeMarks$($cycle$);
$\ldots$;
let $cycle$ = $newMarks$() in
$SCAN$($u_n, \alpha_{u_n}, q_n, MARK_M, NOOP_M, CYC$);
$freeMarks$($cycle$);
let $anti$ = $newMarks$() in
let $cycle$ = $newMarks$() in
$SCAN$($v_1, \alpha_{v_1}, p_1, NOOP_A, FREE_A, CYC$);
$freeMarks$($cycle$);
$\ldots$;
let $cycle$ = $newMarks$() in
$SCAN$($v_m, \alpha_{v_n}, p_m, NOOP_A, FREE_A, CYC$);
$freeMarks$($cycle$);
$freeMarks$($matter$);$freeMarks$($anti$);
$t$
```

where the compile-time functions $MARK_M$, $NOOP_M$, $NOOP_A$, $FREE_A$ and CYC are defined as

```
$MARK_M$($x$)=$mark$($x$,$cycle$); $mark$($x$,$matter$)
$NOOP_M$($x$)={}
$NOOP_A$($x$)=$mark$($x$,$cycle$)
$FREE_A$($x$)=
    if $marked$($x$,$matter$) then {} else
    if $marked$($x$,$anti$) then {} else
    $mark$($x$,$anti$); $free$($x$)
$CYC$($x$)=$marked$($x$,$cycle$)
```

Figure 6.8: Expansion of $CLEAN$ with cycle detection

```
let $x_1$ = ... in                  $g$($y$) =
let $y$ = {$F$ = $x_1$} in              let $x_2$ = ... in
$g$($y$);                               $y$.$F$ <- $x_2$;
$\ldots x_1 \ldots$                     return {}
```

Figure 6.9: Mutation of data aliasing across functions

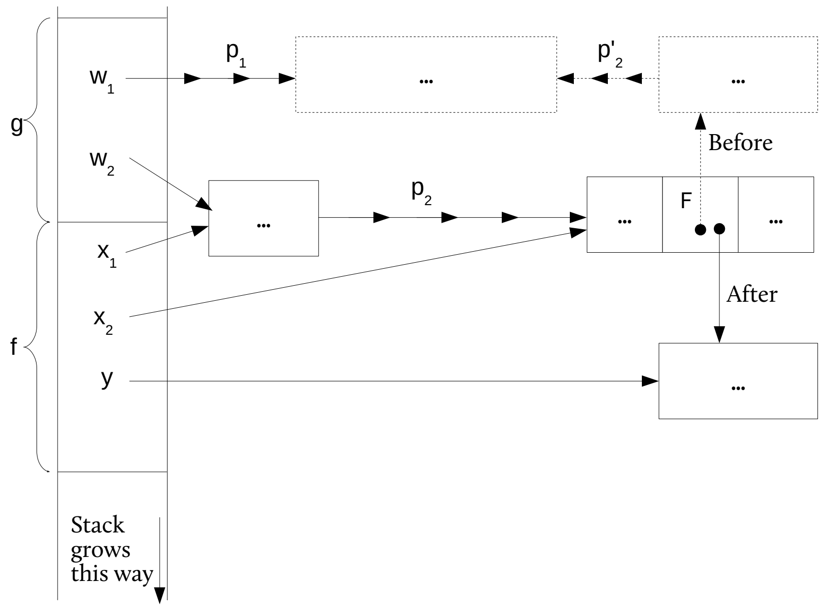


Figure 6.10: Contentious mutation in call

- The function g calls f with w_2 as an argument for the parameter x_1.
- The function f destructs x_1 and obtains x_2 such that

$$Shape((x_1, p_2), (x_2, \epsilon)) \geq \top$$

- The function f mutates x_2, executing x_2.F <- y (where y is a local variable).

Note that, from the point of view of f, the mutation does create unreachable memory blocks. Specifically, some of the memory becomes unreachable from the roots available to f (i.e., unreachable from x_1, x_2 and y) – marked with dashed lines and boxes on Figure 6.10. However, not all this memory should be deallocated. Specifically, if g accesses (w_1, p_1) after the call, it will dereference values located in that part of the memory that has become unreachable from f.

There is a subtlety in this issue. Some of the memory actually becomes unreachable. Thus, in order to ensure correctness, f must deallocate some of the memory that is reachable before the mutation through (x_2, F), but not all of it.

Formalisation of the problem

To formalise the problem we must be able to express the following properties: some aliasing exists between local variables of a caller, one of the local variables is passed to a callee, and the callee destroys some of the aliasing mentioned above. Moreover, in order for ASAP to detect the problematic mutations, we must express these properties using the $Access$, $Shape$ and $Share$ properties. Indeed, these are the only three properties that ASAP has access to.

It turns out that ASAP's analyses cannot in general express the property that mutation happens. (Note that, in some cases, the $Shape$ summary of a function can be used to detect some mutations. Specifically, if there is aliasing between two parameters and a mutation destroys it, then the summary mentions it. But, apart from this specific case, there is no way to tell.)

We augment ASAP with an effect system that tracks mutations. Specifically, the effect system tracks what, if any, parts of the parameters are mutated. More formally, we introduce a new property $Mutate$ such that $Mutate(f)(x, p, F)$ is the 3VL-certainty that the procedure f mutates the field F of the blocks of the zone (x, p) – note that x is a parameter of f. The effect analysis is made easier by the availability of the $Shape$ analysis. Indeed, in the body of a function f, every time a mutation x.F <- y takes place, the aliasing between x and the parameters of f is known. The effect analysis, in the body of a function f with parameters $x_1, \ldots, x_n$, finds all the mutations ${}^{\textcircled{\pi}}x$.F <- y. For each of these mutations, it gathers $[(x_i, p, F) \mapsto Shape(\textcircled{\pi})((x_i, p), (x, \epsilon))]$. Additionally, the effect analysis, in the body of a function f with parameters $x_1, \ldots, x_n$, finds all the calls ${}^{\textcircled{\pi}}g$(y_1, …, y_n). For each of these calls, it gathers $[(x_i, p, F) \mapsto \bigcup_j Mutate(g)(y_j, p, F) \cap Shape(\textcircled{\pi})((x_i, p), (y_j, \epsilon))]$.

We can now formally characterise problematic functions. They are the `fun` f(x_1,…,x_n) = t such that:

- There is a call ${}^{\textcircled{1}}f$(y_1,…,y_n)${}^{\textcircled{2}}$ with a local variable y in scope such that
  - $Shape(\textcircled{1})((y, p_y), (y_i, p_i \cdot F \cdot p'_i)) \geq \top$ and
  - $Access(\textcircled{2})(y, p_y) \geq \top$.
- There is a mutation in f such that $Mutate(f)(x_i, p_i, F) \geq \top$.

The first condition selects functions that receive arguments with dangerous aliasing. The second condition selects functions that perform mutations in the dangerously aliasing area of memory.

Proposed solution

Having characterised which calls are problematic, we now describe a set of changes that render these calls safe. (Note that there are other set of changes that solve the issue of problematic calls highlighted above; we only explore one.) The changes provide additional parameters that carry information from caller to callee. With this added information, the callee can safely decide what part of the memory should be deallocated.

Consider a function f with parameters $x_1, \ldots, x_n$ which performs mutations such that, first, $Mutate(f)(x_i, p_i, F) \geq \top$ and, second, blocks of the zone $(x_i, p_i \cdot F \cdot p'_y)$ is used by the caller after the f returns through (y, p_y). We add two parameters x_{keep} and p_{keep}. On each call site, the caller can use these parameters to pass, during execution, a root and a path that need to be preserved. During the execution of f, at the mutation point, the parameters x_{keep} and p_{keep} are added to the matter-set.

This proposed solution departs from the as-static-as-possible policy: additional arguments are passed during execution. Moreover, the path that is received (via p_{keep}) is interpreted at execution instead of pre-compiled. However, this solution has a major advantage: different callers can pass different zones to preserve different parts of the heap.

Cost

Using mutation under ASAP induces extra costs during execution. First, it reduces the possibilities to move calls to $CLEAN$. Remember that moving these calls can reduce the amount of scanning performed during execution.

Second, mutating shared state adds a runtime cost. Two additional arguments are used in problematic function calls to carry information about a zone that must not be deallocated. Scanning along the zone specified by these two arguments is done by execution-time interpretation of a path.

Even though the cost can seem entirely acceptable, it should be noted that a single mutation can be problematic for several of the caller's local variables. Indeed, consider the case where a caller holds two pointers to distinct elements of a list and the callee mutates the spine of the list: each of the elements kept by the caller must be scanned. In this case, the function takes additional arguments for each of the zones that must be kept.

Note however, that, on call points where no memory needs to be kept (i.e., when $Access(y, p_y) = 0$), the ϵ path can be passed. As a result, no execution-time $SCAN$ning occurs at mutation.

6.2 Polymorphism

We now add polymorphism to µL and explore its effect on ASAP.

6.2.1 µL+$\forall\alpha$

The types of µL are augmented with constructors for parametric polymorphism. Terms of µL use the augmented syntax for types but are otherwise unaffected.

$$\begin{array}{rll}
\textit{product} ::= & \{\,\textit{fieldname} : \tau;\ \ldots;\ \textit{fieldname} : \tau\} & \\
\textit{sum} ::= & \textit{discriminantnamename}\ \tau\ +\ \ldots & \\
& \quad +\ \textit{discriminantnamename}\ \tau & \\
\tau ::= & \textit{word} \ \mid\ \textit{product} \ \mid\ \textit{sum} & \\
& \mid\ \textit{typename typevariable} & \text{(application)} \\
& \mid\ \textit{typevariable} & \text{(variable use)} \\
\Gamma : & \textit{variable} \to \tau & \text{(type environment of variables)} \\
\Delta : & \textit{typename} \to \tau &
\end{array}$$

Type variables are introduced when defining a type.

6.2.2 Impact on paths

Polymorphism impacts paths and zones in two ways. First, given a polymorphic type τ, it is not possible to compute the wild path set $\mathit{Wild}(\tau)$. Indeed, computing a wild path set requires a complete exploration of the type definition in order to find recursive occurrences and non-recursive terminations. Consider the wild path set for a list of pairs (as per the example in Chapter 4): it mentions the *fieldname*s that belong to the pair type. With a polymorphic list, $list\ \alpha$, it is impossible to generate the paths that explore α (because it is not known what paths are compatible with α).

Second, the $SCAN$ function relies on type information to generate code. Types are important for avoiding scanning $word$ values and to select appropriate destructors.

6.2.3 Example

We use the example in Figure 6.11 to showcase these two impacts. Note that this function is recursive: its summaries and call contexts are fixpointed.

$$\begin{aligned}\forall\alpha, \mathit{cons}\ \alpha &= \{\mathit{Head}\colon \alpha;\ \mathit{Tail}\colon \mathit{list}\ \alpha\}\\ \forall\alpha, \mathit{list}\ \alpha &= \mathit{Cons}\ (\mathit{cons}\ \alpha) + \mathit{Nil}\ \{\}\end{aligned}$$

(a) Type definitions

