A language-independent framework for region inference

Ph.D. thesis
University of Copenhagen

Henning Makholm (henning@makholm.net)

August 15, 2003

Preface

This thesis is the primary result of three years of Ph.D. studies at the University of Copenhagen. It is not the only tangible result; I also have co-authorship of three conference papers [Henglein et al. 2001; Makholm and Sagonas 2002; Taha et al. 2001] and of an invited monograph chapter [Henglein et al. 2005] to show for my efforts, plus two solo conference papers [Makholm 2000a,c] based on my M.Sc. work, and the website of a major conference cluster, ⟨`http//floc02.diku.dk/`⟩. Most of these other articles contain material that is not duplicated in the thesis, except that a large piece of [Henglein et al. 2001] was reworked and extended into Chapter 2.

My original plan for the thesis was a great deal more ambitious than the finished result. Given that the result almost reaches page 200 before the conclusion chapter, which makes it rather long for a Ph.D. thesis (but remember that quantity does not equal quality) it is safe to say that the original plan must have been over-ambitious in the extreme. For example, the idea was that the region optimizations I describe in Section 7.4 would account for about a third of the dissertation, throughly discussed and backed with solid experimental work. But I underestimated the amount of work it would take to arrive at a point where the description would make sense, and (I having foolishly accepted a job offer before I was sure I could finish the thesis) the section ended up as 9 pages hacked together in an afternoon, less than a week before the hand-in deadline.

By some small miracle, what I did manage to write seems to me to tell a relatively coherent story nevertheless. And there will be other opportunities to write about what I left out.

I suspect that many readers will find the longish proofs of Theorems 3.31, 3.46, and 5.16 somewhat distracting and not really interesting. Certainly they do not exhibit any novel innovative proof techniques, but since the formal reason for writing this text is to prove that I am worthy of having a degree conferred on me, I have not always resisted the temptation to show off that I do know how to construct proofs. I hope that my proofs indeed show that, though I could not bring myself to write them in the field's customary "fully rigorous style" with page on page of numbered equations, linked by one-liners like "This, together with (21) and (56), implies" – a style that I personally find utterly unreadable, even if the proof in question is about matters that I feel I understand well.

I also hope, however, that people who read the thesis to learn about region inference (rather than learning what a great person I happen to be) will be able to cope with these occasional fits of self-glorification without being thrown completely off the track.

Language

Our area of computer science has a tradition of avoiding the word "I" in favor of "we" in works with a single author. Always the rebel, I am not quite happy with that convention, and in the thesis I have tried (but probably failed miserably) to choose consistently between "we" when I mean "the author and the reader, investigating the subject together", and "I" when I am about to make a claim that readers may have legitimate reasons to disagree with.

Acknowledgements

Thanks are due to my thesis advisor *Neil Jones* without whose encouragement and ever constructive criticism of my drafts this thesis would be a lot harder to understand than it is,

– to my co-investigators *Henning Niss* and *Fritz Henglein*, with whom I developed the basic shape of the region system I present in the thesis,

– to *Kostis Sagonas* who hosted me during a pleasant stay at Uppsala in the fall of 2001 and introduced me to the mysteries of how Prolog is used in the real world,

– to my friends and coworkers in the TOPPS *group*,

– to *Peter Finderup Lund*, for proofreading, and

– to my *friends and family* who provided moral support and gracefully coped with my occasional "dropping out of the world" in the cause of science.

Thanks also to the many people, most of whose names I don't even know, who created the excellent free software I used for experiments and writing the report, including Moscow ML, the GNU C compiler, GNU Emacs, TEX, LATEX, CVS, XY-pic, and dvips. And, of course, the GNU/Linux system in general.

Apologies

... are due to the reader, because the thesis is not as polished as it ought to be. It has been a race against time to reach this point, and such conventional luxuries as quotations in chapter headings, indices of keywords and notation, and a through proofreading at the end had to be foregone in favor of getting the thing finished at all. I hope to get time to prepare a revised version after the defense where these matters are corrected, but right now I will have to stop typing and hand in what I have, even if the preface has to end in the middle of

Copenhagen,
August 15 2003

Henning Makholm

Contents

1 Introduction **6**
- 1.1 Region-based memory management . . . 6
- 1.2 The region manager . . . 10
- 1.3 Region inference . . . 13
- 1.4 Language-independent region inference . . . 16
  - 1.4.1 Implementations, not languages . . . 17
  - 1.4.2 The UHL model . . . 18
- 1.5 Overview . . . 21
- 1.6 Notation and formal conventions . . . 22
  - 1.6.1 Set theory . . . 22
  - 1.6.2 Maps, especially finite ones . . . 23
  - 1.6.3 Variables and metavariables . . . 24

2 Previous approaches to region inference **25**
- 2.1 Tofte–Talpin . . . 26
  - 2.1.1 Regions and closures . . . 28
  - 2.1.2 Region polymorphism . . . 28
  - 2.1.3 The TT region type system . . . 29
  - 2.1.4 Algorithms for TT region inference . . . 30
  - 2.1.5 Other region inferences based on TT . . . 31
  - 2.1.6 Unused region parameters . . . 31
- 2.2 The ML Kit . . . 31
  - 2.2.1 Storage modes and region resetting . . . 32
  - 2.2.2 Storage-mode polymorphism . . . 34
  - 2.2.3 Storage modes and region aliasing . . . 34
- 2.3 Aiken–Fähndrich–Levien . . . 35
  - 2.3.1 Region aliasing and the AFL system . . . 36
- 2.4 Henglein–Makholm–Niss . . . 37
  - 2.4.1 Game of Life in the HMN system . . . 40
  - 2.4.2 Simulation of the Kit and AFL systems . . . 41
  - 2.4.3 The HMN region type system . . . 43
  - 2.4.4 Non-optimality of region annotations . . . 50

3 The universal host language **52**
- 3.1 Uniform mutators without region annotations . . . 52
  - 3.1.1 A simple example . . . 53
  - 3.1.2 The procedure-less fragment of UHL . . . 55
  - 3.1.3 Subroutines . . . 59

3.1.4 Ideal execution of uniform mutators . . . 63
3.1.5 A scoping discipline for uniform mutators . . . 67
3.2 Region annotations for universal mutators . . . 69
3.2.1 What is an agent? . . . 72
3.2.2 Managed execution of uniform mutators . . . 73
3.2.3 A scoping discipline for agents . . . 78
3.3 Region soundness . . . 82
3.3.1 One half of soundness is trivial . . . 84
3.3.2 Soundness from safety: Pointer-blind programs . . . 86
3.3.3 Preservation of soundness . . . 92
3.4 Annotatable edges and flowchart chunks . . . 93
3.4.1 Call chunks . . . 94
3.4.2 Guidelines for chunk size . . . 94
3.4.3 Pruning the annotatable edges . . . 96
3.5 Possible extensions of the UHL model . . . 97
3.5.1 Region closures as in the TT model . . . 97
3.5.2 Concurrency . . . 98
3.5.3 Finite regions . . . 98

4 Application to ML **99**
4.1 Indirect calls . . . 100
4.2 The translation . . . 103
4.2.1 Variables and let bindings . . . 104
4.2.2 Arithmetic, I/O, and conditionals . . . 105
4.2.3 Lists . . . 107
4.2.4 Function abstractions and application . . . 108
4.2.5 Simpe exceptions . . . 111
4.3 Region annotations for ML . . . 112
4.4 Extending to full Standard ML . . . 114
4.4.1 Datatype and pattern matching . . . 114
4.4.2 References . . . 116
4.4.3 Constant expressions . . . 116
4.4.4 Standard ML exceptions . . . 116
4.4.5 Mutual recursion . . . 119
4.4.6 The module language . . . 119

5 A region type system for UHL **120**
5.1 A stepwise introduction . . . 121
5.1.1 Basics: Simple pointers and procedure calls . . . 121
5.1.2 Subplacing: HMN without complex data structures . . . 123
5.1.3 Pointers to pointers . . . 126
5.1.4 Tuples . . . 127
5.1.5 Tagged sums . . . 127
5.1.6 Destructive update . . . 129
5.1.7 Recursive types . . . 131
5.2 The full region type system . . . 132
5.3 Safety proof for the region type system . . . 135

6 Other host-language features **143**
6.1 Generic imperative language features . . . 143
6.1.1 Destructive update of heap cells . . . 143
6.1.2 Loops and gotos . . . 144
6.1.3 Global variables . . . 144
6.1.4 By-reference parameters . . . 145
6.1.5 Pointers to global variables . . . 146
6.1.6 Pointers to local (stack-allocated) variables . . . 146
6.1.7 Separate initialization of new heap blocks . . . 148
6.1.8 Pointer tricks . . . 149
6.2 Object-orientation . . . 149
6.2.1 Class hierarchies and subtyping . . . 149
6.2.2 Dynamic method dispatch . . . 151
6.3 Logic programming . . . 152
6.3.1 Backtracking . . . 153
6.3.2 Types . . . 154
6.3.3 Unification . . . 155
6.3.4 The WAM . . . 156

7 Region inference algorithms **157**
7.1 Overview . . . 157
7.1.1 The prototype implementation . . . 159
7.2 Constructing skeleton types . . . 161
7.2.1 From ML-like types to type graphs . . . 161
7.2.2 Region-annotated type expressions . . . 163
7.2.3 The skeleton typing in detail . . . 163
7.2.4 Adding closures . . . 165
7.3 Basic region inference . . . 165
7.3.1 General invariants and region-operation selection . . . 166
7.3.2 No loops, no procedures, no subplacing . . . 169
7.3.3 Handling uncounted variables . . . 171
7.3.4 Calling a leaf procedure . . . 172
7.3.5 Unrestricted calls . . . 174
7.3.6 Subplacing in the prototype . . . 176
7.3.7 Subplacing by propositional networks . . . 177
7.3.8 Loops in procedure bodies . . . 180
7.4 Region optimizations . . . 182
7.4.1 Local alias propagation . . . 183
7.4.2 Global alias propagation . . . 184
7.4.3 Correctness of alias propagation . . . 186
7.4.4 Region merging . . . 187
7.4.5 Parameter lifting . . . 190
7.4.6 The “unrename” transformation . . . 190

8 Conclusion **192**

Chapter 1

Introduction

This thesis is about **region inference**. Many readers will not know in advance what I mean by that, and even those who think they know it may be mistaken. I am using the term a little more generally than people usually do.

Region inference is a topic within the greater subject of **region-based memory management**, so to explain what the thesis is about, let me start by introducing region-based memory management.

1.1 Region-based memory management

Region-based memory management is an alternative to garbage collection which relies on compile-time analysis rather than run-time pointer tracing for correctness. So goes the one-sentence abstract that can be given when someone at a conference dinner asks, so what do you work with? The answer is mostly true, but in itself it does not really tell the reader anything. So it will be followed up by further explanations and the drawing of diagrams on napkins.

Usually, the first diagram I draw in such situations is something like the one in Figure 1.1. It does not really say much more than the one-sentence version, but it says it in smaller bits that are more likely to take hold in the listener's consciousness. It also introduces a couple of key terms that I have invented specifically for this thesis and will be using unashamedly for the rest of it, so let us dwell upon it a little.

Figure 1.1(a) is a block diagram of the software modules that are involved with memory management in a language such as C that use **manual** memory management. There are three boxes: the mutator, the C library and the operating system.

The **mutator** is the program that the programmer wrote. This term originates in the literature about incremental garbage collection, where the principal feature of the mutator is its tendency to change (or "mutate") the contents of heap cells in parallel with the garbage collector's scanning operation. Because I know of no other nicely short and anthropomorphic name for the-program-as-written-by-the-programmer, I will continue using that word throughout.

The mutator talks to the C library using an interface consisting of two functions. `malloc()` is a request to have a block of heap memory set aside for the

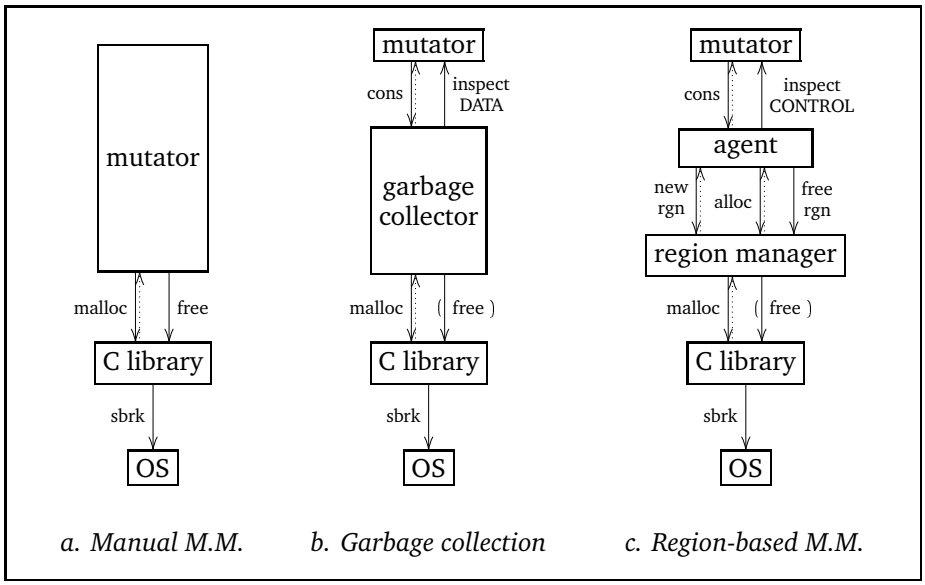


Figure 1.1: Different regimes of memory management

mutator's use (a pointer to the allocated block is returned to the mutator, symbolized by the dotted arrow on the figure). `free()` asserts that that the mutator will not use a specified block of memory anymore; the addresses in that block can be returned in response to future `malloc()` requests, or perhaps be overwritten with the C library's private data.

The C library, in turn, typically acquires memory in bulk from the operating system. The details of this depends on the operating system; the figure implies that the classic Unix `sbrk()` interface is being used, but in practise it is as likely to be done by other means, such as `mmap()`.