```
length(xs: list α): word =
    match xs with
        [ Cons {Tail = ys} ->
            let l : word = length(ys) in
            let r : word = l + 1 in
            return r
        | Nil {}->
            return 0
        ]
```

(b) Function definition

Figure 6.11: Polymorphic *length* function

6.2.4 Summaries and call contexts with parametricity

We now explain why, despite the unavailability of a widening operator for polymorphic values, both the summaries and call contexts are computable.

Fixpointing the summaries and call contexts in ASAP's analyses requires the widening function *Widen* which relies on the wild path set. However, notice how the *length* function never interacts with the elements of its list argument. As a result, paths for polymorphic elements are never produced during computation of summaries. Specifically, analysing *length* leads to sets of zones of the form

$$\{(xs,\epsilon),(xs,\mathit{Cons}),(xs,\mathit{Cons}\cdot\mathit{Tail}),(xs,\mathit{Cons}\cdot\mathit{Tail}\cdot\mathit{Cons}),\ldots\}$$

which widens to

$$\{(xs,(\mathit{Cons}\cdot\mathit{Tail})^*),(xs,(\mathit{Cons}\cdot\mathit{Tail})^*\cdot\mathit{Cons})\}$$

The widening operation naturally never generates the wild path set for the polymorphic part of the input. This is due to parametricity [40]: the function must work on lists of α for every possible α and thus cannot actually operate on the α values themselves.

Call contexts are different: they are amalgamated from different program points. At each of these program points the polymorphic parameter can be instantiated with different types. E.g., one call might compute the length of a list of words, another the length of a list of pairs. Naïvely amalgamating paths for values of different types leads to confusing results: a map containing different paths for different types. Fortunately, it is not necessary. Indeed, remember the parametricity argument made above that a function never accesses the polymorphic parts of their arguments. Thus, before amalgamating the call contexts, it is safe to substitute the parts of the path that correspond to the polymorphic part of the arguments by ϵ. E.g., for any path p, transforming $[(x,\mathit{Cons}\cdot\mathit{Head}\cdot p)\mapsto 1]$ into $[(x,\mathit{Cons}\cdot\mathit{Head}\cdot\epsilon)\mapsto 1]$ when x is an argument of the *length* function.

Note that the amalgamated call context as computed here is a safe approximation of the actual amalgamated call context (as detailed in Chapter 5). To distinguish the two in the discussion below, we call polymorphic the amalgamated call contexts where ϵ is used as a substitute to the paths that correspond to the polymorphic parts of a parameter. Polymorphic amalgamated call contexts are safe, not because they conservatively approximate reality, but for reasons that we explain in Section 6.2.6

6.2.5 Wild path sets in $\mathit{Transfer}$ function for Access

Another use of wild path sets appears in the Access analysis. More specifically, the $\mathit{Transfer}_{\mathrm{r}}$ (defined in Figure 5.7d) function uses it to initialise the value of Access of all the variables in scope to false: $\mathit{Close}([(y, \mathit{Wild}(\Gamma(y))) \mapsto 0] \lhd m)$

In this case, as for amalgamated call contexts, it is safe to replace the polymorphic parts of the wild path set by ϵ. Indeed, because a polymorphic function never accesses the polymorphic part of their local variables, we know that the Access for this part will always be false. As a result, we only ever need to keep track of the surface of the polymorphic part of the memory but never the inside. Thus, replacing the polymorphic part of the wild path sets by ϵ is safe here.

6.2.6 No deallocations

We now look at the different possible values for the Access polymorphic amalgamated call context. Specifically, we consider the Access polymorphic amalgamated call context for the elements of the list: $f^{\uparrow}(\mathit{xs}, \mathit{elems})$ where the path $\mathit{elems} = (\mathit{Cons} \cdot \mathit{Tail})^{*} \cdot \mathit{Cons} \cdot \mathit{Head} \cdot \epsilon$ – notice that ϵ's place in the path corresponds to α's place in the type. We show that the length function does not perform any deallocations for this path, regardless of the computed Access call context.

If $f^{\uparrow}(\mathit{xs}, \mathit{elems}) = 0$ (i.e., the elements of the list are never used after calls to f), then the memory representing the elements of the list is always deallocated before calls to length. Indeed, at any call point, the Access for the elements of the arguments are already 0 because there are no uses during nor after the call. As a result, the elements have already been deallocated.

If $f^{\uparrow}(\mathit{xs}, \mathit{elems}) = 1$ (i.e., the elements of the list are always used after calls to f), then the length function does not deallocate anything. Indeed, because of the Access call context the zone $(\mathit{xs}, \mathit{elems})$ is in the matter set at every deallocation point which guarantees both the elements and their prefix (i.e., the whole list) is not deallocated.

In the case when $f^{\uparrow}(\mathit{xs}, \mathit{elems}) = \top$ (i.e., the elements of the list are dereferenced after some calls), calls trigger the same deallocations as when it is 1.

6.3 Mutability and polymorphism

Note that, as far as ASAP is concerned, the mutability and polymorphism are not orthogonal. Indeed, polymorphic functions with mutations generate additional complications.

6.3.1 Instance of compound complications

Consider the function set in Figure 6.12. As explained in Section 6.1, the deallocation of $r\,.\,F$ must happen before it becomes unreachable. Also note that it cannot happen before the function call, because the result of the conditional is not known yet. Thus, the function set is responsible for deallocating $r\,.\,F$ at program point $\textcircled{\pi}$.

This deallocation requires the ability to scan both $r\,.\,F$ and (in case the two arguments alias) x. This is impossible because SCAN relies on type information.

$$\forall\alpha, \mathit{ref}\ \alpha \quad = \quad \{\mathit{Content}\colon \alpha\}$$

(a) Type definition

```
$set(r\colon ref\ \alpha,\ x\ \colon\alpha)\colon \{\}$ =
    if ... then
        $\textcircled{\pi} r.F$ <- $x$;
        return {}
    else ... then
        return {}
```

(b) Function definition

Figure 6.12: Polymorphic mutation function

```
let l : 'a     list ref = ref [];;                   (*create polymorphic reference*)
let ls: string list ref = l;;                      (*specialise reference to string*)
ls := "kerboom" :: !ls;;                           (*insert string in the reference*)
let li: int    list ref = l;;                         (*specialise reference to int*)
li := [ 3;2;1 ] :: !li;;                             (*insert ints in the reference*)
List.fold_left (+) 0 !li;;                           (*ERROR when adding the string*)
```

Figure 6.13: Runtime error under naïvely typed combination of mutation and polymorphism

6.3.2 Relation to value restriction

The combination of mutation and polymorphism is known to cause issues when extending the Hindley-Milner type discipline (which already supports polymorphism) to support mutation. Specifically, naïvely combining mutation and polymorphism can lead to programs where a mutable cell holds values of different types in different parts of the program. It is then possible to cause runtime errors as shown in Figure 6.13.

Wright proposes a solution to this issue [42] known as *value restriction* or *value-only polymorphism*. Under value restriction, the type checker rejects programs in which non-values have a polymorphic type. Non-values are expressions other than literals, variables, functions, and constructors only if they are different than `ref` and if all their components are values. With this restriction, the program of Figure 6.13 is rejected. Specifically, on the first line, the `ref` constructor cannot be given a polymorphic type.

Asap's issue with the combination of mutation and polymorphism is different. Specifically, the mutation forces asap to perform a deallocation, and the polymorphism prevents asap from emitting deallocation instructions. Asap's issue arises under the same combination of features but not for the same patterns of code.

6.3.3 Proposed solutions

For asap to handle both polymorphism and mutability at the same time, modifications are necessary. Specifically, functions that mutate the polymorphic parts of their arguments need additional information. These can take several alternative forms, some of which we briefly explore now.

Higher-order scanning A possible solution to the issue highlighted above is to pass, as arguments, functions that perform the scanning. In this situation, the scanning function is generated in the caller – for which the type is available – and passed to the callee. However, note that µL does not have higher-order functions. Thus, they need to be either eliminated (through defunctionalisation [14] as mentioned in Chapter 2) or supported by the back-end.

Execution-time scanning with type reification Another possible solution is to pass, as an argument, a reification of the polymorphic type. In this case, the caller scans values during execution based on the reified type description. This only works because the caller treats each of the polymorphic values as a whole: i.e., it either saves the whole value or deallocates it all, there is no middle ground. In the example above, the function set would deallocate the whole of $r.F$ but save the whole of x.

Note that this solution has a runtime component: a function that, given a reified type, can scan a value of that type. This runtime component can either compile the reified type (into code that scans values of that type) or it can interpret the path (to scan the values directly). In either case, the representation of values in memory is left untouched: the runtime-type information is passed along rather than inside the values.

Also note that this method is coarse: it only ever treats values as a whole. Specifically, it cannot optimise the scanning code based on aliasing knowledge. By contrast, in vanilla ASAP it is possible to specialise the scanning code for the situation where, say, two lists might share elements but not any part of their spine.

Inlining Functions such as set above can be inlined. This removes the need for both amalgamated call contexts and polymorphism. However, inlining is not always an option: the function may be big or recursive.

Monomorphisation Using monomorphisation, the function set can be replicated as many times as necessary, such that each one is called from a single, non-polymorphic type.

6.4 Concurrency

In its current form, ASAP is not equipped to handle concurrent programs. Managing memory in concurrent programs poses its own set of challenges. We leave these challenges to be answered in future work and only survey them here. For ASAP specifically, these challenges fall into two categories.

First, concurrency interferes with the analyses. When several threads can mutate the same values, the Shape and Share analyses have to account for the multiple possible executions. Similarly, when several threads can read the same values, the Access analysis has to account for the multiple possible execution.

Data-flow analysis of concurrent program is an active research topic – e.g., [15, 26]. Dealing with concurrent programs would require adapting solutions found in this field to the analyses of ASAP.

Second, concurrency can interfere with the deallocation code. Specifically, consider code with two data-structures a and b that share cells (e.g., two list that share elements). The program starts two threads A and B. Thread A modifies a in-place to remove some elements; thread B modifies b in-place to remove some elements. When an element is removed from either a or b, deallocation code generated by ASAP must scan both lists in order to decide whether the memory holding the element can be deallocated. Because the threads are modifying the lists concurrently, the deallocation code must avoid race conditions.

Note that this challenge varies with different forms of concurrency. In the case of cooperative concurrency, because yield points are explicit, deallocation code cannot be interrupted. In the case of fork-join concurrency, it is possible to delay the deallocation of shared values until the join point. In the case of work-stealing concurrency, there are no statically known join points and the previous delay cannot be used.

Chapter 7

Implementation

We developed a prototype of ASAP. We now briefly describe its implementation, discuss its limitations, show an example run, and present efficiency measurement.

7.1 Code overview

The prototype is written in OCaml and consists of 2250 lines of code including support modules for 3VL, paths, and zones. An additional 8500 lines of code provided a front-end (using three intermediate representations), a back-end (using an evaluator), pretty printing (for dumping inferred information and intermediate representations), as well as several support modules (for managing environments, unique identifiers, and maps).

Front-end The lexer and parser are automatically generated using `ocamllex` and `menhir`[1]. They produce an AST which is transformed into dL (pronounced "deci-language"), into cL ("centi-language"), into mL ("milli-language"), into µL. Together, these compilation passes provide the syntactic sugar of Chapter 2 as well as the following abstractions: nested functions, nested patterns, nested constructors and nested bindings.

Analyses The prototype analyses all of the three properties: $Shape$, $Share$, $Access$. It is also able to dump the results of the analyses (as demonstrated below).

Transformation Our prototype inserts calls to $CLEAN$ within the analysed code. However, these calls are not currently expanded into the scanning code using the staging approach of Chapter 5. Instead they are interpreted by our back-end.

Back-end The evaluator recursively traverses the program in its final intermediate representation (i.e., after it is analysed and transformed). It emulates memory by maintaining a map from addresses (represented as integers) to memory blocks (represented as native OCaml values). This map is implemented naïvely as a list.

7.2 Prototype limitations

The implementation is a prototype with various limitations not arising from the theory.

[1] `ocamllex` is distributed with OCaml, `menhir` is available at `http://gallium.inria.fr/~fpottier/menhir/`.

```
type cons = { Head:word ; Tail:list }
and nil = { }
and list =
  | Cons of cons
  | Nil of nil

letrec main (): word =
  let x : int =
    match Cons {Head=0; Tail=Nil {} } with
      [ Cons {Tail=Cons _} -> 0
      | _ -> 1
      ]
  in
  x
;;
```

Figure 7.1: Example program triggering the matching bug in the prototype

No calls from a recursive function to a recursive function The prototype does not support programs in which a recursive function makes a call to another recursive function. This is caused by our implementation of paths: in the prototype, only paths of a limited form are allowed. Specifically, the prototype only handles sequences of possibly repeating alternatives of sequences. This restricted form of path is not powerful enough to express the heap properties that appear when recursive functions call recursive functions. Specifically, the prototype cannot handle nested repetitions in paths.

In order to support these programs, the prototype's path library would need to be improved to handle the full range of paths described in Chapter 4 rather than the limiting subset it currently handles.

Nested matching bug Nested patterns are compiled as nested matching operations. However, because of the naïve implementation of the transformation, backtracking from one level to the previous one is not supported. As a result, it is easy to write programs that fail during evaluation due to matching errors.

E.g., the program in Figure 7.1 fails at evaluation time because the matching engine commits to the first branch (upon matching the top `Cons` node) and is unable to backtrack (upon failing to match the `Tail` field).

Performance No effort was made towards any form of acceptable performance in the prototype. In particular, 3VL sets and relations are represented by triplets of lists, one containing values that map to 0, one $\top$ and one 1. Some operations such as transitive closure of aliasing information are particularly inefficient.

The evaluator is implemented as a recursive function traversing the final representation. During this evaluation, an abstract representation of the stack and heap is passed around. This is also inefficient.

7.3 Example

We show the result of the different analyses on a sample program, as inferred by the prototype. The source program, written in the front-end syntax with nested patterns and constructors, is shown in Figure 7.3. Other syntax difference include: `letrec` instead of `fun` and double semi-colon (`;;`) to terminate function definitions.

The result of the analyses is presented in Figure 7.2. Note, first, that the syntactic sugar of the source program has been compiled down – e.g., the nested pattern is replaced by a nested match with the fresh variable `x1`. They are presented as printed by the prototype with additional line breaks for readability. The decorations are presented as triplets `<` *Shape* `%` *Share* `%` *Access* `>` and they read as follows.

1 The 3VL-certainty of *Access* for both (ret, ϵ) (rendered as `_ret.`) and $(\mathtt{l}, \epsilon)$ (rendered as `l.`) is 1.

The 3VL-certainty of *Access* for the other zones is $\top$ as indicated by the question marks. E.g., `?l.Cons.Head` indicates the uncertainty that the zone $(\mathtt{l}, \mathtt{Cons} \cdot \mathtt{Head})$ is accessed.

Note that, for brevity, the prototype does not print reflexive components of *Shape* (such as `l.=l.`) at this point of the program. However, this is not the case for all reflexive components at all program points (see below).

3 The *Shape* decoration indicates that the wildcard pattern (treated as a variable by our prototype) corresponds to the `Nil` component of the list `l` (rendered as `_.=l.Nil`). The decoration also includes some reflexive components.

7 The *Shape* decoration indicates here that the return value (`_ret`) aliases with the value `x0`.

10 The *Shape* decoration indicates that, in this branch, the variable `x1` is bound to the cons cell of `l` (rendered as `l.Cons=x1.`).

11 The *Access* decoration indicates that `l`'s cons cell is further explored. Specifically, the `l1.Cons.Head` appears in the decoration.

14–16 The *Shape* decoration indicates the existence and structure of the aliasing between `l`, `x1` and `h`.

17 This *Access* decoration is particularly interesting. Notice how, it immediately precedes a return construct and it only mentions the returned value (through its different aliases). This ensures the memory is not deallocated before it is returned to the caller.

19–24 This *Shape* decoration is similar to the preceding one (lines 14–16) with additional mention of `_ret`.

26–29 This *Shape* decoration is similar to the preceding one (lines 19–24) but removes mentions of `h`. This is because `h` has fallen out of scope at line 25.

31,32 This is the exit point of the function where both branches join together. The *Shape* decoration is the result of merging the decorations at the end of the two branches. As a result, it maps some pairs of zones to $\top$ (rendered as `_ret.?=l.Cons.Head` where `?=` indicates potential but uncertain aliasing).

```
first(l) = <  %    %  _ret. l. ?l.Nil ?l.Cons ?l.Cons.Head>
  match l with
    [ Nil(_) ->
      <_.=_. _.=l.Nil l.Nil=_. l.Nil=l.Nil  %    %  _ret.>
      let x0 = 0 in
      <_.=_. _.=l.Nil l.Nil=_. l.Nil=l.Nil  %    %  _ret.>
      return x0
      <_.=_. _.=l.Nil l.Nil=_. l.Nil=l.Nil
       _ret.=_ret. _ret.=x0. x0.=_ret. x0.=x0.
       %    %  >
    | Cons(x1) ->
      <l.Cons=l.Cons l.Cons=x1. x1.=l.Cons x1.=x1.
      %    %  _ret. l.Cons x1. l.Cons.Head x1.Head>
      match x1 with
        [ {Head=h} ->
          <h.=h. h.=x1.Head l.Cons=l.Cons l.Cons=x1. l.Cons.Head=h.
           l.Cons.Head=x1.Head x1.=l.Cons x1.=x1. x1.Head=h.
           x1.Head=x1.Head
           %    %  _ret. l.Cons.Head x1.Head h.>
          return h
          <_ret.=_ret. _ret.=h. _ret.=l.Cons.Head _ret.=x1.Head
           h.=_ret. h.=h. h.=l.Cons.Head h.=x1.Head l.Cons=l.Cons
           l.Cons=x1. l.Cons.Head=_ret. l.Cons.Head=h.
           l.Cons.Head=l.Cons.Head l.Cons.Head=x1.Head x1.=l.Cons
           x1.=x1. x1.Head=_ret. x1.Head=h. x1.Head=l.Cons.Head
           x1.Head=x1.Head  %    %  >
        ]
        <_ret.=_ret. _ret.=l.Cons.Head _ret.=x1.Head l.Cons=l.Cons
         l.Cons=x1. l.Cons.Head=_ret. l.Cons.Head=l.Cons.Head
         l.Cons.Head=x1.Head x1.=l.Cons x1.=x1. x1.Head=_ret.
         x1.Head=l.Cons.Head x1.Head=x1.Head  %    %  >
    ]
    <_ret.=_ret. _ret.?=l.Cons.Head l.Cons=l.Cons l.Cons.Head?=_ret.
     l.Cons.Head=l.Cons.Head l.Nil=l.Nil %    %  >
```

Figure 7.2: Analysed program (annotations are curated for readability)

```
(*type definitions*)
type cons = { Head:word ; Tail:list }
and nil = { }
and list =
  | Cons of cons
  | Nil of nil

(*a function definition*)
letrec first (l:list) : word =
  match l with
  [ Nil _ -> 0
  | Cons { Head = h } -> h
  ]
;;
```

Figure 7.3: Source program (with front-end syntax)

7.4 Scalability

As explained above, the performance of the prototype are poor: the implementation has not been optimised for performance, nor has there been any effort towards improving its efficiency. Consequently, analysing performance in terms of time of execution would conflate the poor performance of the prototype's library with the performance of the algorithm in general.

We present below relative measurements of time of execution for two categories of programs. The results are normalised for the average execution time of the smallest program of their category.

7.4.1 Flat programs

We programmatically generated programs as presented in Figure 7.4. We call this category of programs *flat* because the depth of the call graph is constant: the main function calls each of the other functions once.

The performance of each of the three analyses, normalised for flat programs where $n = 1$ is presented in the tables of Figure 7.5. Specifically, the execution time of all three analyses (Shape, Share, and Access) was measured 75 times for different sizes of program. For each set of measurements, the average, minimum, maximum, first quartile and third quartile, are normalised to the average for the program of size one.

The execution time of both the Shape and Access analyses appear exponential in the size of flat programs. On the other hand, the Share analysis is not – as shown in Figure 7.6. In this Figure, the result for different size are set on the horizontal axis. For each size of program, a number along with two lines are displayed: the number value as well as its height represent the average measure; the line below the number joins the minimum value to the first quartile; the line above the number joins the third quartile to the maximum value. Because the results are normalised to the average time, they are presented without a vertical scale. Note the particularly high value for the maximum time in size two; this outlier may be due to interference during measurements.

```
type tuple = { Left: word; Right: word }

letrec $f_1$($x_1$1: word, $x_1$2: word): tuple =
  { Left=x12; Right=x11 }
;;

letrec $f_2$($x_2$1: word, $x_2$2: word): tuple =
  { Left=x22; Right=x21 }
;;

(* and so on defining $f_3$, $f_4$, etc. *)

letrec main() : word =
  let $t_0$: tuple = { Left=4; Right=7 } in

  let $t_1$: tuple = f1($t_0$.Left, $t_0$.Right) in
  let $t_2$: tuple = f2($t_1$.Left, $t_1$.Right) in
  (* and so on calling $f_3$, $f_4$, etc. *)
  let $t_n$: tuple = fn($t_{n-1}$.Left, $t_{n-1}$.Right) in

  $t_n$.Left + $t_n$.Right
;;
```

Figure 7.4: Automatically generated flat programs

7.4.2 Deep programs

We programmatically generated programs as presented in Figure 7.7. We call this category of programs *deep* because the depth of the call grows with the program size. Specifically, the main function calls a first function which calls a second which calls a third and so on.

The performance of each of the three analyses, normalised for deep programs where $n = 1$ is presented in the tables of Figure 7.8. Specifically, the execution time of all three analyses (Shape, Share, and Access) was measured 75 times for different sizes of program. For each set of measurements, the average, minimum, maximum, first quartile and third quartile, are normalised to the average for the program of size one.

The execution time of all three analyses scale better for deep programs than flat programs. The results are presented in Figures 7.9, 7.10, and 7.11.

7.5 Precision

We tested the precision of analyses using hand-written sample programs. These programs illustrate simple concepts: branching, aliasing, etc. We comment on the precision of the analyses for these programs.

In all the examples of this Section, the analysis results have been simplified. First, entries in the analysis results that represent the reflexivity of the Shape analysis have been removed. Second, entries that involve temporary variables inserted by the compiler to hold intermediate values have also been removed. In other words, the analysis results are presented as if on the surface syntax

program size	1	2	3	4	5	7
minimum	0.6703	2.790	8.111	18.78	34.86	120.5
1st quartile	0.7450	3.002	8.721	20.10	39.75	122.4
average	1.000	3.493	9.221	21.21	41.88	129.5
3rd quartile	1.191	3.553	9.435	21.64	43.69	134.1
maximum	1.536	5.989	15.12	36.09	55.24	141.2

(a) Shape analysis

program size	1	2	3	4	5	7
minimum	0.7174	1.004	1.463	1.779	1.779	2.898
1st quartile	0.7747	1.033	1.549	1.865	1.865	2.955
average	1.000	1.369	1.744	2.049	2.090	3.144
3rd quartile	1.090	1.406	1.693	2.008	2.073	3.033
maximum	1.578	9.096	3.500	3.500	3.414	4.447

(b) Share analysis

program size	1	2	3	4	5	7
minimum	0.5891	1.588	3.304	6.129	9.820	24.19
1st quartile	0.6372	1.669	3.507	6.544	10.76	24.66
average	1.000	2.062	3.802	7.094	11.25	27.30
3rd quartile	1.046	2.388	3.948	7.075	11.63	26.88
maximum	3.507	4.347	6.596	11.36	14.24	42.67

(c) Access analysis

Figure 7.5: Scalability of ASAP's analyses for flat programs

(which supports nested patterns and other such constructs) even though the actual analysis was run on μL. Third, the result of the analysis on some less interesting program points has been removed. The aim of all these simplifications is to improve readability and reduce noise.

Branching is illustrated in Figure 7.12. In the `main` function, a simple two-value record is created and passed to the `choose` function. The `choose` function returns either of the two fields. This simple case is accurately tracked by ASAP. The "may"-alias relations are explicitly tracked by 3VL – and shown as ?= in the Figure.

More aliasing is illustrated in Figure 7.13. In the `main` function, a simple record is created. This record is passed to the `swap` twice. The function `swap` creates a new record where the fields are swapped. Finally, the twice-swapped record is passed to the `add` function which sums both fields of the record. In this simple program, ASAP tracks aliasing and access accurately.

A more realistic example is given in Figure 7.14. In this program fragment, the function, `tltl` ("tail-tail") descends through a list, two elements at a time. Effectively, `tltl` returns the sub-list starting with the last even-indexed element.

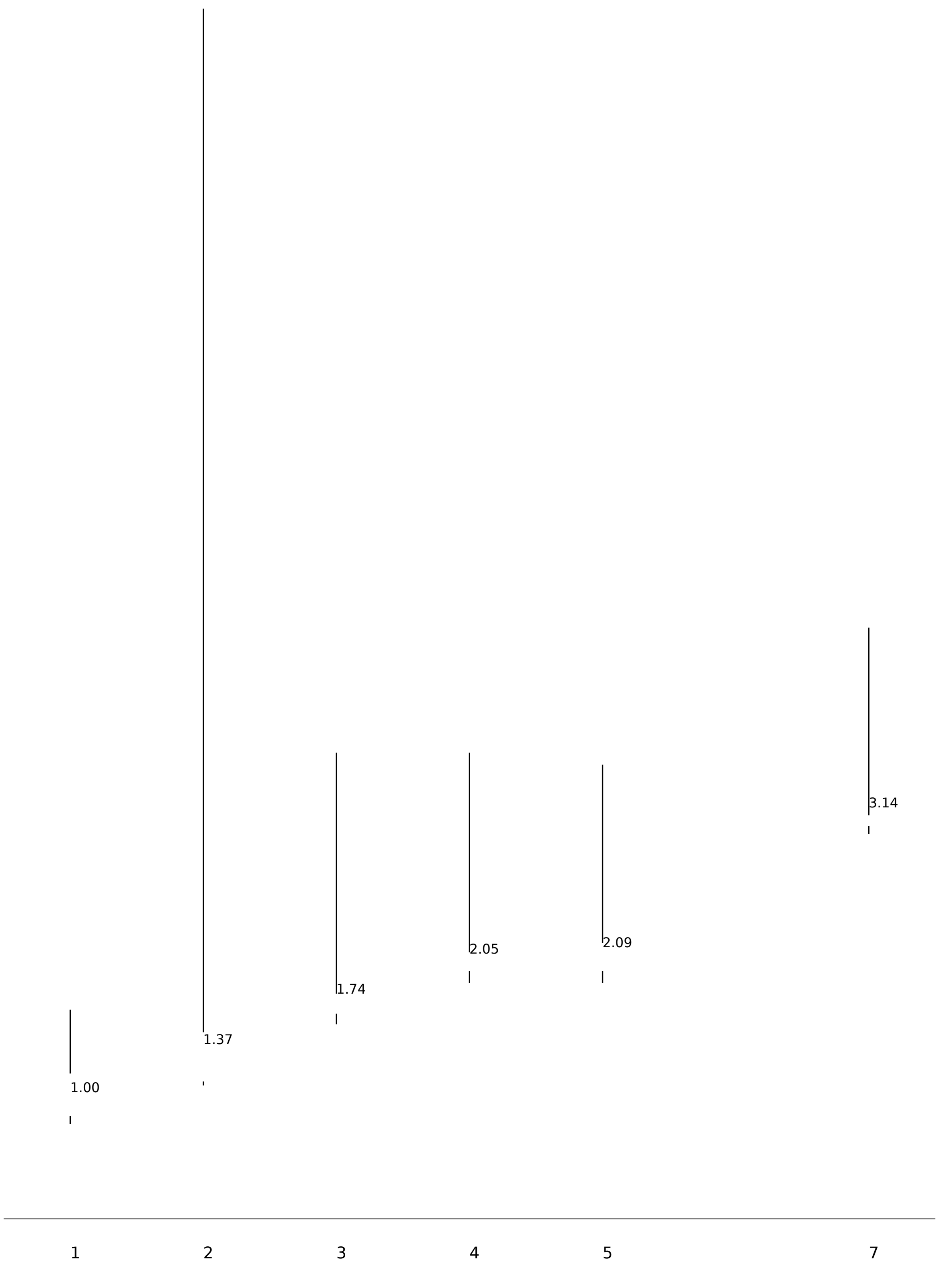


Figure 7.6: Scalability of the Share analysis for flat programs