Implicit in this description is the assumption that whenever I talk about **memory management**, the memory I refer to is the dynamically assigned memory that the mutator may find, while it runs, that it is going to need for indeterminate periods of time. Specifically, we do *not* care about (call) stack memory where small pieces of intermediate data are stored, or about statically allocated memory for global variables (in languages that support those) or the code of the mutator itself. Also, we are concerned about the "retail" allocation of small individual pieces of heap memory, perhaps a few or several machine words long each. The "wholesale" allocation of big pieces of memory that is dealt with by the operating system and its virtual-memory subsystem, is a large and interesting research subject, but in the entire thesis we will just take its services for granted.

The correctness of manual memory management is entirely up to the human programmer's skill. If he writes a mutator that calls `free()` prematurely, later accesses the the heap block may or may not find it overwritten with unrelated data – and the "may not" makes such errors quite hard to discover. The reverse case is also bad: If the mutator completely neglects to call `free()`, the result is a *space leak*, sometimes unappreciated by inexperienced programmers but dreaded by developers of server software or complex interactive applications.

The problem is that human programmers are often not up to the task of programming a mutator with the `free()`s placed just right. And though some are, fewer still are capable of *maintaining* a mutator such that the `free()`s stay placed right.

Therefore **garbage collection** was invented. As shown on Figure 1.1(b), the responsibility of controlling when heap blocks are freed have been removed from the mutator. This leads to a mutator that is smaller and easier to understand – the human programmer can concentrate his efforts on specifying a computation rather than planning how to make efficient use of space for intermediate data.

In between the mutator and the C library is now sandwiched a special runtime component: the **garbage collector**. It provides a simple `cons` service to the mutator, with a semantics much like that of `malloc()` in the manual model. However, the garbage collector is supposed to find out *itself* when it is safe to reused memory that has already been returned from `cons` once.

To enable the garbage collections to make such decisions, it is allowed to *inspect the contents* of the heap memory it has provided to the mutator, as well as the values in the mutator's local variables and call stack. The mutator is expected to maintain certain conventions about the data in the heap and stack that allow the garbage collector to draw conclusions from the bit patterns it finds in the memory. These conventions may range from elaborate tagging and "stack map" schemes to the minimal, implicit rules for conservative garbage collectors. Even though the latter may work without explicit compiler support, the mutator must not, say, store pointers into allocated memory in encrypted form and afterwards expect the decrypted pointers to be useful.

The garbage collector itself typically gets its memory in large blocks from the C library using the standard `malloc/free` interface, and does its own subdivision into smaller `cons` blocks. In principle it could have gone directly to the operating system, but implementers often refrain from this, because the garbage collector needs to coexist with the C library anyway, and because it means that they won't have to embed detailed knowledge about how to interact with different operating systems into the garbage collector's source code. (Figure 1.1 shows the `free` operation in parentheses, because it is common for garbage collectors never actually to free memory to the lower layers in the stack, except in special circumstances).

Intuitively, the garbage collector is successful if someone who watches the mutator's result cannot learn that memory is being reused. The programmer who writes the mutator should be able to assume that each `cons` operation returns a fresh block of memory that will never be reused. If the mutator's behavior *with* the garbage collector differs from the behavior the mutator *should* have, given the semantics of the programming language, then by definition the garbage collector must have reused memory "too much" or "too early".[1]

The garbage collector must work, however, within a terrible constraint: A garbage collector is a one-size-fits-all component that is supposed to work for all mutators compiled by a particular compiler. Therefore, the garbage collector cannot let its decisions depend on which mutator it works for – because it

[1] We need to except the situation where an intelligent observer can conclude that memory is being reused simply because the mutator keeps running past the point where a reuse-free implementation would have stopped with an out-of-memory error. Box 3.1, on page 55, discusses some technical issues with this exception.

Box 1.1—Why "agent"?

What I call the "agent" is usually just referred to as "region annotations" on the source code for the mutator, but I need to speak about it as something that *does* things. Therefore I have decided to use the term "agent" about it throughout the thesis. This choice of words reflects my view of the agent as an independently acting co-process (written in a very domain-specific language) rather than passive comments to the mutator.

The word "agent" is supposed to invoke images of men in nondescript suits and dark sunglasses who hover discretely around heads of state or government to prevent them from being shot in the foot (or elsewhere). As far as I know, these agents rarely care about the political content of the president's (or whomever the VIP in question is) speeches, but they will get quite agitated if he decides to stroll through a mall with another trajectory than the planned one. It seems to me that this is a good metaphor for the software component I speak about, which reacts to the control flow of the mutator rather than to the data it processes. Even more so because the president seldom openly acknowledges the presence of the agents.

Still, the particular word "agent" itself might be contested. I have not researched these matters in detail, but it might be that only in the United States are the black-clad gentlemen in question officially known as "Secret Service agents". A more generic word might be "bodyguard". However, Hollywood's influence on the global mind-share is a force to be reckoned with, and if I have to choose between spurious mental images of James Bond and ones of Whitney Houston, I choose Mr. Bond.

doesn't know.[2]

To summarize: A garbage collector *cannot* observe the mutator, but it *does* observe the data that the mutator works on. **Region-based memory management** can be viewed as an attempt to reverse this relation. Its model is shown on Figure 1.1(c). Here the job formerly done by the garbage collector is split into two: The **region manager** handles the subdivision of memory into mutator objects. It is a one-size-fits all component like the garbage collector, but its interface is based on the more elaborate **region model** (see Section 1.2) where its client specifies explicitly when to deallocate memory.

The mutator is still the same as before; in particular it still uses the same simple interface to allocate memory. Between the mutator and the region manager we place the **agent**, whose task is to convert the mutator's simple allocation requests and region-based allocations and deallocations for the region manager.

As with garbage collection, the agent is successful if an observer of the mutator cannot learn from its behavior that memory is being reused. When that is the case, we say that the agent is *region sound*.

The agent works under conditions that are in some sense dual to the garbage collector's: It does *not* observe the mutator's data (not because it can't, but because it takes time to extract information from it), but it *does* observe the mutator's *control flow*. It can do this because the agent is specific to one particular mutator, created specifically for that mutator by the compiler. In practise the

[2]Depending on the hardware and OS protection abstractions, the garbage collector may be allowed to look at the mutator's machine code, but algorithms that can analyze machine code to get useful data about its memory-usage patterns – and do it fast enough to be practical at run time – are not known.

compiler will interleave machine code for the agent and the mutator, making the agent piggy-back on the mutator's control flow.

Because the agent is tailored to a particular program, it stands a much better chance of preventing the mutator from learning that memory is being reused. Whatever devious trick the mutator uses (such as encrypting a pointer and later, after the memory block pointed to is not itself used anymore, comparing the encrypted pointer to the encryption of a freshly allocated pointer, to see whether memory is being reused), that trick will be in the mutator's source code, which means that we can just select an agent that prevents the trick from being useful (such as making sure not to deallocate the pointer being encrypted before the comparison has taken place).

Of course, *noticing* the trick in the mutator's source code and using it to select an appropriate agent is not trivial and probably hard to automate to the degree that useful agents for mutators written in "dirty" C can be constructed automatically. However, it would still be conceivable to automatically do a static analysis on a C program such that "well-behaved" mutators that do not cheat with pointers would be recognized as such and then had a "good" agent constructed for them, while mutators that failed the check would have to make do with more conservative agents.

1.2 The region manager

What is the role of the region manager in all this? Could the agent not just talk directly to the C library, doing a `malloc` operation for each of mutator's cons requests, and eventually a doing a matching `free` on its own initiative?

Well, in principle it could, but it would need to keep a large amount of internal state to do it, which would take time to maintain and make the agent rather complex. That is where the region manager enters the picture: It implements the **region abstraction** which allows the agent to keep track of unboundedly many memory blocks with only a finite amount of internal state (per recursion level, at least).

Figure 1.1 enumerated the three primitives of abstract interface between the agent and the memory manager:

Create a region. Intuitively a **region** is a part of the heap where memory can be allocated. Exactly which memory belongs to the region may change with time; when the region is first created it contains no memory at all.

When the agent asks for a new region to be created, the region manager returns a **handle** to the region. The handle can be used in subsequent region-manager operations.

Formally, a region is best viewed as simply an abstract concept that could also be though of as a "license to allocate memory", and in terms of the abstract interface to the region manager we could say that the handle *is* the region. However, the metaphor of a region as an area on the heap that can "contain" memory blocks is a strong and helpful intuition, which would not work if we spoke about the region itself being passed back and forth between the agent and the region-manager. Thus the auxiliary concept of a "handle" or "reference" to a region.

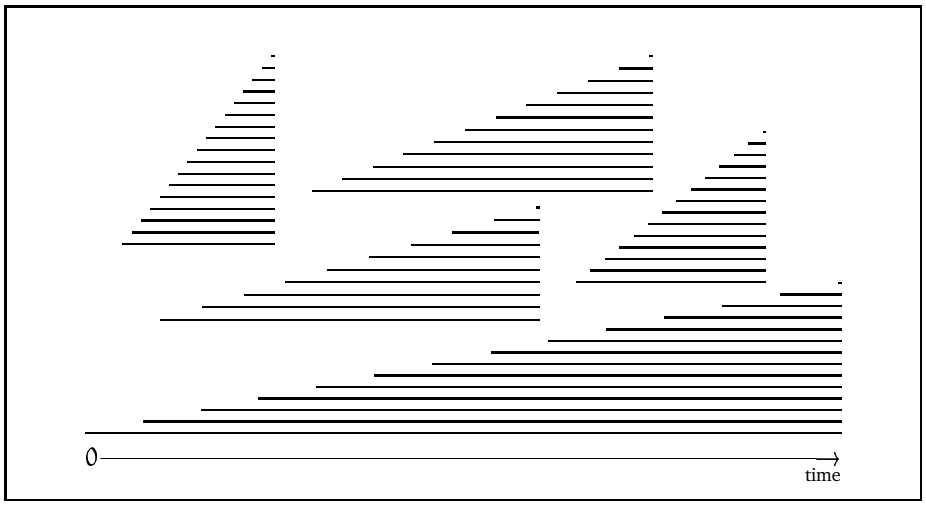


Figure 1.2: Memory-block lifetimes in the region model. Each horizontal line corresponds to the lifetime of one memory block. The triangular collections of blocks correspond to regions. The figure shows that several regions can exist at the same time, all growing; but a region can shrink only by being completely deallocated.

Allocate memory. The agent can allocate memory only by supplying a handle to the region that it wants the new memory to belong to. The agent also specifies how many memory cells it needs to allocate (actually, by passing on the number from the mutator's request), and the region manager somehow selects that many consecutive unused cells and returns a pointer to them, after adding them to the specified region.

A useful half-fiction is to think of this as if the region manager locates some unused space that already belonged to the region before – we can then speak of allocating memory "in" or "from" a specific region. One should, however, be aware that this language does not reflect the formal properties of the abstract region-manager interface, which does not recognize any connection between currently unused memory and specific regions.

Destroy a region (or "deallocate" it). After a region has been destroyed, it is of course not valid for further allocation requests. Destroying the region also has the important side effect that all memory blocks ever allocated in the region are deallocated and considered available for reuse. *This is the only way to deallocate memory in the region model*.

This deallocation is the whole point of the region model; the *raison d'être* of the region concept is to provide the agent with an efficient way of telling the region manager when to deallocate memory blocks. Thus, a third view of regions is as a "brand" that the agent puts on allocations in order to be able to find and deallocate them later.

Figure 1.2 shows an example of five regions and the lifetimes of the memory block within them.

This description of the region manager is intentionally quite abstract. It is important for parts of the formal development in the rest of the thesis that we can assume that the region manager has full freedom in choosing which cells it selects for each allocation operation, subject only to the condition that they must be currently unused.

In practise, however, there is a specific implementation principle for the region manager that is usually implied by "region-based memory management". It consists of dividing the available heap memory into fixed-size **cards**. A region consists of a set of cards, tied together in a linked list. Allocations in the region happens at one end of the front card in the list; when the unused space in a card becomes too small to satisfy the next allocation another card is fetched from a global list of unused cards and added to the region as the new front card. When the region is deallocated the entire card list is appended to the free-list, making it available for use in other regions.

A region handle is a pointer to a small record of management data, containing pointers to the first and last card in the list and information about the amount of yet-unallocated space in its front card. The management record is typically located in the first card of the region, although in the ML Kit it is allocated as a local variable on the call stack.

(Traditionally the pieces of memory that make up a region are called "pages". I prefer the term "card" which suggests a smaller granularity than the several hundred cells that are used for "pages" in virtual-memory schemes at the operating system level).

The standard region-manager implementation eliminates fragmentation of the heap and has low administrative overhead because there is no need to keep track of how the used part of a card is divided into individual memory blocks. In fact, it is possible to implement each of the three operations in the region manager's interface such that they run in guaranteed constant time. (Of course, if the free list runs empty and a fresh batch of cards must be requested from the lower layers, the running time will be a the mercy of the C library and operating system – but the work done by region manager *itself* stays bounded).

This means that it should be possible to reason naturally about the running time of programs – if you count the number of agent operations done between points A and B in the program, you can get a fairly tight estimate of the total time spent on memory management. This property should make region-based memory management an attractive option in real-time or interactive environments where it is important to be able to guarantee response times. Strangely enough, I am not aware of any actual published experiments to reap this benefit, perhaps because time-constrained systems are not usually written in the languages that have hitherto been used for region-based memory.

Another important feature of the region manager is that it is a very simple component. The source code for the region manager I have used for benchmarking experiments Henglein et al. [2001, see] totals about 120 lines of straightforward portable C code[3]. Furthermore, the interface between the region manager and the mutator/agent is fully specified by the the three region operations described above. The region manager does not depend on the cooperation of its

[3]Not counting about twice as many lines of instrumentation for profiling and general statistics collection.

client for traversing the stack or moving memory blocks after allocation, because it does not do any of these things.