```
type tuple = { Left: word; Right: word }

letrec f0(x0: word, y0: word): tuple =
  Left = y0; Right = x0
;;

letrec f1(x1: word, y1: word): tuple =
   f0(y1, x1)
;;

letrec f2(x2: word, y2: word): tuple =
   f1(y2, x2)
;;

(*and so on defining f3, f4, etc.*)

letrec main(): word =
   let a = fn(4, 7) in
   a.Left + a.Right
```

Figure 7.7: Automatically generated deep programs

Asap is less precise on this program than on the previous simple examples (Figures 7.13 and 7.12). Specifically, asap ignores the fact that `tltl` descends two elements at a time in the list. Indeed, all repetition operators in the analysis results are for the path `Cell.tl`. In other words, asap detects that `tltl` can descend arbitrarily deep within the list, but ignores the specific way in which `tltl` does so. This loss of information is caused by our definition of the widening operator – it is not a limitation of the prototype.

Also note that the Access analysis includes references to the `Nil` constructor. This is due to the way patterns are compiled: the program actually checks for `Nil` in its μL representation.

Additionally, notice that asap includes unnecessary information in the results of the analysis. E.g., the analyses include both `ll.=l.(Cell.tl)*` and `ll.Cell=l.(Cell.tl)*.Cell` even though the former implies the latter.

program size	1	2	3	4	5	7
minimum	0.9414	1.329	1.664	2.022	2.377	3.184
1st quartile	0.9714	1.345	1.693	2.049	2.441	3.255
average	1.000	1.387	1.747	2.157	2.523	3.347
3rd quartile	0.995	1.371	1.766	2.157	2.562	3.341
maximum	1.567	2.119	2.624	3.249	3.606	5.070

(a) Shape analysis

program size	1	2	3	4	5	7
minimum	0.875	1.039	1.258	1.422	1.586	1.969
1st quartile	0.930	1.094	1.258	1.477	1.641	2.024
average	1.000	1.281	1.391	1.612	1.883	2.240
3rd quartile	0.984	1.258	1.422	1.586	1.750	2.201
maximum	2.024	2.954	2.133	2.735	4.540	3.446

(b) Share analysis

program size	1	2	3	4	5	7
minimum	0.9095	1.087	1.266	1.376	1.589	1.913
1st quartile	0.9168	1.098	1.321	1.415	1.653	1.963
average	1.000	1.209	1.429	1.531	1.745	2.142
3rd quartile	0.9437	1.274	1.410	1.520	1.741	2.130
maximum	1.531	1.848	2.229	2.545	2.697	3.282

(c) Access analysis

Figure 7.8: Scalability of ASAP's analyses for deep programs

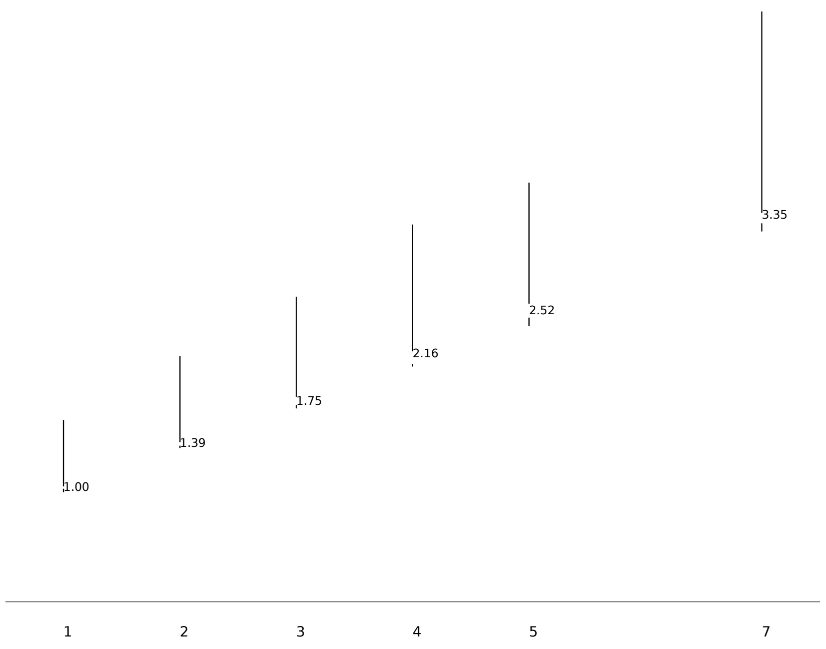


Figure 7.9: Scalability of the Shape analysis for deep programs

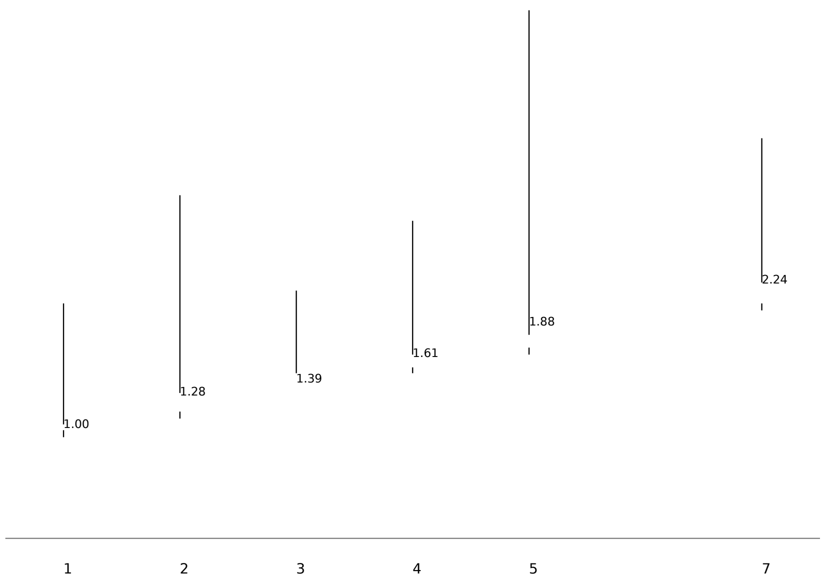


Figure 7.10: Scalability of the Share analysis for deep programs

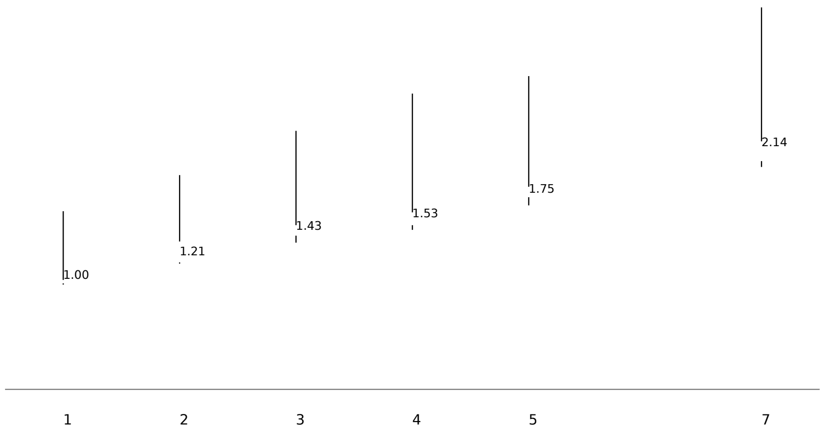


Figure 7.11: Scalability of the Access analysis for deep programs

```
type r = { a:int; b:int }

choose(x:int, r:r) : int =
   < % % _ret. r. ?r.a ?r.b>
   match (x<10) with
   [ 1 ->
     < % % _ret. r. r.a>
     match r with
     [ {a=a} ->
       <a.=r.a % % _ret. a. r.a>
       return a
       <_ret.=a. _ret.=r.a a.=r.a % % >
     ]
     <_ret.=r.a % % >
   | 0 ->
     < % % _ret. r. r.b>
     match r with
     [ {b=b} ->
       <b.=r.b % % _ret. b. r.b>
       return b
       <_ret.=b. _ret.=r.b b.=r.b % % >
     ]
     <_ret.=r.b % % >
   ]
   <_ret.?=r.a _ret.?=r.b % % >

main() =
   let v = {a=0;b=1} in
   <% % _ret. ?v.a ?v.b v.>
   let vv = choose(9, v) in
   <v.a?=vv. v.b?=vv. % % _ret. ?v.a ?v.b vv.>
   return vv
   <_ret.?=v.a _ret.?=v.b _ret.=vv. v.a?=vv. v.b?=vv. % % >
```

Figure 7.12: Precision on branching programs

```
type r = { a: int; b: int }

swap(r) =
   match r with
   [ { a=a; b=b } ->
     <a.=r.a b.=r.b % % _ret. r.a r.b a. b.>
     let x2 = { a=b; b=a } in
     <a.=r.a a.=x2.b b.=r.b b.=x2.a r.a=x2.b r.b=x2.a % % _ret. x2.>
     return x2
     <_ret.=x2. _ret.a=b. _ret.a=r.b _ret.a=x2.a
      _ret.b=a. _ret.b=r.a _ret.b=x2.b
      a.=r.a a.=x2.b b.=r.b b.=x2.a r.a=x2.b r.b=x2.a % %>
   ]
   <_ret.a=r.b _ret.b=r.a % %>

add(r) =
   match r with
      [ { a=a; b=b } ->
         <a.=r.a b.=r.b % % _ret. r.a r.b a. b.>
         return (a + b)
      ]

main() =
   let r = { a=100; b=101 } in
   <% % _ret. r.a r.b>
   let u = swap(r) in
   <r.a=u.b r.b=u.a % % _ret. r.a r.b u. u.a u.b>
   let v = swap(u) in
   <r.a=u.b r.b=u.a r.a=v.a r.b=v.b u.a=v.b u.b=v.a % %
    _ret. r.a r.b u.a u.b v. v.a v.b>
   let x6 = add(v) in
   <r.a=u.b r.b=u.a r.a=v.a r.b=v.b u.a=v.b u.b=v.a % % _ret. x6.>
   return x6
```

Figure 7.13: Precision on an aliasing program

```
type cell = { hd: int; tl: list }
and nil = { }
and list =
    | Cell of cell
    | Nil of nil

tltl(l) =
    match l with
    [ Cell { tl = Cell { tl = r } } ->
        <r.=l.Cell.tl.Cell.tl % %
         _ret. r.(Cell.tl)* r.(Cell.tl)*.Cell
         r.(Cell.tl)*.Nil l.Cell.tl.Cell.tl.(Cell.tl)*
         l.Cell.tl.Cell.tl.(Cell.tl)*.Cell
         l.Cell.tl.Cell.tl.(Cell.tl)*.Nil.>
        let ll = tltl(r) in
        <ll.=l.Cell.tl.Cell.tl.(Cell.tl)*
         ll.Cell=l.Cell.tl.Cell.tl.(Cell.tl)*.Cell
         r.=l.Cell.tl.Cell.tl
         ll.=r.(Cell.tl)* ll.Cell=r.(Cell.tl)*.Cell % %
         _ret. ll. l.Cell.tl.Cell.tl.(Cell.tl)* r.(Cell.tl)* >
        return ll
        <_ret.=ll. _ret.Cell=ll.Cell _ret.Cell.tl=ll.Cell.tl
         _ret.=r.(Cell.tl)* _ret.Cell=r.(Cell.tl)*.Cell
         _ret.=l.(Cell.tl)* _ret.Cell=l.(Cell.tl)*.Cell
         ll.=r.(Cell.tl)* ll.Cell=r.(Cell.tl)*.Cell
         ll.=l.(Cell.tl)* ll.Cell=l.(Cell.tl)*.Cell % %>
    | _ ->
        <% % _ret. l.>
        return l
        <_ret.=l. % %>
    ]
    <_ret.=l.(Cell.tl)* _ret.Cell=l.(Cell.tl)*.Cell % %>
```

Figure 7.14: Precision on a recursing program

Chapter 8

ASAP in relation to other strategies

We now take a look at ASAP in relation with the other approaches listed in Chapter 3. We focus on fitness for system programming.

8.1 Comparison with existing strategies

Here we compare ASAP to other approaches. We also explore ways in which to combine ASAP with these other approaches.

8.1.1 Manual memory management *à la* C

ASAP and the manual approach of C have little in common from the programmer's perspective: ASAP abstracts the memory entirely from the programmer whilst C leaves the memory explicit. Despite this major difference, they share some common ground. Specifically, both strategies provide control over the memory representation of values. This is achieved, in both cases, by avoiding to rely on the runtime.

Note that ASAP does not let the programmer choose the deallocation points (unlike C). Nevertheless, the programmer can, after compilation, inspect the emitted code to see what deallocation points have been chosen by ASAP. For ease of reading, ASAP can be made to dump the intermediate representation with calls to $CLEAN$ before their expansion into scanning code. This facility can help predict behaviour during execution.

Where C outshines ASAP is in the ability it gives programmer to optimise memory management. Specifically, the programmer can decide to wait until all known aliases of a value are dead before deallocating it. By contrast, ASAP will generate scanning code that is run during execution. Where ASAP outshines C is safety: C has none. This is important for system programs because they underpin all other software.

We posit it is possible to use ASAP to check human-specified deallocations are correct. In such an approach, the programmer has access to the $CLEAN$ compile-time function and decides when to deallocate what. Moreover, unlike with `free`, the programmer can specify a matter-set: values to keep. The role of ASAP is then reduced to checking correctness and expanding calls to $CLEAN$. To ensure a program is leak-free, ASAP must enforce that all zones are deallocated before they fall out of scope. Additionally, to ensure a program does not contain use-after-free, ASAP must enforce several properties. First, only zones with $Access\ 0$ ever appear in the anti-matter set. Second, when a zone z is deallocated at point $\textcircled{\varpi}$, any zone z' such that $Shape(\textcircled{\varpi})(z, z') \geq \top$ must appear in either the matter-set or the anti-matter set. This last point ensures that overlap between zones is never overlooked. Additional properties might be necessary; further investigation is left as future work.

Note that integrating ASAP with C specifically is more complicated than just checking for the properties above. Indeed, C's type system is difficult to translate into µL's. Additionally, ASAP assumes that programs are well-typed which is not guaranteed, even in the middle-end of C compilers. However, the use of compiler-checked $CLEAN$ could be applied to a different C-like language, providing a sound basis for a system-programming-friendly memory management.

8.1.2 GC

ASAP is suited as a drop-in replacement for GCs because, from the point of view of the programmer, they are similar: memory management is abstracted away in a safe and correct way. However, the execution of programs under ASAP differs significantly: GCs batch memory management operations in big and asynchronous collection cycles whilst ASAP performs small and synchronous sets of operations. As a result, the relative efficiency of these two strategies can vary with different program patterns. In particular, consider programs that are short lived or do not use a lot of memory: under a GC the program might exit before a collection cycle ever happens, in which case memory deallocation uses absolutely no computing resources. (Note however, that allocations are different under GCed and non-GCed languages. Thus, some resources might be used during allocations to initialise GC marking fields.)

One of the way to characterise ASAP is as an inlined GC which is statically optimised. Specifically, ASAP zealously runs GC-like cycles at every program point but only from a limited set of roots and along a pruned set of paths. The set of roots and paths form zones which are chosen statically based on information inferred by ASAP's analyses. The zone specialisation is such that, at some program points, the "inlined GC call" is removed altogether. However, this characterisation is imprecise in three major ways. First, the timeliness of the two approaches differs: the approximation of waste by $Access$ is more precise than the characterisation by unreachability of GCs. Second, the collection points are disjoint: in ASAP collections appear after each binding and match[1] instead of allocation points. Third, in ASAP, the scanning operations do not rely on runtime type information (such as block size and pointerness).

Another important difference with GCs is the relation with the underlying OS. Indeed, runtimes that include GCs request a chunk of memory from the OS and manage it for the program. When the programmer allocates a value (either explicitly such as with Java's `new` or implicitly as with constructors in ML), the runtime reserves some of the pre-allocated chunk for that value. By contrast, under ASAP, the memory can be reserved and released using `malloc` and `free`, just like in C. This latter style also reserves memory, albeit in smaller chunks. Under ASAP, the memory management primitives can be specialised to support either scheme.

8.1.3 ASAP-GC hybrid as a liveness-assisted GC

An interesting hybrid is to use ASAP for marking with a more standard GC-like sweeper. Compared to ASAP, this hybrid requires runtime type information – unless a tagless GC is used for sweeping. Compared to GC, the hybrid approach is timelier: waste is approximated by $Access$ (essentially liveness) which is more precise than reachability. In this respect, the hybrid behaves like a liveness-assisted GC.

The ASAP-GC hybrid has timeliness similar to LAGC's (because both use a similar waste approximation) but they differ in other aspects. The main difference is that LAGCs characterise pointers as dead and prevent the GC from following them (either by setting them to null or by communicating a path set to the GC). ASAP, on the other hand, characterises memory blocks as dead and synthesises code that, in the hybrid presented above, marks them as garbage. As a result, LAGC

[1]Remember that when it is statically decidable that a specific collection will free no memory at all, it is removed altogether.

adds null pointers in the memory of the program (when it knows this pointer will not be followed) whilst ASAP deallocates blocks which creates dangling pointers.

8.1.4 RC

Reference counting, like ASAP, is a safe, synchronous memory management strategy. However, RC approximates waste by unreachability which is less timely than ASAP's approximation by $Access$.

Rc, like ASAP, is a synchronous strategy: deallocation points are known, and thus static information (such as type) is available. Thus, the cascading deallocation code can be generated at compile time. Specifically, a $SCAN$-like compile-time function can be used. This $SCAN$-like function integrates tests to check the value of the reference counters.

Using a $SCAN$-like function gives opportunity for optimisations. Specifically, if the compiler is able to statically determine that a reference counter, or a set thereof, is 0, it can specialise the generated code to perform unconditional deallocations.

8.1.5 Re-use

Re-use systems, such as Mercury's CTGC, reduce the workload of a memory management strategies by combining deallocation-allocation sequences into simple mutations.

There are two ways to combine ASAP with a re-use system. The first possibility is to start with the re-use transformation and follow up with ASAP. In this situation, ASAP receives an optimised program in which some blocks are re-used. Note that, in the optimised program, some values get mutated. As was noted in Chapter 6, mutation can be handled by ASAP with the following caveats. At the program point where the mutation happens, a call to $CLEAN$ is always inserted. This call might be optimised away, depending on the statically inferred information. Mutation of non-local values (specifically, of arguments that are used by the caller after return) incurs runtime costs. Thus we posit that limiting re-use to local values is preferable.

The second possibility is to start with ASAP and follow up with the re-use transformation. In this situation, the re-use system's work is simplified: it can see explicit deallocations instead of having to guess non-liveness of values. Thus, the re-use system merely needs to find pairs of deallocations and allocations close together and transforms them into a mutation. This approach is simpler: ASAP is not involved, and the re-use system only detects deallocation/construction pairs. Note however, that only unconditional deallocations are ever considered for re-use.

8.1.6 Regions

Region systems share many features with ASAP such as synchrony, agnosticism to memory representation, and correctness. Comparing the means by which regions and ASAP provide these features is interesting. The following comparison assumes Tofte-Talpin style of regions [39] inferred by the compiler.

To infer regions, the compiler first assigns a region variable to each program variable. Then, the compiler emits constraints on region variables based on instructions of the program. E.g., if a variable located in region ρ_1 is stored inside a list located in region ρ_2, then ρ_1 must be an outer region and ρ_2 an inner region – or, in other words, ρ_1 must be deallocated after ρ_2. These constraints form a partial order over the region variables. Finally, the compiler solves these constraints to find which region variables are associated to the same region. This is achieved by propagating the partial order by transitivity and detecting equality by antisymmetry. The result is a coarse description of the heap: which value is stored in which region and in what directions are inter-region pointers allowed.

This bears resemblance with the way ASAP infers information about the heap. Indeed, ASAP too collects constraints based on the program instructions. These constraints are also based on instructions that affect points-to relationship between values in the program. Unlike regions however, ASAP solves these constraints by inferring fine-grain information about the shape of the heap: in what way may a value point to another value.

This leads to an interpretation of regions as a lossy simplification of heap shape analysis. With regions, all that remains of the description of the heap is a specific partitioning with a partial order of the partitions. Because region inference sets out to produce this coarse description, it is simpler than ASAP's Shape and Share analyses. On the other hand, region systems describe the heap less precisely and are consequently less timely than ASAP.

8.1.7 Necessity analysis

In [18], G. W. Hamilton and Simon B. Jones describe necessity based garbage collection (NBGC): an analysis and subsequent code transformation to assist with memory management. Specifically, a necessity analysis is performed in order to decide, for each program point, what part of each value is needed later. Equipped with this information, the code is transformed to deallocate unneeded values earlier than a GC would allow. Additionally, when possible, the code is transformed so that would-be deallocated cells are re-used.

Thus, NBGC is similar to the working of ASAP. Specifically, NBGC describes the necessity of a value by a tree. These trees follow the grammar $P ::= 0|1|(\{0|1\} \times P \times P)$ where 0 represents a value that is not needed, 1 represents a value that may be needed, and (r, p_h, p_t) represent a list with r the necessity of the root cons cell, p_h the necessity of the head, and p_t the necessity of the tail.

There are two important differences between ASAP and NBGC. First, necessity trees are less precise than paths. Indeed, it is impossible to use trees to express such statements as "for all the elements of a list l, the *Left* fields are needed, but the *Right* fields are not." This is impossible because of the absence of the repetition operator in trees. The statement is formalised in paths using the following equations.

$$\begin{array}{rcl} \mathit{Access}(l, \mathit{elems} \cdot \mathit{Left}) & = & 1 \\ \mathit{Access}(l, \mathit{elems} \cdot \mathit{Right}) & = & 0 \end{array} \text{ where } \mathit{elems} = (\mathit{Cons} \cdot \mathit{Tail})^* \cdot \mathit{Cons} \cdot \mathit{Head}$$

Second, when the necessity of a value cannot be decided, NBGC falls back to a standard GC. That is NBGC assists the GC by introducing re-use and early deallocations. By contrast, when the Access property of a value cannot be decided, ASAP emits code that determines the safety of the deallocation during execution. That is ASAP replaces the GC by emitting complete deallocation code.

8.1.8 Individual object deallocation

In [11], Cherem and Rugina present a method for compile-time deallocation of individual objects. Their approach relies on an analysis and a code transformation. The analysis statically determines the number of references to each individual objects. To avoid the analysis diverging, the count is bounded to a constant k, after which the value $\inf$ approximates the reference count. When a reference count falls to 0, the code transformation inserts a `free` statement. This statement may change the reference count of other objects; this is dealt with by fixpointing the analysis and the code transformation until no more `free` statements are inserted.

Experimental results from the authors indicate that the fixpoint is generally reached within three iterations. They also show a execution time low overhead. Finally, an interesting measure is that, depending on the program, the approach frees between 0% and 90% of objects. To accommodate for the remainder of the objects, the authors propose to complement their approach with a GC.

Thus, this approach differs from ASAP in multiple ways. First, the aim differs in that the approach does not intend to be complete: it lightens the workload of the GC rather than replacing it. Second, the means differ in that the approach uses analyses that statically approximate reference counting instead of approximating the heap shape. The means also differ in that the approach generates simple free instructions for individual objects instead of deallocation code that descends through a data structure.

8.2 Comparison with Rust and C

We use the same table as in Chapter 3 to compare ASAP with two languages that are used in system programming: Rust and C. The result is detailed in Figure 8.1. Note that ASAP offers features similar to C's with the addition of safety.

Strategy	Agnostic(👍)	Correct(👍)	Restrictive(👎)	Synchronous	Approximative	Re-use(👍)
À la C	✓			✓	N/A	✓(a)
Rust	✓	✓	✓	✓	✓(b)	
ASAP	✓	✓	(c)	✓	✓(d)	

(a) manual re-use
(b) liveness (aided by ownership annotations)
(c) assumes the program is type correct
(d) liveness (inferred by compiler)

Figure 8.1: Asap and other system-programming-friendly strategies

Chapter 9

Linear and Region regimes

We now study the behaviour under ASAP of programs that adhere to either the linear or region regimes. More precisely, we look at the result of ASAP's analyses and transformation applied to linear or region-based programs. To some extent (detailed below), ASAP subsumes both regimes: it generates deallocation instructions that are similar to the ones generated by the regimes' own memory managers. This is summarised in Figure 9.1. We detail this subsumption, including caveats, for linear programs in Section 9.1. Regions are given the same scrutiny in Section 9.2.

The purpose of this chapter is to study the handling of programs under linear or regional restrictions. When we talk about ASAP subsuming linear types or regions, we only compare the memory management aspects.

9.1 Linear types

We now focus on linearly typed programs. For that purpose, we assume the front-end is equipped with a linear type checker. As explained below, ASAP subsumes the memory management part of linear types at the procedure level. Additionally, we detail a simplification of ASAP's inter-procedural analyses that makes the subsumption complete: deallocations become exactly identical

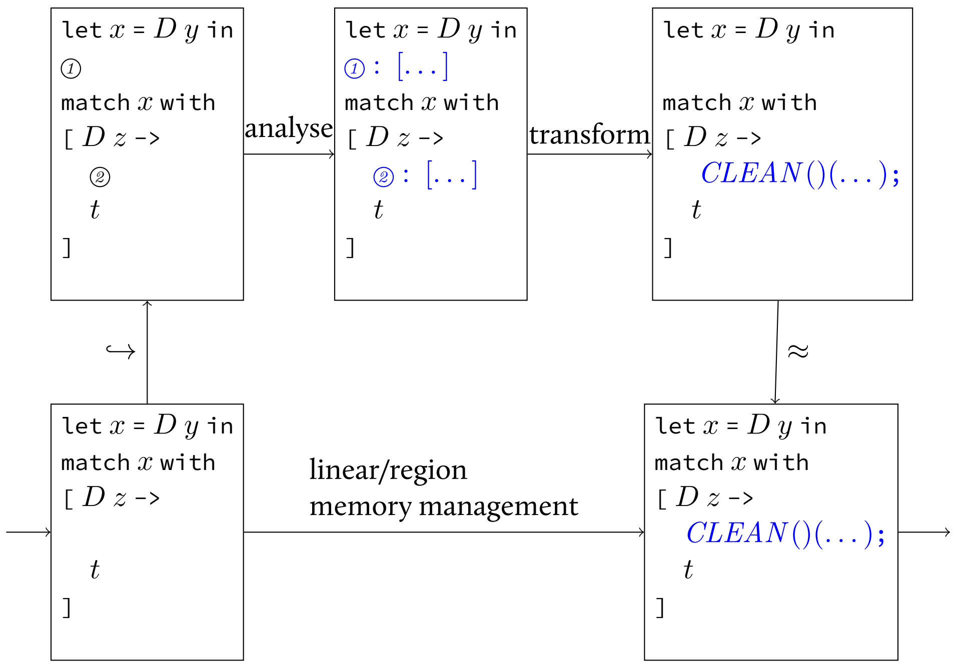


Figure 9.1: ASAP subsumes linear and region memory management

to the linear case. We first discuss the simplifications to the inter-procedural analyses in Section 9.1.1 and 9.1.2, and then give an example of linear program in Section 9.1.3. Finally, we explain how the example generalises to any linear programs in Section 9.1.4.

9.1.1 ASAP's analyses: too precise for their own good

First, note that the $Share$ analysis is not needed: internal sharing is entirely prevented by linear types – as detailed by Wadler [41]. It is possible to run the $Share$ analysis, but the results are trivial: $\forall\textcircled{\pi}, \forall z, Share(\textcircled{\pi})(z) = 0$.

Second, note that the amalgamated call context for the $Shape$ analysis always has the same form. Indeed, whilst it is possible to create some aliasing in a linear program (as discussed in Chapter 3), it is impossible to use both aliases for a value. More specifically, linear types enforce that only the most recent name for a memory block is used. As a result, in a function call, distinct arguments cannot alias – because one would be the older of the two and thus unusable. Consequently, we know in advance that the call context for the $Shape$ of a function f with argument $x_1, \ldots, x_n$ is always

$$f^{\uparrow}_{Shape} = [((x_i, \mathit{Wild}(\Gamma(x_i))), (x_j, \mathit{Wild}(\Gamma(x_i)))) \mapsto 0 \mid i, j \leq n]$$

Third, remember that in linearly typed programs, the responsibility for managing the memory of arguments of a call falls on the callee. However, this is not always picked up by ASAP because of aliasing. Indeed, as discussed in Chapter 3, despite the common misconception aliasing occurs in linear programs. All that linear type constraints guarantee is that aliasing cannot be observed by the programmer. Unfortunately, ASAP detects this aliasing as shown in Figure 9.2: the $Shape$ summary of the function $either$ is

$$either^{\downarrow} = [((x, F_a), (ret, \epsilon)) \mapsto \top, ((x, F_b), (ret, \epsilon)) \mapsto \top]$$

Remember that, under linear constraints, the caller of a function must use the return value. As a result, at a call point `let` y `=` $either$`(`x`)` `in` ${}^{\textcircled{\pi}}t$ we have

$$\begin{array}{rcl} Access(\textcircled{\pi})((y, \mathit{Wild}(\Gamma(y))) & = & 1 \\ Access(\textcircled{\pi})((x, F_a \cdot \mathit{Wild}(\Gamma(x).F_a))) & = & \top \\ Access(\textcircled{\pi})((x, F_b \cdot \mathit{Wild}(\Gamma(x).F_b))) & = & \top \end{array}$$

where the second and third lines indicate implied accesses. Consequently, the $Access$ amalgamated call context for $either$ mentions (x, F_a) and (x, F_b). This amalgamated call context prevents $either$ from deallocating the values passed to $ignore$.

9.1.2 Simplifications to the inter-procedural analysis

A small modification of ASAP recovers the linear style of deallocations: the inter-procedural component of the $Shape$ analysis is fixed. Specifically, the $Shape$ summary for any function f with parameters $x_1, \ldots, x_n$ is

$$f^{\downarrow} = \left[\begin{array}{l} ((x_i, \mathit{Wild}(\Gamma(x_i))), (ret, \mathit{Wild}(\Gamma(ret)))) \mapsto 0, \\ ((x_i, \mathit{Wild}(\Gamma(x_i))), (x_j, \mathit{Wild}(\Gamma(x_j)))) \mapsto 0 \end{array} \middle| \, i, j \leq n \right]$$

regardless of the body of f. This fixed summary informs callers that arguments do not alias with the return value nor with each other – which is false but safe under linear types.