This simplicity makes region-based memory management a strong choice in connection with mobile code, where one has to establish trust that a piece of code of unknown (possibly malicious) origin can be executed safely. It is easier to satisfy oneself that a 120-line region-manager does not contain any security holes that an attacker could exploit, than to reach the same degree of certainty about a complex state-of-the-art garbage collector. (Of course, then one needs to establish trust in an *agent* of unknown origin, but that can be done by a *region type system* that allows a formal proof of agent safety to be encoded and transmitted along with the mobile mutator and its agent).

A third reason to use region-based memory management is simply that some times it works better than garbage collections. Of course this is not always true – there are programs that are simply not well suited to having decisions of memory management made statically. Consider, for example, an interpreter for a general-purpose language. Even though an agent may be tailor-made to the *interpreter*, it does not know the interpreted program (and is prevented by design from adjusting its actions to it). When the interpreted program requests an allocation and the interpreter passes it on, the agent will have no idea about how long it is going to be needed for.

But when region-based memory management works, it tends to work really well. The region manager is cache-friendly, since its list of free cards naturally works on a first-in-first-out principle (whereas a garbage collector is sensitive to differences between the size of the nursery generation and the available first-level cache that can be afforded for short-lived heap allocations). And there are few time-to-space tradeoffs for the region manager. As long as there is enough memory available to run at all, it works at full speed, and if there is not – well, in the lucky cases this happens only far below the threshold where a garbage collector goes belly-up.

The region manager is inherently more space efficient than a typical garbage collector – most of the region cards will be completely filled with payload data, expect for the link to the next card in the region and the "slack" at the end where there was not sufficient space for the next allocation. Another reason is that an agent can sometimes get away with deallocating memory objects that a garbage collector would consider live, because it knows that the mutator does not contain the *code* to access it.

1.3 Region inference

Region-based memory management was born as an independent research area by a seminal paper by Tofte and Talpin [1994] which suggested that agents could be *generated automatically* by a compile-time analysis of the mutator's source code. Such automatic generation is known as **region inference**, and is the topic of this thesis:

> *Region inference is the (automatic) construction of a sound, efficient agent for a particular given mutator.*

From the above discussion, we know (hopefully) what I mean by "agent". A "sound" agent is one that does not allow an observer to notice that memory is

Box 1.2—Why "region inference"?

The term "region inference" is well established. The generic use to which I am putting it is perhaps not.

According to the *Oxford Advanced Learner's Dictionary*, "inference" means "a conclusion reached on the basis of knowledge or facts", or the process of arriving on such a conclusion. That does not seem to have much to do with my definition. What is going on?

The story starts with **type inference**, originally the process in a compiler where the type of an expression was deduced from knowledge of its subexpression's types. During the years, type inference techniques matured and reached a famous high point with ML [Milner 1978] where the types of all expressions in a program could be inferred without a single explicit type declaration.

When Tofte and Talpin [1994] proposed the first automatic region inference (in by broad sense) they based its structure on sophisticated descendants of Milner's type inference. Therefore they named their particular algorithm, as well as its specification, "region inference".

For several years, Tofte and Talpin's region inference (in their sense) was the only known region inference (in my sense). A couple of algorithmic variants were developed [Birkedal and Tofte 2001; Tofte and Birkedal 1998], but essentially they could be recognized as variants of the same thing. Some extensions to the process were proposed [Aiken et al. 1995; Birkedal et al. 1996], but they built upon the results of the Tofte–Talpin inference.

The Tofte–Talpin monopoly on automatic agent generation were only broken in early 2001, when I, Henning Niss, and Fritz Henglein were developing what later became the HMN model described in Section 2.4. We wanted to be able to generate region annotations automatically and thought of it as wanting "something like the region inference process". This quickly became shortened to wanting "*a* region inference", so in our paper [Henglein et al. 2001] we simply called our technique for producing annotations automatically "region inference" even though it had little to do with the type-inference techniques that inspired Tofte and Talpin.

From there, the term just stuck – time will show if it keeps sticking.

being reused. "Efficiency" concerns space and time: How much space will the region manager need allocate from `malloc` with a particular agent making the decisions, and how much time will the agent (and region manager) need to make (and implement) those decisions? As is so often the case, there is a tradeoff between space efficiency and time efficiency; we will be concerned mainly with optimizing space efficiency, with time efficiently having second priority – as long as the time used by the agent and the region manager is *at most linear* in the time used by the mutator itself.

For the purpose of understanding the similarities and differences among region inference techniques, it will be fruitful to decompose the region-inference problem into three subproblems.

1. **An agent programming language.** While agents can, in principle, be written in any general-purpose programming language, particular region-inference methods will normally only result in agents with a certain simple internal structure. I propose that in such a situation, the resulting agent should be viewed as belonging to a *domain-specific programming language* specifically designed for writing agents.

Programs in a well-designed agent programming language will often be easier to understand mechanically than agents written in a general language. Also, limitations of a region-inference method will in general be easier to understand if they can be described as explicit limitations in the agent programming language's expressiveness than if one had to understand the actual inference algorithms.

Since the mutator is in general written in a different language than the agent, we will refer to the mutator's programming language as the **host language**. The intuition is that the agent language is embedded in the host language.

2. A region type system. By this I mean a structured (and algorithmically feasible, in some sense) characterization of a class of mutator–agent pairs, such that all pairs in the class are region sound.

Usually, this is accomplished with a specially-instrumented type system for the host language, where "region annotations" on types provide a connection to the agent's source code. Indeed, when we derive a region type system in Chapter 5, we will see that a workable system for reasoning about agent soundness will *naturally* take the character of a type system; this is my justification for defining the term "region type system" functionally rather than explicitly requiring it to have to do with types.

The region type system allows one to construct a "soundness certificate" for an agent, consisting of enough information to quickly construct and verify a typing derivation. Soundness of the region type system means that the existence of a typing derivation does indeed imply that the agent is sound. Ideally, this should be proved rigorously, but what the literature often provides in reality is just intuitive plausibility backed with proofs for simpler cases (say, no destructive updates or no module system).

3. A region inference algorithm. This is the question of how to *derive* a efficient agent *and* its soundness certificate for a given mutator, given that the agent programming language has been fixed in advance.

It should be emphasized that this analysis is not meant as a divide-and-conquer decomposition of the original problem. There is a rich mutual interdependence between the three subproblems – the choice for earlier subproblems determine which answers to later ones are valid or at all meaningful, which again means that one's actual choices for early problems are often motivated by what one would like to be able to later on.

The analysis, however, is useful for getting an overview of the field. Having identified the subproblems helps imagine how the big picture fits together, especially as many parts of the literature focus on only one or two of them. For example, there is a sequence of works about programming systems [Crary et al. 1999; DeLine and Fähndrich 2001; Grossman et al. 2002; Walker and Watkins 2001; Walker et al. 2000] that provide an integrated agent programming language and an (often very strong and sophisticated) region type system but expect the user to write the agent himself. On the other hand, work like Birkedal et al. [1996] focus on improving the expressiveness of the agent programming language, and devise algorithms for exploiting this extra expressiveness but does not base these algorithms on a region type system of their own.

1.4 Language-independent region inference

Most of the existing work that include algorithms for region inference use ML or ML-like toy languages as their host languages – indeed, the only exception I know of is my own M.Sc. thesis [Makholm 2000b], where I implemented a simple region-inference for Prolog because I needed benchmark programs for my backtracking region manager and did not want to put in the region annotations by hand.

This is unfortunate, because there is quite some interest in applying region-based memory management to other language paradigms. However, it is not obvious from the existing literature exactly how to transfer its lessons to other languages. Many of the difficulties and peculiarities of the published region-inference techniques are there to solve specific problems in dealing with the (polymorphic) lambda calculus, which are simply not present for languages in other traditions. It is not clear how much of the details one needs to carry over to another language. Therefore, the task of using region-based memory management for a new language may seem more difficult than it actually is.

My goal with this dissertation is to remedy this situation. If I had to state a formal thesis (in the sense of a concrete claim that the dissertation aims to establish) would be something like:

> *It is actually quite easy to add region-based memory management to an existing language (implementation).*

I will argue for this by presenting a *general theory* of region inference, designed to be applicable for a wide range of different host languages. Such a general theory will be somewhat more complex than each concrete instance of it, but the effort in the generalization should be well spent: A reader who wishes to apply the theory in a concrete setting can simply ignore the parts of it that are not relevant to him, but will not have to recreate my formal arguments in his own setting.[4]

To substantiate my claim that the theory can be applied to a wide range of host languages, I will argue that it is applicable to typical representatives of three different language traditions

- ML-like languages, represented by a large subset of the Standard ML Core language, including functions as computed values, updateable references and (limited) exceptions. This is the most fleshed-out example (because most of my practical experiments and the previous literature concern similar languages). A description of this language and an interface between it and the theory appears in Chapter 4; it will be the running example in Chapters 5 and 7.
- Imperative and object-oriented languages in the tradition of Pascal, C, C++, Java. The theory supports languages with unrestricted gotos within method bodies. How to support language features of these languages is discussed in Sections 6.1 and 6.2
- Logic programming, represented by Prolog with backtracking and cuts, described in Section 6.3. A special property of this example is that the

[4] Of course, people *will* recreate my formal arguments in other settings ... reuse of theory and arguments seems to work better as a creed than in deed.

host language is not typed – the region inference needs to include a "soft typing" inference of its own as a before it can start the region inference proper.

The languages that are supported are all sequential, with a control flow that is directly apparent from the program text except for procedure calls that use a conventional call stack according to a last-in-first-out discipline. ML-style exceptions are a slight extension of the LIFO stack discipline but can be handled within the same framework. Backtracking in Prolog may cause violations of the call–return discipline, but as we shall see in Section 6.3.1 this problem can be side-stepped completely, thanks to the special backtracking-aware region manager I developed in [Makholm 2000b,c].

1.4.1 Implementations, not languages

A fundamental premise underlies my approach to generalizing region-based memory management:

> *Automatic region-based memory management is not something you add to a programming language. It is something you add to a particular implementation of a programming language.*

What I mean by this is that it is, in general, futile to develop a theory of region-based memory management for some language based on a high-level description of how the language works, and then assume that the theory so produced will be useful for a particular implementation of the language. This is because high-level descriptions of how languages work commonly do not completely specify when heap objects are allocated and accessed. For example, the Definition of Standard ML [Milner et al. 1997] just assumes that the implementation has some way to implement tree-shaped data and closures, but speaks explicitly of the heap (or store) only in the case of references. Such omissions are often deliberate and intended to give implementations maximal freedom to choose for themselves how to make use of actual heap memory. And implementations do use that freedom, to the extent that two different implementations of Standard ML will probably give rise to the different sequences of heap allocations and accesses for all but the most trivial of source programs. This is caused by differences in closure representation, unboxing strategies, exception handling, optimizations to make lists and datatypes more efficient, et cetera, et cetera.

As a concrete example, consider the Standard ML definitions

```
fun f pair : int list = op:: pair
fun g x = f(x,nil)
```

In a simple, "naive", implementation of Standard ML, the value returned from `g` would be a pointer to a heap structure involving the pair object that `g` creates before calling `f` – but many actual implementations will attempt to unbox function arguments and/or cons cells in lists, possibly with the result that `g` does not itself allocate any part of its own return value. On the other hand, sophisticated unboxing schemes could potentially lead to the introduction of heap-allocating "reboxing" operations at places where there is no obvious allocation construct in the source code. All this means that we should not expect to construct a

free-standing region inferencer for Standard ML in general without reference to a particular implementation.

Of course, some languages exhibit these differences to lesser degrees than others. In Scheme, compound data are almost always modifiable, which means that its definition [Kelsey et al. 1998] has to define heap transactions in much greater detail than that of Standard ML. And in low-level languages such as C, the details of heap allocations are usually directly visible to the programmer.

The typical way the implementation-dependency has usually been handled in work about region-based memory management is that one constructs a theory of regions for the intended host language based on reasonable default assumptions about when memory needs to be allocated and accessed. Then one proceeds to construct an entire implementation around the region theory and its underlying assumptions. It is debatable whether this description fits the addition of regions to the ML Kit (but one may argue that the reason why the TT system does not appear implementation-dependent at first glance is that the ML Kit was already deliberately adhering to a naive data-representation model that happened to fit the "reasonable default" assumptions underlying the TT system). But it certainly describes the history of the prototype implementations I (co)developed for [Makholm 2000b] and [Henglein et al. 2001].

However, if region-based memory management is to conquer the real world, we cannot keep producing new implementations from scratch each time we want to "add regions to language X". We need to be able to take an existing implementation of some programming language, with its particular memory-management subtleties, and construct a system for region inference and region-based memory management that fits that implementation. To the best of my knowledge, the only time this has been attempted with a production-strength implementation of a real language has been the experiment by myself and Kostis Sagonas [Makholm and Sagonas 2002] on adding region-based memory management to an existing Prolog implementation. The theory I present here is derived partly from insights gained by that experience.

1.4.2 The UHL model

My central idea is to consider region annotations and region inference at the level of the *compiler intermediate language* rather than source code. At the intermediate-language level, heap allocation and access is usually explicit. Also intermediate languages for implementations of different languages – or different implementations of the same language – are often similar enough (though not identical) that the same algorithms for region inference can be used for all of them, with only superficial differences.

In order to *present* and *reason about* my region-inference techniques, I will formulate the general theory for for an abstraction which I call the "universal *host* language", in short UHL. As a first approximation, UHL can be looked at as the union of "all sensible intermediate languages" with everything that is irrelevant to memory management abstracted away. I will develop general region-inference techniques for "programs written in UHL" (which we shall refer to as **uniform mutators**, because mutators originally expressed in different languages look the same to the region-inference algorithms), and these techniques can then be used as blueprints for implementations that work on the host implementation's actual intermediate language.

A cookbook recipe for adding automatic region-based memory management for an existing implementation would now be something like

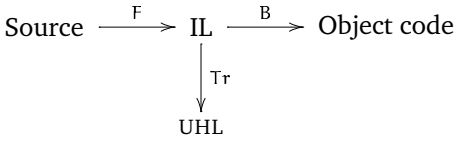
1. Identify the point in the compiler pipeline where region inference should be done. This will usually be quite late, because the agent that will be generated is sensitive to the detailed control flow of the mutator, so region inference should come after any optimization passes that may reorder code. On the other hand, if the source language is typed, the types should still be present at this point; the region inference will need them. Implementations of typed languages that do aggressive optimizations after forgetting types may need major rewriting in order to prepare them for region-based memory management.

   In the rest of the recipe we will refer to the intermediate language at this point as *the* intermediate language (because any other ones are not relevant to us here).

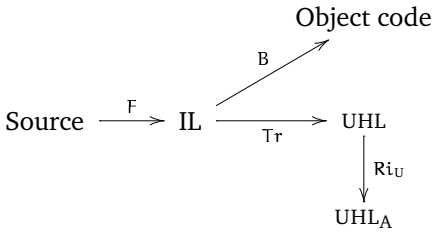
   $$\text{Source} \xrightarrow{\mathtt{F}} \text{IL} \xrightarrow{\mathtt{B}} \text{Object code}$$

   ($\mathtt{F}$ and $\mathtt{B}$ are the compiler's front and back end, respectively).

2. Write down the natural mapping $\mathtt{Tr}$ from the IL to our UHL. This ought to be relatively easy and consist mostly of making explicit the implicit invariants of the concrete intermediate language and abstracting operations that do not touch the heap.

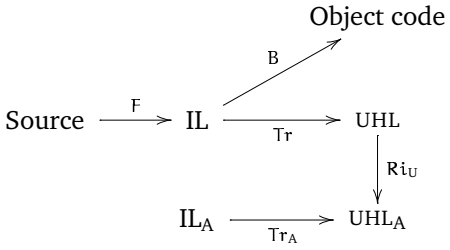
3. The following chapters of this thesis defines a general region inference $\mathtt{Ri}_{\mathrm{U}}$ which takes UHL to UHL-with-annotations, notated as $\mathrm{UHL}_{\mathrm{A}}$ in the diagram:

   In order that we can reason about the correctness of $\mathtt{Ri}_{\mathrm{U}}$, our general theory will include an "ideal semantics" for uniform mutators, which we will assume defines the programmer's intended behavior of his program (without any thought of memory management), and a "managed semantics" for $\mathrm{UHL}_{\mathrm{A}}$.

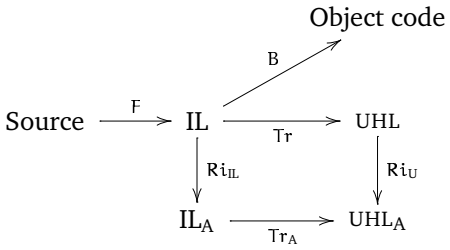
4. Now, because the translation $\mathtt{Tr}$ really just re-expresses the IL code in a different notation, each of the places in the uniform mutator where region annotations attach will correspond to a single place in the intermediate code. Therefore, from the annotation language for UHL we can derive an

annotation language for IL, and extend the translation to Tr_A which just carries the annotations unchanged to the UHL world.



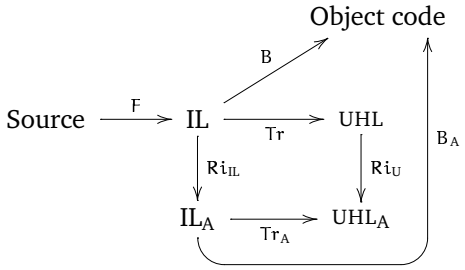
This derivation is an essentially non-creative process.

5. Derive from Ri a region-inference algorithm for the intermediate language that commutes with Tr and Tr_A:



This may sound harder than it actually is. Remember that Ri does not change the underlying uniform mutator; it just adds annotations to it. So we can consider Tr as a "view" of the intermediate language that is invoked incrementally whenever Ri wants to inspect its input. With appropriate code factorization, very substantial parts of the source code for Ri_{IL} can be reused from the region inference for a different intermediate language![5]

6. Extend the existing compiler back end B to translate region annotations along with the rest of the IL.

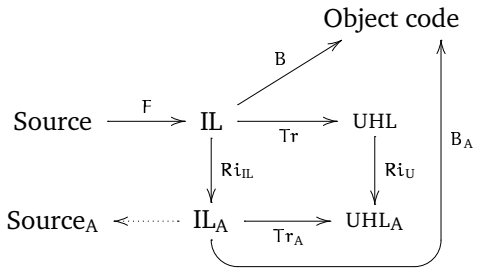


This step requires some creativity, but if an implementation without regions is available, the core of the burden will usually lie in integrating the region operations with the existing run-time support system. The changes in the compiler can be comparatively small – in the experiment where I

[5]That is, if the two compilers themselves are implemented in the same language. For this reason, the region inference in [Makholm and Sagonas 2002] was implemented as an external program written in Standard ML; about six thousand lines of code were reused with only minor changes from the prototype region inference developed for the experiments in [Henglein et al. 2001].

applied this theory to Prolog [Makholm and Sagonas 2002], it took me less than a week to add the ability to translate region annotations to a bytecode compiler that I had never seen before.

7. Optional, but important in practise: Mimic step 4 for the compiler front-end F instead of the translation Tr. This gives an annotated *source* language, which can be used to present the inferred region annotations in human-readable form:



It is a strong practical experience with region-based implementations that vastly better memory behavior can be achieved if the programmer gets a chance to observe the outcome of the region inference and possibly use the information to change the source program to become more region friendly. While the programmer does not need to understand the algorithmic details of the region inference, it is strongly desirable to have a source syntax for the region annotations and an execution model for regions that can be understood in terms of the source code.

Too aggressive code optimizations in the front end may well make this step rather difficult, but it is hard to say how difficult without practical experience to back it. More knowledge will be needed to say anything about this problem in general.

As seen from the final diagram, UHL programs will never need to actually exist in full in the implementation. This emphasizes the fact that UHL is not an implementation language; it is primarily a vehicle for moving region inference techniques between different host languages.

1.5 Overview

The thesis is structured according to the high-level decomposition of region inference I presented in Section 1.3.

Chapter 2 presents some background information on previous work about region inference.

Chapter 3 defines the universal host language and its associated agent programming language. It also proves some general facts about their semantics and introduces some useful ways of reasoning about agents.

Chapter 4 gives an example of how to translate a small ML subset into the UHL model. This shows how the UHL framework relates to the ML-based model of the "HMN system" of Henglein et al. [2001] (which is also described in Section 2.4).

Chapter 5 develops a region type system the works for the ML subset of Chapter 4.

Chapter 6 is a brief interlude that discusses how to generalize the techniques of Chapters 4 and 5 to a number of common language features that are not exhibited by the ML subset.

Chapter 7 presents algorithms for actually doing region inference, *i.e.*, constructing an agent for a given mutator.

Except for Chapter 2, the chapters are meant to be read in sequence. In particular, Chapter 3 defines the basic framework without which all of the rest will be unreadable.

My development is based on a few simplifying assumptions, which I have made in order to be able to get to the real problem. Hopefully later work will show how to put back in what I excluded.

Most importantly, I consider only region inference for entire programs. Most previously published region inference methods [Birkedal and Tofte 2001; Birkedal et al. 1996; Tofte and Birkedal 1998] are compositional enough that region inference for a multi-module program can be done one module at a time (Elsman [1999] uses this region inference as a central motivating example for his Ph.D. work on advanced recompilation management). The further optimization by Aiken et al. [1995] is by its nature nonmodular.

Finding a modular formulation for the region inference techniques I present here would of course be highly desirable, but at present it seems a long way off.

A less severe restriction is that I do not consider strings and arrays, where the memory needs of a single object cannot be determined statically. The ML Kit originally implemented strings as linked lists of segments with bounded size [Elsman and Hallenberg 1995, section 7.2.9] and arrays as multi-branching trees. Though such a representation is intuitively "correct", handling it in the formal reasoning in this thesis would be complex and offer little new insight. Later [Elsman 2003, personal communication] the ML Kit's strategy changed to allocating large objects outside the region heap and maintaining a linked list of such large objects for each region. This scheme means that deallocating a region is not a constant-time operation anymore. It would be straightforward to handle with the formal techniques of this thesis, however.

1.6 Notation and formal conventions

In the formal development, I use some convenient shorthand notations that are not completely standard. The underlying concepts should be well-known to any reader, so I just give a series of definitions to make the notation clear.

1.6.1 Set theory

Notation 1.1. *For an arbitrary set* A*,* $\mathcal{P}(A)$ *is the set of subsets of* A*, and* $\mathcal{P}_{\text{fin}}(A)$ *is the set of finite subsets of* A*.*

Definition 1.2. $\mathbb{W}$ *is the set of all finite mathematical objects.*

The precise definition of what "finite mathematical object" means is not important as long as $\mathbb{W}$ is large enough – it must contain the natural numbers as well as everything else that we're going to assume to be contained in it. In axiomatic set theory, $\mathbb{W}$ might be defined as the set of all sets with finite rank. An equivalent definition that looks more like computer science is that $\mathbb{W}$ is the least (actually unique) solution to the fixpoint equation $\mathbb{W} = \mathcal{P}_{\text{fin}}(\mathbb{W})$.

1.6.2 Maps, especially finite ones

Definition 1.3. *A **map** f is any set of ordered pairs such that $(x, y) \in f$ and $(x, z) \in f$ implies $y = z$. The expression $f(x)$ is defined iff $(x, y) \in f$ for some y; in that case $f(x)$ denotes this y.*

Notation 1.4. *When we describe a map by listing its elements explicitly, we shall generally use the infix symbol "$\mapsto$" to construct the ordered pairs.*

For example, "$\{a \mapsto 12, b \mapsto 5\}$" means the map f such that $\operatorname{Dom} f = \{a, b\}$, $f(a) = 12$, and $f(b) = 15$.

Notation 1.5. *We often use the compact notation $\left[\begin{smallmatrix} x_1 & \cdots & x_n \\ y_1 & \cdots & y_n \end{smallmatrix}\right]$ as an alternative notation for the map $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$. In this notation, the empty map is $[\,]$.*

Definition 1.6. *For any map f, its **domain** is $\operatorname{Dom} f = \{\, x \mid f(x) \text{ is defined} \,\}$, and its **image** is $\operatorname{Img} f = \{\, f(x) \mid x \in \operatorname{Dom} f \,\}$.*

Definition 1.7. *For arbitrary sets A and B, $A \to B$ is the set of all maps f with $\operatorname{Dom} f = A$ and $\operatorname{Img} f \subseteq B$.*

Definition 1.8. *For arbitrary sets A and B, $A \xrightarrow{\text{fin}} B$ is the space of **finite maps** from A to B, that is, the set of all maps f such that $\operatorname{Dom} f \in \mathcal{P}_{\text{fin}}(A)$ and $\operatorname{Img} f \subseteq B$.*

Note that we consider a finite map to be total from its own domain; it does not "know" which space we happen to consider it part of.

Definition 1.9. *For arbitrary sets A and B, $A \xrightarrow[\text{inj}]{\text{fin}} B$ is the space of **finite injective maps** from A to B.*

Definition 1.10. *If f and g are maps, then the **extension** $f \circledcirc g$ is the (finite) map defined by*

$$(f \circledcirc g)(x) = \begin{cases} g(x) & \text{if } g(x) \text{ is defined} \\ f(x) & \text{otherwise} \end{cases}$$

Definition 1.11. *If $\operatorname{Dom} f \cap \operatorname{Dom} g = \varnothing$, then the **joined map** $f \oplus g$ is the same as $f \circledcirc g$. Otherwise, $f \oplus g$ is undefined.*

It is easily seen that $\oplus$ is commutative and associative, and the empty map is a neutral element for it. On the other hand $\circledcirc$ is associative but not commutative.

Definition 1.12. *If f and g are maps with $\operatorname{Img} g \subseteq \operatorname{Dom} f$, then their **composition** $f \circ g$ is defined by*

$$(f \circ g)(x) = f\big(g(x)\big)$$

Definition 1.13. *The **strict composition** $f \bullet g$ is the same as $f \circ g$, except that we only consider it to be defined when $\operatorname{Img} g = \operatorname{Dom} f$.*

As usual, $\circ$ is associative but not commutative. This holds for $\dagger$ as well.

The two restricted operations $\oplus$ and $\dagger$ will allow us to state some side conditions implicitly in formulae. We follow the common convention that an (implicit or explicit) quantifier ranges over only those values for its variable that does not make a subformula within its scope undefined. Notably this includes the (often implicit) quantifiers in set comprehensions, and the implied set comprehension in the presentation of inference rules.

As a (contrived) example, the formula $\left\{f \dagger \left[\begin{smallmatrix} a & b \\ 3 & 8 \end{smallmatrix}\right] \mid f \in F\right\}$, where F is some set of maps, implicitly excludes all the $f \in F$ whose domain is not exactly $\{3, 8\}$.

When we define properties of objects, the property shall be considered not to hold if the evaluation of the defining property requires an undefined value (which cannot be caught by the quantifier rule). For example, imagine the following definitions:

> *The map f is **sweet** iff* $\mathrm{Img}(f \oplus \left[\begin{smallmatrix} 4 \\ 3 \end{smallmatrix}\right])$ *has an even number of elements.*
> *A map is **sour** iff it is not sweet.*

Then $\left[\begin{smallmatrix} 1 & 5 \\ 3 & 9 \end{smallmatrix}\right]$ is sweet and $\left[\begin{smallmatrix} 5 & 8 \\ 4 & 0 \end{smallmatrix}\right]$ is not. However, $\left[\begin{smallmatrix} 2 & 4 \\ 4 & 4 \end{smallmatrix}\right]$ is *also* not sweet, because $\left[\begin{smallmatrix} 2 & 4 \\ 4 & 4 \end{smallmatrix}\right] \oplus \left[\begin{smallmatrix} 4 \\ 3 \end{smallmatrix}\right]$ is undefined. Thus $\left[\begin{smallmatrix} 5 & 8 \\ 4 & 0 \end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} 2 & 4 \\ 4 & 4 \end{smallmatrix}\right]$ are both sour – the "sweetness value" of the latter is not itself undefined but a well-defined "no". [6]

1.6.3 Variables and metavariables

Notation 1.14. *Whenever a convention has been established that some (meta)-variable letter, say "x", ranges over a given set of objects, the notation $[\![x]\!]$ shall mean that set of objects.*

Of course, this convention is most useful when the range convention has been established without explicitly naming the range. This can happen with grammar rules such as

$$\begin{aligned} \text{Expressions: } e ::= {} & 0 \mid 1 \mid 2 \mid \cdots \\ \mid {} & e + e \mid e - e \mid e \cdot e \\ \mid {} & \mathsf{let}\ x = e\ \mathsf{in}\ e \end{aligned}$$

which defines $[\![e]\!]$ to be the set of expressions with the given form, while also establishing a convention that variables like e_5 and e'' range over this set.