```
either(x) =
    match x with
    [ {F_a = a; F_b = b} ->
        if (...) then
            ignore(b);
            return a
        else
            ignore(a);
            return b
    ]
```

Figure 9.2: Aliasing between return value and argument in a linear function

This simplification has major impacts on the rest of the inter-procedural analysis. Specifically, with the *Shape* summaries fixed, the summaries and call contexts for the *Access* analysis become entirely predictable[1]:

$$\begin{aligned} f^{\downarrow} &= [(x_i,\, \mathit{Wild}(\Gamma(x_i))) \mapsto 1] \\ f^{\uparrow} &= [(x_i,\, \mathit{Wild}(\Gamma(x_i))) \mapsto 0, (\mathit{ret},\, \mathit{Wild}(\Gamma(\mathit{ret}))) \mapsto 1] \end{aligned}$$

where the summary indicates that, during calls, arguments are accessed by f and the call context indicates that, after the call, the returned value is used but the arguments are not). As a result, the responsibility for the memory management of the argument is always shifted to the caller.

9.1.3 Example

Consider the example in Figure 9.3. It displays a function definition (9.3a), some interesting analysis results (9.3b) and the added calls to $CLEAN$ (9.3c). We use syntactic sugar in this example: `if-then-else`, omitted type annotations, sequences. We also use a tuple to bind both values returned by the special construct *deepCopy*[2].

The function f receives a parameter x. It destructs it to obtain the content of each of its two fields F_a and F_b. It returns a similar record with fields swapped or, depending on an unspecified condition, both fields identical.

The analysis results are abridged for readability. Specifically, not all program points are displayed, the symmetric and transitive closure of the *Shape* property is ignored, and only the elements of the *Access* property that lead to deallocations are shown.

The transformation of f inserts a single call to $CLEAN$ just after the match. Remember that the *ignore* function is used to satisfy linear constraints. Specifically, it is called to avoid a value (here b) not being used. Removing the calls to *ignore* from linear programs is not an issue for ASAP. Indeed, without *ignore*, the transformation step would simply insert an equivalent call to $CLEAN$. This follows from the analysis: $\mathit{Access}(②)((b,\, \mathit{Wild}(\Gamma(b)))) = 0$.

The most interesting point about this example is that the deallocation instructions inserted by ASAP are the exact same as with a linear manager. Specifically, at the destruction operator (`match`), a simple deallocation instruction is generated. Indeed, note how the call to $CLEAN$ has

[1]It is still possible to run the *Access* inter-procedural analysis, but it is unnecessary.

[2]With linear types, arguments are consumed by calls. Thus, the form `let` x' = *deepCopy*(x), where x becomes unavailable, is useless. Instead, under linear typing, copying is achieved through the form `let`(x', x'') = *deepCopy*(x).

```
f(x) = ⓪
    match x with
    [ {F_a = a; F_b = b} ->
        ①if (...) then
            ignore(b);
            ②let (c,d) = deepCopy(a) in
            ③let r = {F_a = c; F_b = d} in
            ④return r⑤
        else
            let r′ = {F_a = b; F_b = a} in
            ⑥return r′⑦
    ]
```

(a) Example of function definition with program points

$\textcircled{\pi}$	*Shape*	*Access*
⓪		$[(x,\ \mathit{Wild}(\Gamma(x))) \mapsto 1]$
①	$[((x, F_a), (a, \epsilon)), ((x, F_b), (b, \epsilon)) \mapsto 1]$	$[(x, \epsilon) \mapsto 0, (a,\ \mathit{Wild}(\Gamma(a))), (b,\ \mathit{Wild}(\Gamma(b))) \mapsto 1]$
②		$[(b,\ \mathit{Wild}(\Gamma(b))) \mapsto 0, (a,\ \mathit{Wild}(\Gamma(a))) \mapsto 1]$
③		$[(a,\ \mathit{Wild}(\Gamma(a))) \mapsto 0]$
④	$[((r, F_a), (c, \epsilon)), ((r, F_b), (d, \epsilon)) \mapsto 1]$	
⑤	$[((r, \epsilon), (ret, \epsilon)) \mapsto 1]$	
⑥	$[((r', F_a), (b, \epsilon)), ((r', F_b), (a, \epsilon)) \mapsto 1]$	
⑦	$[((r', \epsilon), (ret, \epsilon)) \mapsto 1]$	

(b) Excerpt (only the interesting parts are shown) of the analysis results