• **History of notation:** The symbol $\dagger$ is my own invention. I don't think it's a very good symbol for this use. Suggestions for other ones would be appreciated.

[6] I'm fairly sure that this convention corresponds to standard mathematical intuition, but it does sound rather strange when spelled out in detail. The sweet/sour example shows that definitions in our mathematical metalanguage are not referentially transparent! If the definition of "sweet" is substituted into that of "sour", $\left[\begin{smallmatrix} 2 & 4 \\ 4 & 4 \end{smallmatrix}\right]$ would stop being sour.

Chapter 2

Previous approaches to region inference

In this chapter I give a quick summary of relevant previous work about region inference.[1] The chapter can be skipped without losing much of the technical points in the later chapters.

I do not purport to survey the entire field of region-based memory management, but I try to mention all previous work that concern *automatic* generation of agents. By this I mean work that strives at making it possible to run the region inference on legacy code that has been written without any thought of regions or memory management, and get a decent attempt at a sound agent for the program. Some of the techniques may do *better* if the user nudges them in the right direction by special annotations on the program; I don't hold that against them. But I do not consider "partial" region inference techniques that only work if the user supplies region invariants for all functions (or loops), and just fill in the blanks automatically.

In most of the cases, I will focus on the agent programming language that is inherent in the different techniques. This is because the algorithms that have been proposed are often designed to make full use of the expressive power of their corresponding agent programming language – so understanding the agent programming language will also lead to a feeling for the analytic power of the various techniques.

As a running example, we use a classic benchmark for region-based memory management: The *Game of Life* example, which has been used since the first articles by Tofte and Talpin. The task is is to simulate the "Game of Life" [Gardner 1970] for n generations, but the issues the examples raise apply to iterative computation in general.

The standard way of programming an iteration in a functional language is to use tail recursion:

[1] Part of the text in this chapter has been adapted from an article [Henglein et al. 2001] that I co-authored with Fritz Henglein and Henning Niss. It thus represents shared work, and it has been lost in the mist of time exactly who wrote what. I'm reasonably certain that it was me who wrote the original Game of Life example though.

The discovery that the Kit, AFL, and HMN systems do not always have globally optimal agents (Sections 2.2.3, 2.3.1, and 2.4.4) is among my original contributions to [Henglein et al. 2001].

fun *nextgen* (g) = ⟨read g; create and return new generation⟩
fun *life* (n, g) = **if** n = 0 **then** g
else *life*(n − 1, *nextgen*(g))

We shall leave the details of *nextgen* unspecified here and in the following discussion (see the Kit distribution for an implementation; ⟨`http://www.it-c.dk/research/mlkit`⟩). Furthermore we make the simplifying assumption that a single region holds all the pieces of a generation description, and ignore other non-essential details in the discussion.

2.1 Tofte–Talpin

As mentioned earlier, the first suggestion to automate agent creation was Tofte and Talpin [1994, 1997]. This is still the one that is commonly meant when articles refer to "*the* region calculus" or "*the* region inference algorithm". In the following, we will call it the "TT system", for short. It works for the polymorphically typed call-by-value lambda calculus – *i.e.*, "ML".

Let us introduce its agent programming language by looking at an annotated version of the Game of Life example:

fun *nextgen* **[**ρ**]** (g) = ⟨read g from ρ; create new gen. **at** ρ⟩
fun *life* **[**ρ_a, ρ_g**]** (n, g) =
if n = 0 **then** g
else letregion ρ'_a
in *life* **[**ρ'_a, ρ_g**]** (n − 1, *nextgen* **[**ρ_g**]** (g)) **at** ρ'_a

The region annotations here are

... **[**ρ**]** ... ⟨... from ρ ... **at** ρ⟩
... **[**ρ_a, ρ_g**]** ...
letregion ρ'_a
in ... **[**ρ'_a, ρ_g**]** ... **[**ρ_g**]** ... **at** ρ'_a

The most conspicuous feature of the region annotations is a new class of **region variables** ρ, syntactically different from ordinary variables, and a new expression form that was not present in the original Game of Life program:

Expressions: e ::= ...
| **letregion** ρ **in** e

When evaluated, a letregion expression allocates a new region from the region manager and binds its handle to the region variable ρ within e. After e has been evaluated, the region is deallocated.

Expression forms that allocate memory – such as the construction of the argument pair before the recursive call to *life* – are annotated with an "**at** ρ" construction that specifies which region to allocate the memory from. In the case of the argument pair, the allocation is "**at** ρ'_a", so the pair is allocated in the region constructed by the immediately enclosing letregion expression. Similarly, we assume that the body of *nextgen* contains a number of allocations for the description of the next generations; the abbreviated sketch contains "**at** ρ" to indicate that all of these allocations happen in the region called ρ.

Box 2.1—Tofte–Talpin and the "stack of regions"

The TT model is also known as the "stack-of-regions" model, because the fact that the letregion expressions are aligned with the mutator's expression structure and in particular with each other. Therefore, the lifetimes of the actual regions at runtime will collectively follow a FIFO pattern: Only the newest existing region can ever be deallocated. Furthermore, this stack of regions grows and shrinks in unison with the host language's call stack, so a region manager tailored for the stack-of-regions model can use the call stack to store data about regions.

This property figured prominently in the title of Tofte and Talpin's POPL article [1994], which presented region inference as a scheme for stack-allocating rather than heap-allocating all allocated data. This has the less fortunate effect that some readers have thought the stacking of regions to be an integral part of the region model. I think differently; so there has been no mention of a stack of regions anywhere in Section 1.1. The region stacking in the TT model is an artifact of its agent programming language, not a general property of regions.

But what is ρ in *nextgen*? It gets passed as a **region parameter** from the call in *life*. Region parameters are notated with square brackets on every function definition and function call in the annotated program; they are passed separately from the mutator-level parameter, and the agent programming language supports passing any number of region parameters without tupling them together. (Remember that the agent piggy-backs its control flow on that of the mutator, so functions in the host language correspond to functions in the agent language). In the example we see that ρ in *nextgen* is the same as ρ_g in *life*, which itself is the same as the ρ_g of the enclosing recursive instance, and so forth – in the end the region where the new-generation data gets allocated is one that is specified by the (unseen) main program that calls *life* in the first place.

With these explanations, we can read what the region annotations mean: The main program creates a region to pass as ρ_g; as the iteration proceeds more and more intermediate generations get allocated in this region, and only when the main program is done with reading the final result (showing it to the user, presumably) can the region be deallocated and the memory occupied by the intermediate generations made available for reuse. But then the program is about to terminate! Clearly this is not an example of successful memory management.

It gets worse: Each of the letregion expressions only deallocate its region when its body has been fully evaluated – that is, when the recursive call to *life* returns. So just before the bottommost call returns, we have a stack full of ρ'_a regions just waiting to be deallocated. It is easy to see that this fate is inevitable for any region that contains part of the argument in a tail call. And worse yet: Because the letregion needs to do things to deallocate its region, the body of a letregion is never at tail call context, so the recursive call to *letregion* is not a tail call, such as was the case in the original, unannotated program.

Does this mean that the TT system is worthless? Of course not, or the original article would not have had the impact it had. Rather, I have deliberately chosen an example program that exhibits some weaknesses in the TT system, such that I can tell a story of how these weaknesses have been overcome by later work. There are plenty of examples where the TT system does give rise to nontrivial

memory reuse.

2.1.1 Regions and closures

One feature of the TT model that the Game of Life example did not show is that its region variables are lexically scoped, just like ordinary variables in the lambda calculus. If there is a lambda abstraction sandwiched *between* the letregion expression and the allocating expression that is annotated with its variable, the agent will need to store its region handle in the mutator's closure when the lambda is executed, and retrieve it again when the function is called. Thus, it was not completely faithful to the TT system when we claimed in Section 1.1 that the agent cannot observe the mutator's data structures. It doesn't observe them directly, but it does sometimes store its own private data in them.

This feature has the somewhat unexpected side effect that the TT model seems to be strictly stronger for programs written in a "pure lambda calculus" style, where lambdas are used to build data structures, than for programs that manipulate first-order data. For example, it is possible to build an arbitrarily long chain of closures, each referencing the next as the value of a free variable, and containing its own set of region handles for the local data in the closure. This is not possible with first-order data, because the region type system forces all elements in a list (or another recursive datatype) to be in the same region.[2]

The downside of this added flexibility is that the presence of region handles inside closures vastly complicates the region inference process. The algorithms that are known for the TT system (see below) are very complicated and not yet quite fully understood, exactly due to the problem of taming such "dormant" regions.

In this dissertation, my main goal is generalizing region inference to other languages, many of which do not have lambda abstractions or encourage their use only sparingly. Therefore I have decided *not* to try to duplicate the strength of regions-within-closures from the TT system in the region inference methods I present here.[3]

2.1.2 Region polymorphism

Another peculiarity of the TT system that is not demonstrated by the Game of Life example is how region parameters work in connection with "functions as values". First, a function can take region parameters only if it is declared with Standard ML's "fun" keyword – that is, if it it immediately being bound to a name, rather than just being an anonymous lambda. Second, the actual region parameters must be specified whenever the function's *name* is used, even if it is not for calling it immediately. If the call does not happen immediately after giving the region parameter, a special closure must be allocated to store the actual region parameters, and therefore giving region parameters is an allocating operation that must have an "**at**" annotation of its own.

[2]The linear region discipline by Walker and Watkins [2001] provides a certain amount of this region flexibility for first-order data, but is not designed as a basis for automatic region inference and seems wholly unsuited for that role.

[3]However, it would be an interesting direction for future work to attempt to achieve this strength within the general framework of my model; see Section 3.5.1.

Thus, an *expression* of function type can never evaluate to something that still expects region parameters. Therefore a function that takes region parameters cannot be passed as argument to another function (or stored in a data structure) in its full generality - it will have to have its region parameters passed before, after which the partially applied function is indeed a first-class value.

The oddities of region parameters in the TT system derives from the fact that Tofte and Talpin devised them as a way to have the region-annotated *type* of a function differ between call points, so they modeled them over Hindley-Milner-style type abstractions rather than ordinary parameter passing. In much of the literature, the passing of region parameters is therefore known as **region polymorphism**. However, region parameters differ from ordinary Hindley-Milner type abstractions in that a region-parameter value must also be passed when a (recursive) function mentions its own name in its body. This is in contrast to the usual rule that a recursive function cannot call itself in a different polymorphic instance, and goes by the name **region-polymorphic recursion**.

2.1.3 The TT region type system

The original articles by Tofte and Talpin [1994, 1997] *almost* contained a region inference algorithm. The "almost" is because they contained a formal inference system that described the relation between the input to region inference (the mutator, in my terminology) and the output (an annotated mutator). The only problem was that this inference system is non-deterministic in a non-trivial way, so it cannot really be used as an algorithm. What is left is a *region type system*; Tofte and Talpin proved that if we *somehow* find an agent that happens to match the inference system, then that agent will be sound. (Afterwards, several alternative proofs for the safety of this region type system have appeared [Banerjee et al. 1999; Calcagno et al. 2002; Calcagno 2001; Dal Zilio and Gordon 2000; Helsen and Thiemann 2000]).

The basic idea behind the TT region type system is to take the ordinary ML types for the mutator and decorate the syntax trees for types with the region variables that are bound to the regions that contain the run-time data that correspond to the particular places in the syntax tree. This principle will be found again in the HMN region type system in Section 2.4.3 and in the UHL region type system we will develop in Chapter 5.

Because the regions are specified indirectly (the types contain the region *variables* that will be bound to the actual regions), the types need to change if the region variables change – such as when the (mutator-level) parameter to a function is located in regions that have one name in the caller (the actual region parameters) and another one in the callee (the formal region parameters). This is the basis for the view of region-parameter passing as a form of type polymorphism.

I will not describe the TT type system in detail;[4] most of its intricacies are concerned with keeping track of regions stored in closures. The only point that is important in this context is its rule for letregion expression. Somewhat simplified and paraphrased from the original TT syntax, it is

[4]Henglein et al. [2005] give a cleaned-up presentation of the TT region type system which we hope is easier to read than Tofte and Talpin's original articles.

$$\frac{\rho \notin FRV(\Gamma) \qquad \Gamma \vdash e : \mu \qquad \rho \notin FRV(\mu)}{\Gamma \vdash \textbf{letregion } \rho \textbf{ in } e : \mu}$$

where μ is the variable letter for region-annotated types and *FRV* picks out all the free region variables in a type or set of types. The rule says that a region may only be deallocated at a point where it is not present in the type of the value that was computed while the region existed. (The "$\rho \notin FRV(\Gamma)$" side condition is there to prevent name capture of ρ). Because types describe the structure of values, it is normally the case that "not present in the type" coincides with "not reachable via pointers", so one concludes that a TT-certified agent will never deallocate something that a tracing garbage collector would consider live.

But this is only *normally*! A closure can contain pointers to values that are not described by the type of the closure itself. When this is the case, it is indeed possible for the TT region inference to deallocate something that a tracing garbage collector considers live. This will be safe because the region type system tracks the use of regions. If the code part of the closure may actually need to access the data in question, its region must appear in the "effect" part of the closure's type, which will prevent the letregion rule for matching.

This ability to create structures with "dangling pointers" and still stay (provably) safe helped fuel much of the initial interest in region-based memory management. In my opinion, however, it looks more like a gimmick than an actually useful feature – it only applies when one uses intrinsically higher-order programming, and sometimes needs non-obvious coding styles. A large part of the motivation for the development of the HMN system (Section 2.4) was our wish to make the gimmick real for first-order programs.