```
f(x) =
    match x with
    [ {F_a = a; F_b = b} ->
        ~(CLEAN()((x,ε)));
        if (...) then
            ignore(b);                              (*ignore handles b*)
            let (c,d) = deepCopy(a) in            (*deepCopy handles a*)
            let r = {F_a = c; F_b = d} in
            return r
        else
            let r′ = {F_a = b; F_b = a} in
            return r′
    ]
```

(c) Result of ASAP's transformation on f

Figure 9.3: Linear μL program: before and after ASAP

an empty matter-set (i.e., no scanning for live values) and a trivial path (ϵ) (i.e., no exploring of the dead value). This call expands to a single call to *free*.

9.1.4 Generalisation to all linear programs

Just as with the example above, ASAP subsumes linear memory management for every linear program. That is, given any linear program, ASAP will generate the same deallocation instructions as the linear type system would.

We consider the different forms terms can take and explain how the linear constraints affect their analysis and transformation by ASAP.

Bindings Bindings, of the form ^{①} `let` x `=` e `in` ^{②} t never trigger deallocations. We first consider the cases where e is either a literal or an arithmetic or logic operator application. In all of these cases, all the values involved in the operation are of the $word$ type which are never deallocated by ASAP.

We then consider the case where e is a function call: f`(...)`. Remember that the summaries and call contexts for every function of a functional program are known (Section 9.1.2). Specifically, the $Access$ amalgamated call context is 0 for the whole of the parameters' memory. Also remember that the anti-matter set at a call point is $A = \{z \mid Access(①)(z) \geq \top, Access(②)(z) = 0, \theta^{\vec{a}}_{\vec{p}}(f^{\uparrow})(z) \neq 0\}$ (Chapter 5). As a result, the anti-matter set is empty at each function call.

We finally consider all the other cases for e: a variable (y), a sum constructor ($D\ y$), or a record constructor (`{`F_1 `=` y_1`; ...}`). In these cases, the binding introduces aliasing between x and y (or all the y_i). This aliasing affects the $Close$ helper function of the $Access$ analysis – to record indirect accesses. Thus, the aliasing combined with the guarantee (under linear constraints) that the value x is accessed later, ensures the $Access$ property for the value y (or the y_i) is 1.

Destructors Destructors, of the form ^{①}`match`x`with[...|`p`->`^{②}t`|...]` deallocate the matched value (but only the top-level block thereof). Indeed, linear constraints guarantee that the matched value (x) will never be used again but that the values extracted from it will be. As a result, $Access(②)((x, \epsilon)) = 0$ but $Access(②)((x, p)) = 1$ ($\forall p \neq \epsilon$). The optimisation of the matter and anti-matter sets (detailed in Chapter 5) then simplifies the call to $CLEAN()((x, \epsilon))$.

Return As explained in Chapter 5 for general programs, terms of the form `return` x are not affected by ASAP. Linear programs are a special case of programs in general; thus, their return constructs are also unaffected.

9.1.5 Weaker linear variants

With affine types, values can be used at most once. This is different from linear types in which values must be used exactly once. Specifically, affine types do not require the programmer to insert calls to $ignore$. As noted above, without calls to $ignore$, ASAP introduces calls to $CLEAN$ directly. Thus affine types are also subsumed by ASAP

With quasi-linear types, the programmer can mix linear and non-linear values. Specifically, some values are given a linear mode (written 1) and others a non-linear mode (written δ). The constraints of quasi-linear types allow δ-values to appear in multiple expressions but prevents them from being part of the computed result. Thus, δ-values can be observed multiple times, but they cannot be aliased. Thus the $Shape$ property of δ-values attests that they are isolated.

Although we have not studied the matter thoroughly, we believe that ASAP also subsumes quasi-linear types.

9.2 Regions

We now consider how ASAP handles region-based programs. This is not as straight-forward as with linear types because regions introduce additional syntax for annotations. Depending on the exact region system, the annotations are either written by the programmer or inferred by the compiler. Regardless of their origin, the annotations must be handled by ASAP.

Instead of extending ASAP for these new constructs, we decide to compile them into µL directly. This is achieved through the use of the reserved function identifiers at and $tickle$ which are no-op at runtime but have non-trivial summaries capturing the meaning of the region annotations. The summaries of the functions at and $tickle$ capture the essence of region annotations; they artificially link values and regions together and extend the lifetime of regions to the end of scope – more details below.

9.2.1 µL+ρ

We define µL+ρ by extending µL with the following constructs. The meta-variable ρ ranges over identifiers of the $regionname$ syntactic category. Regions are introduced like so: `region` ρ `in` t. Bindings are affixed with a region name indicating what region the value is allocated in, like so: `let` x `:` α `@` ρ `=` e `in` t. The region name annotation is omitted for values of the $word$ because they are not allocated on the heap. Finally, function definitions are annotated like so: f`(`x `:` α `@` ρ`, ...)` `:` α' `@` ρ' `=` t to indicate the region each parameter belongs to and the region the return value should be allocated in.

9.2.2 Compiling µL+ρ into µL

The region constructs listed above are compiled away as detailed in Figure 9.4. The result of this compilation uses some function calls that, at execution-time are no-op, but have non-trivial summaries for the $Shape$ and $Access$ properties. These no-op functions capture the region annotations into ASAP by mapping their meaning into function calls. First, at`(`x`,` ρ`)` indicates to ASAP that the variable x belongs to the region ρ. Second, $tickle$`(`ρ`)` indicates that the lifetime of the region ρ extends to the call – detailed below. This is achieved by the following summaries:

$$\begin{array}{lrcl} \text{for } Shape & at^{\downarrow}(x,\rho) & = & [((x,\, Wild(\Gamma(x))),(\rho,\epsilon)) \mapsto 1] \\ \text{for } Access & tickle^{\downarrow}(\rho) & = & [(\rho,\epsilon)\mapsto 1] \end{array}$$

9.2.3 Example

We consider how ASAP handles of programs that have been compiled as presented above. We start with an example in Figure 9.5. The deallocations introduced by ASAP are similar to the ones under region management. The correspondence is not always exact as detailed below.

In this example, the function $makeSummary$ gets a sample of raw data and summarises it. The raw data (d_1) is allocated in a local region ($[\![\rho_1]\!]_\rho$). The summary (d_0, and its wrapper s) is allocated in the outer region received as parameter ($[\![\rho_0]\!]_\rho$).

The function $tickle$ extends the $Access$ of $[\![\rho_1]\!]_\rho$ all the way to ⑤, at which point a deallocation instruction is inserted.

$$
\begin{array}{rcl}
[\![.]\!]_\rho & : & \textit{regionname} \rightarrow \textit{variable} \\
\\
[\![\texttt{region}\ \rho\ \texttt{in}\ t]\!]_\mathrm{t} & = & \texttt{let}\ [\![\rho]\!]_\rho : \textit{region} = \texttt{\{\}}\ \texttt{in}\ [\![t]\!]_\mathrm{t} \\
[\![\texttt{let}\ x : \alpha\ \texttt{@}\ \rho = f(\ldots)\ \texttt{in}\ t]\!]_\mathrm{t} & = & \texttt{let}\ x : \alpha = f([\![\rho]\!]_\rho,\ \ldots)\ \texttt{in}\ [\![t]\!]_\mathrm{t} \\
[\![\texttt{let}\ x : \alpha\ \texttt{@}\ \rho = e\ \texttt{in}\ t]\!]_\mathrm{t} & = & \texttt{let}\ x : \alpha = e\ \texttt{in}\ \textit{at}(x,\ [\![\rho]\!]_\rho);\ [\![t]\!]_\mathrm{t} \\
[\![\texttt{return}\ x]\!]_\mathrm{t} & = & \textit{tickle}([\![\rho_1]\!]_\rho); \\
 & & \ldots; \\
 & & \textit{tickle}([\![\rho_n]\!]_\rho); \\
 & & \texttt{return}\ x \\
 & & \text{(where } \rho_i \text{ are the regions in scope)} \\
\\
\left[\!\!\left[\begin{array}{l} \texttt{fun}\ f(x : \alpha\ \texttt{@}\ \rho,\ \ldots) \\ \quad : \alpha'\ \texttt{@}\ \rho' \\ \quad = t \end{array}\right]\!\!\right]_\mathrm{f} & = & \begin{array}{l} f([\![\rho]\!]_\rho : \textit{region},\ x : \alpha,\ \ldots) \\ \quad : \alpha' \\ \quad = [\![t]\!]_\mathrm{t} \end{array}
\end{array}
$$

Figure 9.4: Compiling away region annotations

9.2.4 Generalisation to other region-based programs

The behaviour of ASAP shown on the example above applies to all region-based programs. More specifically, given a region program, ASAP will generate deallocations instructions similar to the ones a region system would. However, unlike region-based memory management, the deallocation instructions generated by ASAP might require scanning. Indeed, remember that values from a region ρ can hold pointers to values in an outer region ρ' – but values in ρ' cannot hold pointers to values in the inner region ρ. In other words, values from ρ are represented by memory located in both ρ and ρ' (and possibly outermore regions). The values in the outer region ρ' outlive the values in the inner region ρ. Consequently, when deallocating values from ρ, some values from ρ' might need to be scanned, marked and preserved.

There are two fundamental reasons region systems can perform the same task with no check at execution time. First, region systems impose a restriction on aliasing: inner regions can point to outer regions but not vice-versa. By contrast, ASAP analyses aliasing without a notion of direction: the *Shape* relation is symmetric. As a result, ASAP is unable to recover the hierarchical structure imposed by regions. Second, regions rely on some runtime code to perform efficient deallocations. Indeed, as detailed in Chapter 3, regions can be efficiently mapped directly onto memory pages. In this case, deallocation of a region's values happens by simply releasing the page to the operating system. Runtime code is necessary to link pages together (when the region grows bigger than a single page) and release linked pages. In ASAP, this runtime code is unavailable; and the notion of regions is absent during execution. ASAP is unable to recover and leverage the region structure because the alias analysis (i.e., the *Shape* property) has no notion of direction.

Thus, this subsumption is weaker than with linear types: deallocations happen at the same point, but in a different manner. Moreover, the code generated by ASAP can have a cost at execution if values alias across regions.

Note that, compiling the region constructs uses the functions *at* and *tickle* to artificially extend the lifetime of affected values. They are inserted only to demonstrate that ASAP can generate deallocations similar to the ones generated by a region-based memory manager. However, ASAP could also ignore those annotations and manage the memory of the program disregarding the region information. In general, ignoring those annotations gives more timely deallocations.

```
$makeSummary$($sampleSize$: $word$): $summary$ @ $\rho_0$ =
    region $\rho_1$ in
    let $d_1$: $data$ @ $\rho_1$ = $getSample$($sampleSize$) in
    let $d_0$: $data$ @ $\rho_0$ = $summarize$($d_1$) in
    let $s$: $summary$ @ $\rho_0$ = {$SampleSize$=$sampleSize$; $Summary$=$d_0$} in
    return $s$
```

(a) A μL+ρ program

```
$makeSummary$($[\![\rho_0]\!]_\rho$: $region$, $sampleSize$: $word$): $summary$ =
    ⊙let $[\![\rho_1]\!]_\rho$: $region$ = {} in
    ⊙let $d_1$: $data$ = $getSample$($[\![\rho_1]\!]_\rho$, $sampleSize$) in
    ⓪let $d_0$: $data$ = $summarize$($[\![\rho_0]\!]_\rho$, $d_1$) in
    ①let $s$: $summary$ = {$SampleSize$=$sampleSize$; $Summary$=$d_0$} in
    ②$at$($s$, $[\![\rho_0]\!]_\rho$);
    ③$tickle$($[\![\rho_0]\!]_\rho$);
    ④$tickle$($[\![\rho_1]\!]_\rho$);
    ⑤return $s$
```

(b) The μL+ρ program compiled

Ⓟ	*Shape*	*Access*
⓪	$[((d_1, \mathit{Wild}(\mathit{data})), ([\![\rho_1]\!]_\rho, \epsilon)) \mapsto 1]$	
①	$[((d_1, \mathit{Wild}(\mathit{data})), (d_0, \mathit{Wild}(\mathit{data}))) \mapsto 0]$	
②	$[((s, \mathit{Summary}), (d_0, \epsilon)) \mapsto 1]$	
③	$[((s, \mathit{Wild}(\mathit{summary})), ([\![\rho_0]\!]_\rho, \epsilon)) \mapsto 1]$	
④		$[([\![\rho_0]\!]_\rho, \epsilon) \mapsto 1, ([\![\rho_1]\!]_\rho, \epsilon) \mapsto 1]$
⑤		$[([\![\rho_0]\!]_\rho, \epsilon) \mapsto 1, ([\![\rho_1]\!]_\rho, \epsilon) \mapsto 0]$

(c) Excerpt of the analysis results