2.1.4 Algorithms for TT region inference

Eventually an actual region-inference algorithm for the TT system was published by Tofte and Birkedal [1998]. It works by trying to construct a typing in the TT region type system for the mutator by something like Milner's Algorithm $\mathcal{W}$, treating region parameters as a form of polymorphism. Region-polymorphic recursion is handled by a local ("Kleene–Mycroft") fixpoint iteration plus some ad-hoc heuristics to ensure termination.

I will not discuss the algorithm in detail; it is unrelated to my own region inference algorithms. A significant source of complexity in the algorithm is the need to extend the unification of Algorithm $\mathcal{W}$ to the set-shaped "effects" the the TT region type system uses to track dormant regions in closures. This complexity is part of the reason why in my own work I have tried to avoid regions-in-closures.

Later, Birkedal and Tofte [2001] reformulated their algorithm in terms of constraints such that the ad-hoc heuristics for termination could be eliminated. The new algorithm was proved complete modulo some technical requirements on the shape of the typing it produces. If these requirements are as innocuous as they look, the new algorithm is "optimal" in the sense that the agent it produces will give rise to minimal lifetimes among all agents that pass a generalization of the TT region-type system. However, an actual proof of this has not been forthcoming, and at the moment it is unknown whether such an optimal agent always exists.

2.1.5 Other region inferences based on TT

In my M.Sc. work [Makholm 2000b] I derived a region type system for a subset of Prolog from the ideas behind the TT system. Because Prolog is a first-order language, I could dispense with the effect mechanisms that TT uses for closures. I developed and described a similarly simplified region-inference algorithm for this subset. This algorithm is unrelated to the one by Tofte and Birkedal, as well as to the one I present in Chapter 7, though some structural ideas from it can be found in Section 7.3.8. The prototype I developed also contained the first known (to me) implementation of a region-merging postphase; it later evolved into the one I describe in Section 7.4.4.

2.1.6 Unused region parameters

The Game of Life example shows one further peculiarity, namely the region parameter "ρ_a". This gets passed to every instance of *life* but is always ignored. The TT region inference produces this essentially useless parameter because it is necessary to convince the TT region type system that it is safe to extracts the two elements of the argument tuple. Therefore the region inference invents the invariant that ρ_a must always be the region that contains the argument tuple; but region polymorphism is so closely tied with parameter passing that ρ_a must actually appear in the parameter list.

At run-time, it is of course not necessary to pass this region parameter around, so the ML Kit (which, as far as I know, is the only production-quality language implementation based on the original TT model) includes a postphase that removes unused region parameter. It is called **get-region removal** and is briefly described by Birkedal et al. [1996][5]. The name stems from the fact that the removal works from data collected during the main region inference, which distinguishes between region parameters that the function "puts" something into (*i.e.*, allocates in), and ones where it only "gets" data from the heap.

2.2 The ML Kit

The ML Kit [Tofte et al. 1997, 1998, 2002] is the most famous implementation of the TT region system. Its first version [Birkedal et al. 1993] was little more than a rendering of the semantics in the Definition of Standard ML as modular, executable ML code, intended to be used for experiments with compiler technology. The grandest such experiments was the implementation of region-based memory management. In version 2 of the Kit, TT region inference and region-based memory management was added, and the subsequent development has aimed at making the ML Kit a production-quality Standard ML compiler with region-based memory management. Version 4 of the Kit adds a garbage collector that coexists with the region-based memory manager [Hallenberg et al. 2002; Hallenberg 1999].

The extension of region-based memory management from the toy language in Tofte and Talpin's original articles to the full Standard ML language entailed a lot of additional development. Some of the innovations – such as the handling of strings, datatypes or exceptions – have not been published and seem

[5] For unknown reasons, this postphase is described as being part of the multiplicity analysis...

to be available only as internal documentation in the Kit distribution and "by the way" comments in its user manual. Few of these have been the subject of stringent formal investigation; apparently the reasoning has been that the TT region type system was formally known to work, so its (more or less) "obvious" generalizations to other settings could be trusted by intuition. Such a practise is of course not unreasonable; formal reasoning on the scale of an entire general-purpose programming language borders on the infeasible anyway. But still it is somewhat disquieting how much of the Kit's region model rests of the principle of Proof By Being Unable To Think Of A Reason Why It Shouldn't Work.

I will limit my description of the ML kit to the main features of its region implementation that are most relevant to my work in general.

2.2.1 Storage modes and region resetting

For our purposes, the most important feature of the ML Kit is its response to the basic problem we have seen with the TT system: The hierarchical nesting of region lifetimes prevents reasonable reuse of memory when tail recursion is used, as in the Game of Life example.

The ML Kit's solution, described by Birkedal et al. [1996] is based on the concept of **resetting** a region. Resetting a region means deallocating all data in the region, while not deallocating the region itself. Another way to view this (in our general model of region-based memory management) is that the region is deallocated and a new one is created and bound to the region variables that used to reference the old one – but the Kit's region manager has an optimized implementation of the operation that does not need any region variables to actually be updated.

The agent programming language is extended with constructs to request resetting of a specific region. Syntactically these constructs are expressed as a refinement of the "**at**" annotations on allocating expressions. Instead of just "**at**" they now say "**atbot**", meaning first reset the region and then allocate, or "**attop**", meaning just allocate. "**bot**" and "**top**" are called **storage modes**. It is possible to have a region reset without allocating something in it at the same time, but the ML Kit will only do this as the result of a programmer-supplied annotation. Systematically, this annotation can be handled as a request to allocate zero cells **atbot** in the region.

Storage modes are inserted by a **storage-mode analysis** which runs after the ordinary TT region inference. The storage-mode analysis builds on the types from the TT region type system. It works by doing a simple liveness analysis on the variables in the ML source and allowing an allocation in ρ to be **atbot** if no variable (or unnamed intermediate result) with a ρ in its type is live. This scheme seems intuitively safe, but no actual proof of its safety has appeared.

The ability to reset regions offers the hope of achieving better memory reuse than in the original TT model with its strictly nested region lifetimes. However, it does not work all by itself but needs the mutator to be written in a special style. The ML Kit manual recommends rewriting the Game of Life example in the following way:

fun *copy* $(g) = \langle$read g; make fresh copy$\rangle$
fun *life'* $(a$ **as** $(n, g)) =$ **if** $n = 0$ **then** a

$$\textbf{else } \mathit{life}'(n-1, \mathit{copy}(\mathit{nextgen}(g)))$$
$$\textbf{fun } \mathit{life}\,(a) = \#_2\ (\mathit{life}'\ a)$$

where *copy* (whose body is omitted here for brevity) takes apart a generation description and constructs a fresh, identical copy. The ordinary TT region inference will then produce

$$\begin{array}{l}
\textbf{fun } \mathit{nextgen}\ [\rho, \rho']\,(g) = \ \langle \text{read } g \text{ from } \rho\text{; new gen. } \textbf{at } \rho' \rangle \\
\textbf{fun } \mathit{copy}\ [\rho', \rho]\,(g) = \ \langle \text{read } g \text{ from } \rho'\text{; create fresh copy } \textbf{at } \rho \rangle \\
\textbf{fun } \mathit{life}'\ [\rho_a, \rho_g]\,(a\ \textbf{as}\ (n, g)) = \\
\quad \textbf{if } n = 0\ \textbf{then } a \\
\quad \textbf{else } \mathit{life}'\ [\rho_a, \rho_g](n-1, \\
\qquad\qquad \textbf{letregion } \rho'_g\ \textbf{in } \mathit{copy}\ [\rho'_g, \rho_g]\ \ (\mathit{nextgen}\ [\rho_g, \rho'_g]\ (g)) \\
\qquad\qquad)\ \textbf{at } \rho_a \\
\textbf{fun } \mathit{life}\ [\rho_a, \rho_g]\ a = \#_2\ (\mathit{life}'\ [\rho_a, \rho_g]\ a)
\end{array}$$

At first sight, this seems to be *worse* than the pure TT solution – now all of the argument pairs are in the *same* region ρ_a. This is because splitting *life* into two functions means that *life*$'$, which contains the recursive case, will be forced to obey the invariant that the result is always in the same regions as the argument. This again forces the argument to the recursive call to be in the same regions as the arguments to the enclosing instance – and that eliminates the letregion expression that prevented the recursive call to be a tail call.

Now, however, the stage is set for storage-mode analysis. In this particular example it identifies all the allocations as **atbot**, so ρ_a will be reset each time a new argument tuple is allocated, and ρ_g will be reset once for each generation:

$$\begin{array}{l}
\textbf{fun } \mathit{nextgen}\ [\rho, \rho']\,(g) = \ \langle \text{read } g \text{ from } \rho\text{; new gen. } \textbf{attop } \rho' \rangle \\
\textbf{fun } \mathit{copy}\ [\rho', \rho]\,(g) = \ \langle \text{read } g \text{ from } \rho'\text{; create fresh copy } \textbf{atbot } \rho \rangle \\
\textbf{fun } \mathit{life}'\ [\rho_a, \rho_g]\,(a\ \textbf{as}\ (n, g)) = \\
\quad \textbf{if } n = 0\ \textbf{then } a \\
\quad \textbf{else } \mathit{life}'\ [\rho_a, \rho_g](n-1, \\
\qquad\qquad \textbf{letregion } \rho'_g\ \textbf{in } \mathit{copy}\ [\rho'_g, \rho_g]\ \ (\mathit{nextgen}\ [\rho_g, \rho'_g]\ (g)) \\
\qquad\qquad)\ \textbf{atbot } \rho_a \\
\textbf{fun } \mathit{life}\ [\rho_a, \rho_g]\ a = \#_2\ (\mathit{life}'\ [\rho_a, \rho_g]\ a)
\end{array}$$

Note that the resetting of ρ_g is possible only because of the seemingly redundant *copy* operation. If the result of *nextgen* had been used directly, *nextgen* would have had to produce its result in the same region as it got its input, and then it would not have been safe to reset it (because, let us assume, that *nextgen* needs to be able to allocated the first parts of its output before it as inspected its entire input).

This solution does make it possible for *life* to run in constant space (assuming that the size of a single g is bounded), but it is by no means obvious that precisely these changes to the original program would improve the space behavior. Furthermore, inserting such region optimizations in the program impede maintainability because they obscure the intended algorithm. Even though the programmer who makes the changes may understand regions well enough to see how the improvement works, the program certainly gets more difficult to *read* for the next, less region-literate, programmer. Finally, the CPU cycles spent on *copy* may be considered "waste" because they do not contribute to the computation *per se*.

2.2.2 Storage-mode polymorphism

The above explanation of storage modes is a little simplified. In order to reset a region variable bound as a formal region parameter, it is necessary that the region variable *itself* does not occur in any free variables at the point of the reset itself. But must also be true that the region variable in the caller that was used as the *actual* parameter did not occur free at the time of the call.

The storage-mode analysis in the Kit does not know the call graph of the program (it works for one module at a time), so the Kit resolves such matters at run time. Whenever a region parameter is passed, the caller also passes a bit that says whether the called function is allowed to reset the region. If not, **atbot** allocations in the callee will work as if they were **attop**. As a syntactic reminder of this, the annotation **atbot** is not actually used for region parameters; instead it is spelled **somewhereat**. This indicates that the actual storage mode is selected by the caller; the concept is called **storage-mode polymorphism** by Birkedal et al. [1996].

2.2.3 Storage modes and region aliasing

Having region parameters in function definitions potentially leads to **region aliasing**. For example, in a (hypothetical) call $\mathit{life}'\ [\rho, \rho](n, g)$, the region variables ρ_a and ρ_g in the function body will denote the same region. Therefore, one has to be careful when resetting regions. The analyses in the ML Kit solve this problem by conservatively approximating the set of aliased region variables [Tofte et al. 1998, Section 12.2] and prohibiting the resetting of possibly-aliased regions.

(It seems to me that a less conservative but still safe principle would be to use storage-mode polymorphism to prevent resetting of aliased regions. If the caller that initiates the aliasing passes both (all) aliases **attop**, the callee will be prevented from resetting the region *when called from that caller* but could still contain **somewhereat** annotations and reset regions when called from non-aliasing callers. This idea would "fix" problem of optimal agents for the counterexample given below. But it would not fix it in general; more complex counterexamples can be constructed where there are still trade-offs of this kind.)

The Kit's handling of region aliasing has the surprising effect that in the Kit system there is not always an "optimal" agent for a given mutator (even though there may be in the TT system, as I speculated in Section 2.1.4). Namely, consider the following program:

let fun f n = **let** p = (*sqrt*(5.0), *sqrt*(7.0)) **in if** n = 0 **then** p **else** f (n − 1)
  fun g n = **let** (x, y) = f n ; z = x :: y :: [ ] **in** ⟨use z⟩
in let x = #1 (f N) **in** ⟨space-critical section that uses x⟩

where we assume that *sqrt* function needs to heap-allocated its return value (imagine that a real is larger than a machine word).

The function g is never called but must still be typed. The list construction means (by the rules of the TT region type system) that x and y must have the same region annotations. Thus g must be able to call f in such a way that the components of the pair that f returns are guaranteed to be in the same region. Ignoring (for brevity) the region where the *pair* containing 5_B and 7_B is

allocated, there are two different ways to (TT) region annotate f which allow this:

a. f takes two region parameters and allocates $\sqrt{5}$ in one and $\sqrt{7}$ in another. When called from g, these two region parameters can be aliased.

b. f takes a single parameter where it allocates both square roots.

In the TT system this is not a problem, because there is no reason to chose (b) rather than (a). But with the Kit's resetting extension things change. Now (b) has the advantage that the single region parameter is not aliased in the call from g, so the storage-mode analysis will be free to allocate the first square root **somewhereat** instead of **attop**. This means that at no point will more than two of the square roots actually take up place on the heap. Therefore, if N is greater than 1, (b) will be the most memory-efficient solution.