```
$makeSummary$($[\![\rho_0]\!]_\rho$: $region$, $sampleSize$: $word$): $summary$ =
    let $[\![\rho_1]\!]_\rho$: $region$ = {} in
    let $d_1$: $data$ = $getSample$($[\![\rho_1]\!]_\rho$, $sampleSize$) in
    let $d_0$: $data$ = $summarize$($[\![\rho_0]\!]_\rho$, $d_1$) in
    let $s$: $summary$ = {$SampleSize$=$sampleSize$; $Summary$=$d_0$} in
    $at$($s$, $[\![\rho_0]\!]_\rho$);
    $tickle$($[\![\rho_0]\!]_\rho$);
    $tickle$($[\![\rho_1]\!]_\rho$);
    $\sim(CLEAN()((d_1, Wild(data))))$;
    return $s$
```

(d) The result of ASAP's transformation

Figure 9.5: Region μL program: before and after ASAP

Chapter 10

Insights

Studying ASAP casts light on other areas of computer science and on resource management in compilers specifically. We discuss these insights discursively below.

10.1 Caller- and callee-save registers

Of particular interest is insight linking the linear and region regimes with register use convention.

Focusing on function calls we note that under linear type systems, the whole responsibility for the management of the arguments' memory is given to the callee. As a result, if both caller and callee need to perform operations, deep-copying is necessary. Specifically, in a linear program if a caller needs to continue using a value after a call, it needs to make a copy before the call. Dually, under region regimes, the responsibility for the management of the memory is kept by the caller. If a callee wants to make an argument escape its region, it needs to make a copy of it. This is visible in the examples given in the Introduction (Figures 1.2 and 1.3).

Note how, in the linear case, the caller is responsible for copying the values that need to outlive the call whilst, in the region case, the callee is responsible for the copy. This is similar to the way registers fall into two separate categories: caller- and callee-save. In the former, the caller is responsible for copying the value of the register (onto the stack), in the latter the callee is.

Registers are a special kind of resource: their number is predetermined and small. Additionally, registers are flat: there is no aliasing, nor any form of structure to worry about. By contrast, vast amounts of memory can be requested from the operating system and they can contain structured data. Despite these differences, there are similarities in the way that responsibility is split between callers and the callees. One of the roles of resource management strategies (whether for registers or memory) is to arbitrate this agreement between caller and callee. ASAP arbitrates this agreement based on decorations inferred by static analyses, Rust arbitrates it based on annotations provided by the programmer.

We now hypothesise how this insight can be applied to compiler writing. By carrying ASAP-style decorations or Rust-style annotations all the way to the back-end of the compiler, the register allocator could decide what temporaries are caller- and callee-save. Such a register allocator would select the flavour of register based on information that relates to function calls at a higher-level of reasoning. This contrasts with current register allocators, often based on local analysis of register use, often ignoring costly inter-procedural information.

The impact this change would have on register allocators is unknown. One expected impact is about compilation time: inter-procedural register allocation can be costly. Re-using inferred decorations or written annotations would presumably be more efficient than performing inter-procedural analysis. We do not know whether the change would also result in more efficient allocations. Future research is necessary to confirm the improved performance and investigate the potential increased precision.

10.2 Liveness vs scope

An important choice in the design of ASAP is the use of the *Access* property to approximate usefulness. We noted in Chapter 5 how this approximation gives the earliest deallocations but not necessarily the most efficient in terms of CPU usage. We presented an alternative where the deallocations are delayed so as to reduce the amount of scanning: trading CPU work for timeliness.

Placing these alternatives in the context of the subsumptions presented in Chapter 9, we note that *Access* is an exact match for the linear-based memory management. That is because the constraints of linear types are designed to make liveness the exact characterisation of non-waste. On the other hand, note how the translation from region-based μL into standard μL requires the introduction of the *tickle* function. This function artificially extends the *Access* property of its argument. Also note that calls to *tickle* on a region ρ are introduced at the end ρ's scope to avoid early deallocations of values.

This points to another correspondence: linear type systems are based on liveness whilst regions are based on scope. We can see how this insight relates to resource management in different programming languages.

10.2.1 Region-like file-descriptor management

Consider official explanations for the `defer` statement in Go[1]. It is based on an example of managing file descriptors – more specifically: ensuring they are closed once they are not needed. Deferred statements are executed when the function returns, or, in other words, when the scope of inner variables ends. (The main point of deferred statements is that they are executed even if the program `panics` – the Go equivalent of raising an exception.) These deferred statements place file descriptor management under the region style management.

Similarly, in programming languages with higher-order functions, it is common for libraries to provide abstraction over resources in the following way[2]: `with_file: file_name -> (file_descr -> 'b) -> 'b`. During the execution of the call `with_file` n f, the file named n is opened and its descriptor is passed to the function f. When f returns, the function `with_file` closes the file descriptor and passes f's return value along. Using an anonymous function as the second argument highlights how this is a scope-based approach to resource management: `with_file`n`(fun`fd`->...)`. (Note however, that in many of these libraries, it is the programmer's responsibility to ensure the file descriptor does not escape. By contrast, region regimes are enforced by a type-like analyser that prevents such escapes.)

10.2.2 Linear-like file-descriptor management

Using linear types for managing file descriptors (and other similar resources) could be useful. In such a scheme, each file descriptor is consumed by functions such as `write` and `read`. This is similar to the way Clean and Mercury handle I/O: with a linearly typed value[3] referred to as "the world". Every I/O related function of the standard library expects the special, linearly-typed world. This argument is consumed and a new representative for the world is returned – along with the result of the operation.

[1] `https://blog.golang.org/defer-panic-and-recover`

[2] See, e.g., `with_file` and `with_connection` in `http://ocsigen.org/lwt/2.5.1/api/Lwt_io` or `withConnection` in `https://www.playframework.com/documentation/2.0.2/api/scala/index.html#play.api.db.DB\protect\TU\textdollar`.

[3] In this context, linear types are referred to as *unique* or *uniqueness* types.

Instead of managing the whole I/O system with one world value, a language could offer a linearly typed representation of each external resource (file descriptor, socket, etc.). This gives a better granularity than the unique-world technique: separate threads can concurrently write on distinct file descriptors.

10.3 Linear types with alias patterns

Observe that, under linear type systems, it is possible to deallocate values without performing any scanning because of the linear constraint. However these constraints are too tight: they disallow some programs even though their memory can be managed as well as that of a linear program.

Asap lets us explore this area by, first, analysing and transforming an almost but not quite linear program and, second, checking that all deallocations happen without scanning. These programs, whilst not linear enough for the linear type systems, are linear enough for ASAP to handle efficiently. One such not-quite-linear form is programs under affine types: they are handled by ASAP without resorting to scanning. Affine types are a weaker variation of linear types. Thus, by finding other code patterns that are not-quite-linear, we might uncover leads towards new variations of linear types.

One such not-quite-linear form is a restricted use of *alias patterns*. Alias patterns are available in ML (using the `as` keyword), Haskell (using the `@` symbol) and Scala (also using `@`). As their name suggests, alias patterns create aliasing: they allow the programmer to give an additional name to parts or the whole of the matched value. To the best of our knowledge, there are no extensions nor weakenings of linear types that handle alias patterns. However, when used in a specific way (see below), alias patterns are handled by ASAP as gracefully as linear code – i.e., without incurring any form of scanning.

Consider the example in Figure 10.1. The code does not satisfy linear constraints. (One of the breaches of linearity happens at ⓪: the value hys is not used. Similarly, the value hxs is not used at ①. However, these are handled by affine types which we discussed before.) Of particular interest is the alias pattern and its use: both hxs, txs and nxs are used. However, in the branch where nxs is used (at ②), neither of the two others are – and vice versa. As a result, ASAP is able to handle the code without scanning.

We have not formalised the use of alias patterns within a linearly typed program. We posit that, as long as not both of the alias (in our example nxs) and the components (in our example hxs and txs) are used, the memory can be managed efficiently.

10.4 Design space tetrahedron

Designing ASAP and comparing it to existing approaches sheds light onto new aspects of the design space explored in Chapter 3. Of particular importance, the comparison of ASAP with linear and region regimes highlighted the need for a more precise actuator differentiation. Indeed, consider how linearly typed programs are handled by ASAP: after minimal changes to its general purpose analyses, ASAP gives the optimal results we expect from linear types and region systems. Even though both linear types and region regimes are strategies that rely heavily on the compiler, they differ significantly from ASAP. We define two new actuators, the combination of which subsumes the compiler actuator.

Type-like analyser The *type-like analyser* actuator is a component of the compiler that checks the program follows a certain form. In other words, this actuator implements type-like

```
maxMerge(xs, ys) =
    match xs with
    [ Cons {Head = hxs; Tail = txs} as nxs ->
        match ys with
        [ Cons {Head = hys; Tail = tys} ->
            let zs = maxMerge(txs, tys) in
            if hxs >= hys then
                let $r_0$ = Cons {Head = hxs; Tail = zs} in
                return $r_0$⓪
            else
                let $r_1$ = Cons {Head = hys; Tail = zs} in
                return $r_1$①
        | Nil ->
            return nxs②
        ]
    | Nil ->
        return ys
    ]
```

Figure 10.1: Using alias patterns

analyses: if they succeed, the program is compiled, otherwise the compilation is interrupted.

Transformer The *transformer* actuator is a component of the compiler that modifies the program. Note that the transformer can run some analyses in order to decide how to transform what part of the program. However, these are not type-like in that their result is richer than a single boolean.

Note that ASAP is mostly transformer-driven but has a small type-like component. Indeed, remember that one of the hypothesis ASAP makes about µL is well-typedness.

This enriched design space can be represented as an actuator tetrahedron as per Figure 10.2 (instead of the triangle of Chapter 3). The strategies (omitted for clarity) can be placed in the tetrahedron such that proximity to a vertex indicates how much the actuator contributes to the strategy. This tetrahedron can be projected onto the actuator triangle of Chapter 3.

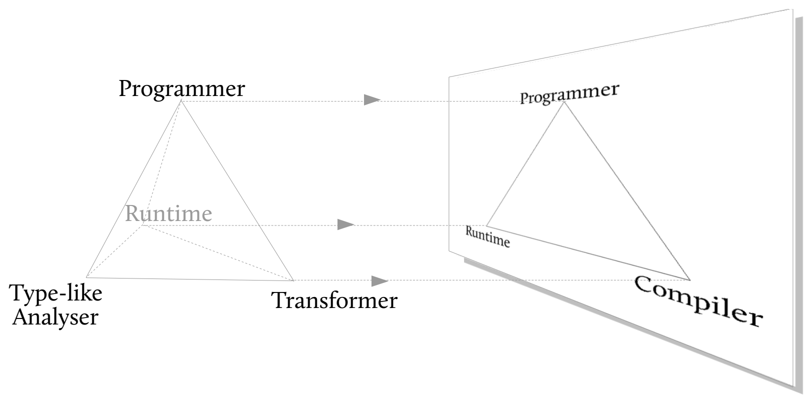


Figure 10.2: The actuator tetrahedron with projection

Chapter 11

Conclusion and future work

We have established a framework to study the design space of memory management strategies. This study revealed the static-automatic gap: an unexplored portion of the design space. Memory management approaches that fit in the static-automatic gap are promising for system programming for two major reasons: their automatic aspect can make them correct by construction, and their static aspect can make them agnostic to memory representation.

We developed such a memory management strategy: ASAP. ASAP is, by construction, correct and agnostic to memory representation; it is able to handle any well-typed program; and its waste approximation can be customised to trade decreased timeliness for increased execution-time efficiency. Designing, developing and studying ASAP provided insight into memory management and other forms of resource management.

Future work In order to test ASAP on real world programs, a full implementation (rather than the prototype described in Chapter 7) would need to be implemented. Other areas where ASAP could be improved include support for higher-order functions, support for optional programmer annotations and better handling of paths. We leave these as future work.

Other interesting leads tangentially linked to ASAP are detailed in Chapter 10. One such lead is to develop new variants of linear types that retain the efficient memory management of linearly typed language whilst accepting a larger number of programs. ASAP can help characterise those programs. Another is to use access decorations to increase the precision of register allocation; specifically, to decide which temporaries are caller- and callee-save. Using these decorations would most probably be faster and possibly more precise than current inter-procedural register allocation.

Bibliography

[1] Boost. `http://www.boost.org/`, 1999. [Online].

[2] Rust. `https://www.rust-lang.org/index.html`, 2015. [Online; accessed July-2016].

[3] A garbage collector for C and C++. `http://www.hboehm.info/gc/`, 2016.

[4] Rahul Asati, Amitabha Sanyal, Amey Karkare, and Alan Mycroft. Liveness-based garbage collection. In Albert Cohen, editor, *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 85–106. Springer Berlin Heidelberg, 2014.

[5] Thomas H. Axford. Reference counting of cyclic graphs for functional programs. *Computer Journal*, 33(5):466–470, 1990.

[6] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[7] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer, 2001.

[8] Edoardo Biagioni, Robert Harper, and Peter Lee. A network protocol stack in standard ML. *Higher-Order and Symbolic Computation*, 14(4):309–356, 2001.

[9] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Imple mentation*, ACM SIGPLAN Notices, pages 197–206, Albuquerque, NM, 1993. ACM Press.

[10] Manuel M.T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. Cocv'03, compiler optimization meets compiler verification a functional perspective on ssa optimisation algorithms. *Electronic Notes in Theoretical Computer Science*, 82(2):347 – 361, 2004.

[11] Sigmund Cherem and Radu Rugina. Compile-time deallocation of individual objects. In Petrank and Moss [31], pages 138–149.

[12] Thomas W. Christopher. Reference count garbage collection. *Software: Practice and Experience*, 14(6):503–507, 1984.

[13] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. page 269–295. Springer-Verlag, 1992.

[14] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, page 162–174, New York, NY, USA, 2001. ACM.

[15] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. *Safety and Liveness in Concurrent Pointer Programs*, pages 280–312. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[16] Kevin Donnelly, Joe Hallett, and Assaf Kfoury. Formal semantics of weak references. In Petrank and Moss [31], pages 126–137.

[17] David Gay Rob Ennals and Eric Brewer. Safe manual memory management. In Greg Morrisett and Mooly Sagiv, editors, *6th International Symposium on Memory Management*, pages 2–14, Montréal, Canada, 2007. ACM Press.

[18] G. W. Hamilton and Simon B. Jones. Compile-time garbage collection by necessity analysis. Technical Report 67, Department of Computer Science and Mathematics, University of Stirling, 1990.

[19] Martin Hofmann. A type system for bounded space and functional in-place update. In Gert Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1782 of *Lecture Notes in Computer Science*, page 165–179. Springer, 2000.

[20] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In Carla Schlatter Ellis, editor, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, page 275–288. USENIX, 2002.

[21] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, page 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[22] Richard Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Notices*, page 13–22. ACM Press, 1995.

[23] U. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[24] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems 30(1), Article*, 2007.

[25] Naoki Kobayashi. Quasi-linear types. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, page 29–42. ACM, 1999.

[26] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: Liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 748–761, New York, NY, USA, 2017. ACM.

[27] John Launchbury and Simon Peyton Jones. State in Haskell. page 295–308. ACM Press, 1996.

[28] Patrick Lincoln. Linear logic. *SIGACT News*, 23(2):29–37, May 1992.

[29] Nancy Mazur, Peter Ross, Gerda Janssens, and Maurice Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury, 2001.

[30] Mike McGaughey. Bounded-space tagless garbage collection for first order polymorphic languages. Technical report, Department of Computer Science, Monash University, 1995.

[31] Erez Petrank and J. Eliot B. Moss, editors. *5th International Symposium on Memory Management*, Ottawa, Canada, 2006. ACM Press.

[32] Venkata K. Pingali, Sally A. McKee, Wilson C. Hsieh, and John B. Carter. Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31:2003, 2003.

[33] François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, page 173–184, September 2013.

[34] Niklas Röjemo and Colin Runciman. Lag, drag, void and use – heap profiling and space-efficient compilation revisited. In *Proc. Intl. Conf. on Functional Programming*, page 34–41. ACM Press, 1996.

[35] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, page 105–118, 1999.

[36] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, page 30–50. Springer-Verlag, 2004.

[37] Andrew S. Tanenbaum. Lessons learned from 30 years of MINIX. *Commun. ACM*, 59(3):70–78, February 2016.

[38] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation Journal*, 17:245–265, 2004.

[39] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[40] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

[41] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[42] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

Index

Colophon

This document was typeset in Calluna, using LaTeX driven by `mk(1)`. Some diagrams were produced using `\tikz`, others `libreoffice(1)`. The source files were written using `acme(1)` and `vim(1)`, under version control by `git(1)`.