On the other hand, if N happens to be 0, (a) is more desirable, because there will be no loss from not being able to reset, and the single $\sqrt{7}$ that the program allocated can be deallocated before the space-critical section.

If N is not known statically (it may be read in an I/O operation at runtime), this means that neither (a) nor (b) will be clearly superior to the other – no matter what we chose, there is a risk that the other choice would sometimes lead to a better memory behavior of the program.

2.3 Aiken–Fähndrich–Levien

Aiken et al. [1995] proposed an extension of the TT system in another direction, decoupling dynamic region allocation and deallocation from the introduction of region variables in the interpretation of the **letregion** construct.

In this system, which we call the AFL system, entry into a **letregion** block introduces a region variable, but does not allocate a region for it. During evaluation of the body of **letregion**, a region variable goes through precisely three states: unallocated, allocated, and finally deallocated. After a TT region inference, a constraint-based analysis – guided by a higher-order data-flow analysis for region variables – is used to insert explicit region allocation and deallocation commands which update the state of the region variables. The allocation and/or deallocation points can be inside functions that have been passed the region variable as a region parameter.

Aiken et al. presented their system as an extension to the Kit system (with resetting), but it can equally well be viewed as an independent and orthogonal extension to the basic TT system. The two extensions (AFL and resetting) are useful in different situations, but there are also areas of overlap. For example, the AFL system provides a solution of its own to the Game of Life problem:

In the AFL system the Life example can be improved by rewriting the original program to

fun *copy* (g) = ⟨read g; make fresh copy⟩
fun *life* (n, g) = **if** n = 0 **then** *copy*(g)
else *life*(n − 1, *nextgen*(g))

where the only difference from the original program being that the base case returns a fresh copy of its input rather than the input itself. Ordinary TT region inference results in the following TT annotations:

$$
\begin{array}{l}
\textbf{fun } \textit{nextgen}\ [\rho, \rho']\ (g) = \langle \text{read } g \text{ from } \rho\text{; new gen. } \textbf{at } \rho' \rangle \\
\textbf{fun } \textit{copy}\ [\rho, \rho']\ (g) = \langle \text{read } g \text{ from } \rho\text{; fresh copy } \textbf{at } \rho' \rangle \\
\textbf{fun } \textit{life}\ [\rho_a, \rho_g, \rho']\ (n, g) = \\
\quad \textbf{if } n = 0 \textbf{ then } \textit{copy}\ [\rho_g, \rho']\ (g) \\
\qquad\qquad \textbf{else letregion } \rho'_a, \rho'_g \\
\qquad\qquad\quad \textbf{in } \textit{life}\ \ [\rho'_a, \rho'_g, \rho']\ (n-1, \textit{nextgen}\ [\rho_g, \rho'_g]\ (g))\ \textbf{at } \rho'_a
\end{array}
$$

The introduction of *copy* in the base case means that the region where the input to *life* need not be the same as the one where the final result is allocated. Therefore, in the recursive case, the new generation can be allocated in a new region ρ'_g of its own.

This still looks bad from a TT viewpoint, but then the AFL post-analysis adds allocation and deallocation points and produces[6]

$$
\begin{array}{l}
\textbf{fun } \textit{nextgen}\ [\rho, \rho']\ (g) = [\![\textbf{alloc } \rho']\!]\ \langle \text{read } g \text{ from } \rho\text{; new gen. } \textbf{at } \rho' \rangle\ [\![\textbf{free } \rho]\!] \\
\textbf{fun } \textit{copy}\ [\rho, \rho']\ (g) = [\![\textbf{alloc } \rho']\!]\ \langle \text{read } g \text{ from } \rho\text{; fresh copy } \textbf{at } \rho' \rangle\ [\![\textbf{free } \rho]\!] \\
\textbf{fun } \textit{life}\ [\rho_a, \rho_g, \rho']\ (n, g) = \\
\quad [\![\textbf{free } \rho_a]\!]\ \textbf{if } n = 0 \textbf{ then } \textit{copy}\ [\rho_g, \rho']\ (g) \\
\qquad\qquad\qquad \textbf{else letregion } \rho'_a, \rho'_g \\
\qquad\qquad\qquad\quad \textbf{in } \textit{life}\ \ [\rho'_a, \rho'_g, \rho']\ (n-1, \textit{nextgen}\ [\rho_g, \rho'_g]\ (g) \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![\textbf{alloc } \rho'_a]\!]\ )\ \textbf{at } \rho'_a
\end{array}
$$

Because deallocation of each region is done explicitly and not by **letregion**, the body of **letregion** *is* a tail call context, and the regions containing the argument pair and the old generation can be freed the arguments have been matched and and *nextgen* (g) have been computed, respectively. Without rewriting the original program this would not be the case, because a function must either *always* free one of its input regions or *never* do it.

The AFL analysis is inherently non-modular; it needs a whole program to analyze at once. The system does come with a safety proof for the analysis, but it is not framed as a type system, so it is unclear whether compact certificates of a particular agent's safety can be produced.

2.3.1 Region aliasing and the AFL system

The AFL system's handling of region aliasing is more ambitious than that of the Kit system. The analysis computes the exact[7] set of possible aliasing situations for each function with region parameters. For example, one function may be marked:

> "This function may be called with no aliasing among the parameters, *or* with ρ_1 and ρ_2 aliased (and no other aliasing), *or* with ρ_2 and ρ_3 aliased (and no other aliasing).

Then, the main algorithm makes sure that the **alloc** and **free** annotations makes sense for *each* of the computed aliasing instances. For example, if an annotation

[6] The syntax here is not identical with the one used by Aiken et al.; for example, they write "**free_after** ρ e" for what we write as "e $[\![\textbf{free } \rho]\!]$".

[7] The analysis is "exact" relative to a previously computed call graph and an assumption that all call paths in the graph are possible.

to free ρ_2 is inserted, neither ρ_1 nor ρ_3 can be freed in that function, because a region must be freed exactly once.

This interplay between aliasing and annotations means that the combination of TT+AFL, taken as a whole, does not always have "optimal" agents, in the same way that optimal agents does not exist for TT+Kit (Section 2.2.3). Namely, consider this example program:

```
let fun f (x, y) = let z = x + y in ⟨space-critical section⟩
    fun g x = f(x, x)
in let x = sqrt(5.0) ; y = sqrt(7.0)
   in if ⟨something⟩ then f(x, y)
                     else ⟨space-critical section that uses x⟩
```

Again, the call from g means that it must be possible to annotate f such that its two arguments are in the same region. And again, there are two TT annotations that achieve this:

a. f takes two region parameters, expecting x to be in one of them and y to be in the other.

b. f takes a single region parameter containing both of x and y.

In variant (a) the two region parameters must be aliased in the call from g. This means that f can be allowed to free only one of them early, lest it would try to free the same region twice when called from g. Thus in the call from the main program, either x or y must stay allocated during the space-critical section within f.

Variant (b) does not have that problem, but would on the other hand require the main program to allocate $\sqrt{5}$ and $\sqrt{7}$ in the same region, keeping both allocated during the space-critical section in the **else** branch.

This kind of trade-offs in the Kit and AFL systems has apparently not been reported in the literature before the HMN article [Henglein et al. 2001]. Their existence questions the fundamental design approach of these systems, which is to build on a classical TT region inference. An unspoken assumption here is that region annotations produced by a region inference that aims at a pure TT execution model will also be "good" as a baseline for more advanced execution models. Our examples show that in some situations the memory behavior can be better if one uses a baseline that would be inferior in a pure TT setting.

2.4 Henglein–Makholm–Niss

The HMN (Henglein-Makholm-Niss) system is the basis for the generalization to multiple languages I present in this thesis. It was developed jointly by myself, Fritz Henglein, and Henning Niss in 2000–2001. Our starting point was to take a fresh look at region inference and create a system that (a) made the dangling-pointers gimmick from TT real for first-order data, and (b) include a solution to the TT system's problems with tail recursion in the region type system itself, rather than as postphases.

Thus, the HMN system comprises a novel agent programming language, a novel region type system, and some region inference algorithms. The first two items were presented in Henglein et al. [2001], but save for a very sketchy

description in that article, the present dissertation is the first written account on how our region inference algorithms for HMN work.

From the beginning, the HMN system was conceived as being applicable to different host languages. We did the initial development for a small imperative (but value-oriented) calculus, later to be published as the REGWHILE calculus of Niss [2002]. Before publication, we reexpressed the model with an ML subset as its host language; partly to emphasize the relevance to a conference whose deadline we then hoped on meeting, partly to emphasize its similarities and differences from the TT system.

This history means that HMN handles a smaller subset of ML than the TT system does: It support neither higher-order functions nor polymorphic typing (at the mutator-level). Those restrictions also carry over to my generalization in this thesis.

There are two central ideas of the HMN system. First: The agent programming language should be an *imperative* language even in cases where the host language is functional. Second: The agent maintains for each region a *reference count*, counting how many region variables are bound to the region. According to one's temperament, this can either be viewed as a more refined interface to the region manager or something that happens in the agent itself – or even a special reference-counting layer between the agent and the region manager. (In practise, reference counts are maintained by a specialized region manager).

The imperative agent programming language has four basic operations:

- $[\![\textbf{new}\ \rho]\!]$: allocates a new region with reference count 1 and assigns it to ρ; ρ must not be assigned a region before this.
- $[\![\textbf{release}\ \rho]\!]$: decrements the reference count of the region assigned to ρ and then makes ρ unassigned; the region is deallocated if (and only if) the reference count drops to 0.
- $[\![\rho' := \textbf{alias}\ \rho]\!]$: assigns the region in ρ to ρ' and increments its reference count; ρ' must be unassigned and ρ must be assigned a region.
- $[\![\rho' := \rho]\!]$: this **renaming** operation is equivalent to the sequence $[\![\rho' := \textbf{alias}\ \rho]\!]\ [\![\textbf{release}\ \rho]\!]$. It has no net effect on the reference count of the region originally bound to ρ. (One can think of this as a *linear* variant of region assignment in the previous case.)

Region operations can be inserted at arbitrary places in the mutator. When the host language is functional, as is the case in the formal development later in the section, this means that one or more region operations can be attached to each expression as a pre- or post-operation. Thus, for example, an effect similar to the TT system's "**letregion** ρ **in** e" can be simulated in HMN by "$[\![\textbf{new}\ \rho]\!]\ e\ [\![\textbf{release}\ \rho]\!]$", but **new** and **release** operations are not required to match up in this way.

The internal ordering of region operations is not subject to any hierarchical discipline. The "scope" of a region variable, if one insists on using such a concept, reaches from when it is assigned using a **new**, **alias**, or renaming operation and until it is consumed by **release** or renaming. The scope need not have a "nice" shape (the same region variable can toggle arbitrarily back and forth between bound and unbound), and there are no ties between the scope of different region variables.

The two exceptions to the general rule that the region sublanguage is independent of the structure of the base language are conditionals and function calls.

In a conditional, the two branches of a conditional must lead to the same region variables being assigned at the end, but they need not do it by the same means. Thus both of

if ... **then** ... ⟦**release** ρ⟧ ⟦**new** ρ⟧ ... **else** ...
if ... **then** ⟦**new** ρ'⟧ ... ⟦**release** ρ⟧ **else** ⟦$\rho' := \rho$⟧ ...

are legal but

if ... **then** ... ⟦**release** ρ⟧ **else** ...

is not.

Function calls are more complex (and the reader may want to refer to the examples on page 48f while digesting the general discussion that follows). The region annotation for a function call specifies three lists of **actual region parameters**:

$$\textbf{[c:}\ \rho_1, ..., \rho_k; \textbf{i:}\ \rho'_1, ..., \rho'_m; \textbf{o:}\ \rho''_1, ..., \rho''_n\textbf{]}$$

Here the ρ_i's are the **constant** parameters, the ρ'_i's are the **input** region parameters, and the ρ''_i's are the **output** region parameters.

The constant regions and input regions must be bound before the function call. When the called function starts executing, the bindings are transfered to region variables listed as formal region parameters in the function definition. No other region variables are bound at the beginning of the function body. The function body may not change the bindings of its formal constant parameters, but may do anything with the formal input region parameters. After the function body has been executed, the bound region variables must be exactly the formal constant parameters plus a set of formal output region parameters. The bindings of these are then transfered back to the caller's actual constant regions and output regions.

If any region variable that is neither an actual constant parameter nor an actual input parameter is bound before the function call, it is hidden from the called function but reappears in the caller after the call. It must not have the same name as an actual output parameter.

An actual input region parameter must not be identical to another actual input parameter or actual constant parameter in the same call. Likewise, each actual output region parameter must be distinct from other actual output parameters and from the constant parameters. However, two actual constant region parameters can be identical. These rules make sure that the region reference counts are maintained correctly: Any region aliasing must be done by explicit **alias** operations, and previously bound region variables may not be rebound without first being explicitly released. The relaxation for constant actuals is sound because the called function may not change the bindings of the corresponding formals; thus the bindings of the constant actuals is always the same before and after the call.

Note that an actual input region parameter loses its binding during the call (but can get a new one if it is used as an actual output parameter too).

2.4.1 Game of Life in the HMN system

The Game of Life example shows the the power of input and output regions (seeing the need for constant regions requires knowledge of the region type system, so we defer the rationale to the comments to the typing rule for function calls). The HMN system makes it possible to handle the Game of Life with no rewriting at all. We get the annotations

$$
\begin{array}{l}
\textbf{fun}\ \mathit{nextgen}\textbf{[i:}\ \rho;\textbf{o:}\ \rho'\textbf{]}\ (g) = \\
\quad [\![\textbf{new}\ \rho']\!]\ \langle \text{read } g \text{ from } \rho\text{; new gen. } \textbf{at}\ \rho' \rangle\ [\![\textbf{release}\ \rho]\!] \\
\textbf{fun}\ \mathit{life}\textbf{[i:}\ \rho_a, \rho_g;\textbf{o:}\ \rho'\textbf{]}\ (n, g) = \\
\quad [\![\textbf{release}\ \rho_a]\!]\ \textbf{if}\ n = 0\ \textbf{then}\ g\ [\![\rho' := \rho_g]\!] \\
\qquad\qquad \textbf{else}\ \mathit{life}\textbf{[i:}\ \rho'_a, \rho'_g;\textbf{o:}\ \rho'\textbf{]} \\
\qquad\qquad\qquad (n-1,\ \mathit{nextgen}\textbf{[i:}\ \rho_g;\textbf{o:}\ \rho'_g\textbf{]}\ (g)\ [\![\textbf{new}\ \rho'_a]\!])\ \textbf{at}\ \rho'_a
\end{array}
$$

where each iteration of *life* decides for itself whether to release the region it gets as its second parameter or to return it back to the caller.

The $[\![\rho' := \rho_g]\!]$ operation serves the same purpose as the *copy* operation in the AFL solution, but has no runtime cost. Indeed, even the $[\![\rho' := \rho_g]\!]$ operation itself is superfluous in this particular example; as an alternative all occurrences of ρ' could simply be changed to ρ_g. Then ρ_g would appear as a formal output region as well as a formal input region, but that does not matter; there is no *a priori* relation between the input and output regions.

To see the benefits of reference counting and the $[\![\rho' := \textbf{alias}\ \rho]\!]$ operation, consider the function

$$
\begin{array}{l}
\textbf{fun}\ \mathit{twolife}\ (g, n, m) = \ \textbf{let}\ g' = \mathit{life}(n, g) \\
\qquad\qquad\qquad\qquad\qquad\quad\ g'' = \mathit{life}(m, g') \\
\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ (g', g'')
\end{array}
$$

In the HMN system this can be annotated as

$$
\begin{array}{l}
\textbf{fun}\ \mathit{twolife}\textbf{[i:}\ \rho_g, \rho_a, \rho_m;\textbf{o:}\ \rho'_g, \rho''_g\textbf{]}\ (g, n, m) = \\
\quad \textbf{let}\ g' = \mathit{life}\textbf{[i:}\ \rho_a, \rho_g;\textbf{o:}\ \rho'_g\textbf{]}\ (n, g) \\
\qquad g'' = [\![\rho^{\diamond}_g := \textbf{alias}\ \rho'_g]\!]\ \mathit{life}\textbf{[i:}\ \rho_m, \rho^{\diamond}_g;\textbf{o:}\ \rho''_g\textbf{]}\ (m, g') \\
\quad \textbf{in}\ (g', g'')
\end{array}
$$

The second call to *life* is prevented from deallocating g' because the binding in ρ'_g keeps the region alive after *life* releases $\rho^{\diamond}_g$. All other invocations of *life* still deallocate the generation input.

This behavior is not possible in the AFL system: If any call to *life* needs to preserve the region where some of its inputs live, *no* call can be allowed to deallocate its argument.

The ML Kit system can handle certain instances of this situation by using storage-mode polymorphism to let a region be reset conditionally. However, the *twolife* example is not handled well by the ML Kit, since the Kit-friendly *life* implementation from Section 2.2 depends on being allowed to reset the region with the generation data.

A combination of the AFL and Kit solutions would be able to obtain a behavior similar to HMN for *twolife*. It would need to be guided by an explicit annotation from the programmer, though, because by default the Kit attempts to reset a region only when something gets allocated in it.

2.4.2 Simulation of the Kit and AFL systems

It is immediately clear that HMN can express everything allowed by the (first-order fragment of the) TT system. We now argue that a similar relation holds between HMN and the Kit and AFL systems.

Region resetting in the Kit can be simulated in HMN simply by the combination "⟦**release** ρ⟧ ⟦**new** ρ⟧". The Kit prohibits resetting if any live value has a type that mentions the region variable. When this is not the case, HMN's region type system will also allow the **release**–**new** combination. Resetting a region parameter in the Kit requires that the actual parameter was not live at the call site, in which case the HMN system would allow the parameter to be converted from a constant parameter to a pair of (identically named) input and output regions such that the function can still reset the region.

What HMN *cannot* simulate directly is the storage-mode polymorphism described in Section 2.2.2. We expect that most storage-polymorphic region parameters can be simulated by a combination of an input region for use before the resetting and a constant parameter or output region for use after the resetting. However, this does not always work in cases like

$$\textbf{fun}\ \mathit{nonneg}\ [\rho]\ (x) = \textbf{if}\ x < 0.0\ \textbf{at}\ \rho\ \textbf{then}\ (0.0\ \textbf{somewhereat}\ \rho)\ \textbf{else}\ x$$

where the region is only reset in some execution paths through the function. Region renaming in the **else** branch might work here, but only if all callers can accept that *nonneg* decides which region the result will be allocated in.

Simple cases of early deallocation in the AFL system is easily modeled by HMN's input region parameters. It is less clear whether HMN can model precisely all the effects of the AFL system's handling of region aliasing. Consider, for example, the following program fragment with AFL annotations:

$$\begin{aligned}
&\textbf{fun}\ f\ [\rho_x, \rho_y]\ (x, y) = \textbf{let}\ b = x > y\ [\![\textbf{free}\ \rho_x]\!] \\
&\qquad\qquad\qquad\qquad \textbf{in}\ \langle\text{code not depending on } x \text{ and } y\rangle \\
&\textbf{fun}\ g\ [\rho]\ x = f\ [\rho, \rho]\ (x, x) \\
&\textbf{fun}\ h\ [\rho]\ x = \ \textbf{letregion}\ \rho' \\
&\qquad\qquad\qquad \textbf{in}\ f\ [\rho, \rho']\ (x, (x + 1\ [\![\textbf{alloc}\ \rho']\!])\ \textbf{at}\ \rho')\ [\![\textbf{free}\ \rho']\!]
\end{aligned}$$

The best emulation of this in HMN is

$$\begin{aligned}
&\textbf{fun}\ f\textbf{[i:}\ \rho_x, \rho_y\textbf{]}\ (x, y) = \\
&\quad \textbf{let}\ b = x > y\ [\![\textbf{release}\ \rho_x]\!]\ [\![\textbf{release}\ \rho_y]\!] \\
&\quad \textbf{in}\ \langle\text{code not depending on } x \text{ and } y\rangle \\
&\textbf{fun}\ g\textbf{[i:}\ \rho\textbf{]}\ x = [\![\rho' := \textbf{alias}\ \rho]\!]\ f\textbf{[i:}\ \rho, \rho'\textbf{]}\ (x, x) \\
&\textbf{fun}\ h\textbf{[i:}\ \rho\textbf{]}\ x = f\textbf{[i:}\ \rho, \rho'\textbf{]}\ (x, (x + 1\ [\![\textbf{new}\ \rho']\!])\ \textbf{at}\ \rho')
\end{aligned}$$

In this version, f will delete both of its input regions after the comparison, whereas the AFL version cannot deallocate both ρ_x and ρ_y lest the same region would be deallocated twice when f is called from g. We conjecture that the pattern in this example holds generally: that an AFL agent can always be translated into a HMN agent that gives at least as good lifetimes of all memory blocks.

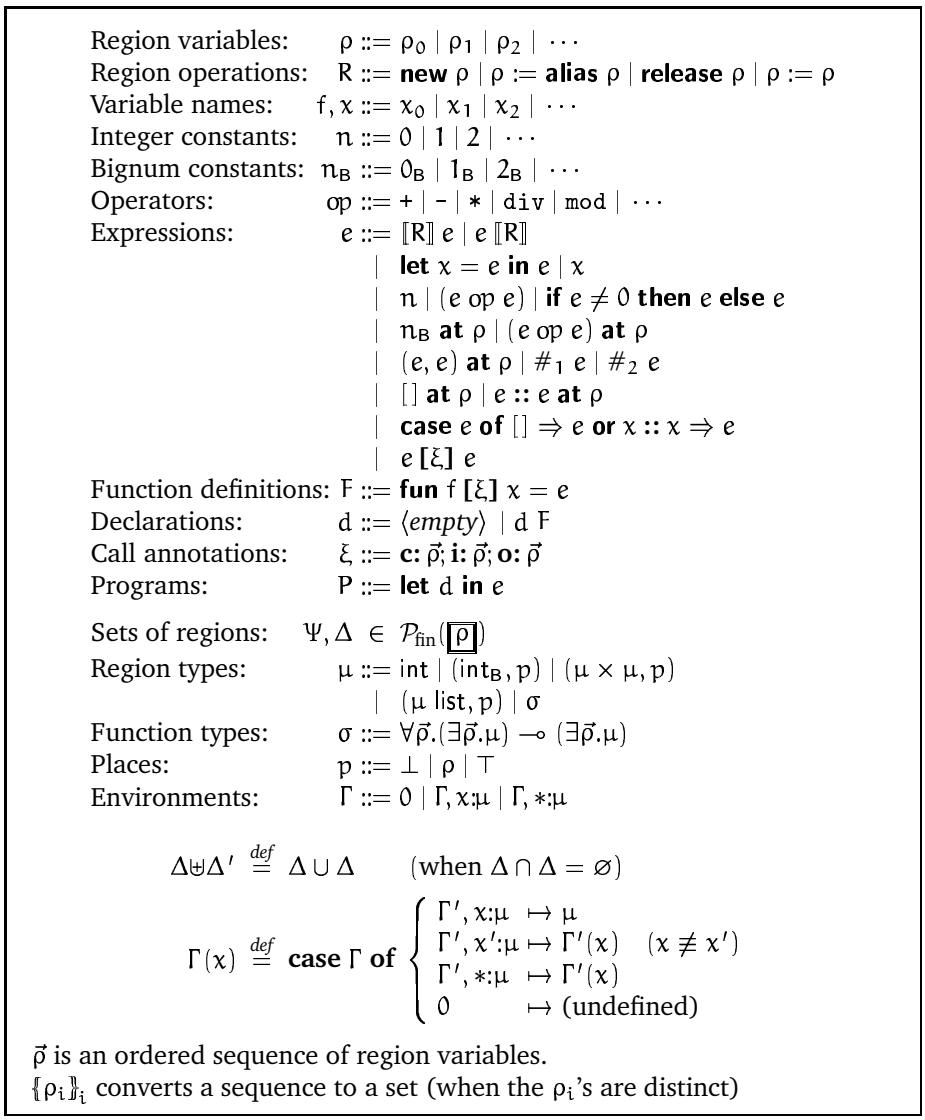


Figure 2.1: Abstract syntax of RegFun *and semantic objects (with associated operations) of the* HMN *region type system*

2.4.3 The HMN region type system

For reference, we now describe the HMN region type system as published in Henglein et al. [2001]. A safety proof for the region type system can be found in Niss [2002].

The host language is a small ML subset called FUN. The primitive data types are integers in unboxed (int) and boxed (int_B) versions. Our main reason for having boxed integers is that they make it easy for small examples to require heap allocation; one can also think of them as infinite-precision integers. We also include constructed data in the form of binary pairs and lists. The language contains the usual variables, **let**-expressions, constructors and destructors for data, and functions. FUN is "almost higher order" in that functions are first-class values, but it is not possible to create closures. Thus function values behave similarly to C's "function pointers".

Region inference takes a FUN-program and emits a region-annotated program. The resulting program is written in the target language REGFUN, which is FUN with the imperative agent programming language embedded.

Figure 2.1 shows the syntax of REGFUN, from which the syntax of FUN can also be inferred by ignoring the region constructs. An evaluation semantics for REGFUN is given in the online version of Henglein et al. [2001].

Also shown on Figure 2.1 are the semantic objects used in the region type system. The various constructions will be explained along with the rules that use them. In addition to the operations, we use the set $\mathrm{frv}(\mu)$ of free region variables of a region type μ (where, by definition, $\mathrm{frv}(\sigma) = \varnothing$). By point-wise extension, the free variables of an environment Γ is called $\mathrm{frv}(\Gamma)$.

The typing judgments that keep track of available regions and data are inspired by Floyd-Hoare Logic and take the form of pre- and post-descriptions of the runtime state. The main typing judgment has the form

$$\Psi \vdash \{\Delta / \Gamma\}\ e : \mu\ \{\Delta' / \Gamma'\}$$

Here Δ and Δ' are the sets of updateable bound region variables before and after the evaluation. They are complemented by Ψ which is the set of formal constant parameters in the current function. They behave mostly like the region variables in Δ, except that their bindings cannot change. Therefore, Ψ stays the same throughout an expression and consequently occurs only once in a judgment. The sets Ψ and Δ (or Ψ and Δ') are always disjoint; their union is the set of bound (or "**live**") regions variables before (or after) evaluating e.

Though the actual *values* in the runtime environment ideally do not change while evaluating an expression, embedded region operations may change the bindings of the region variables that allow access to them, such that their region-annotated type must be updated. Therefore the judgment contains two different type environments Γ and Γ' which both describe the same runtime environment, but relative to the region-variable bindings before and after the evaluation of e, respectively. The typing rules maintain the invariant that Γ and Γ' have the same structure; that is, the only differences are in the places p inside types.

The special place $\top$ is used to mark the types of values that are not accessible through any currently-bound variable. For example, the occurrence of $\top$ in

$$\vdash \{\{\rho, \rho'\} / x{:}(\mathsf{int}_B, \rho')\} \quad (x + 1_B \textbf{ at } \rho') \textbf{ at } \rho \; [\![\textbf{release } \rho']\!] : (\mathsf{int}_B, \rho) \quad \{\{\rho\} / x{:}(\mathsf{int}_B, \top)\}$$

signals that, after evaluation of the expression, the value of x is no longer accessible. Attempting a subsequent operation involving access to the value of x would be rejected by the type system.

Intuitively, HMN is very aggressive about deallocating regions: anything can be deallocated at any point. This is possible if the context of an expression does not depend on the deallocated data – they are semantically dead. If the context requires data from the deallocated region then the region type system gives a type error at the point of the data access. This deallocation strategy is different from the strategy in the TT system that only allows deallocation of unobservable regions.

Region operations

$$\boxed{\Psi \vdash \{\Delta / \Gamma\} \; R \; \{\Delta / \Gamma\}}$$

We start by defining the effect of region operations on the set of available regions and on the region-annotated types of live values:

$$\frac{\rho \notin \Psi \uplus \Delta}{\Psi \vdash \{\Delta / \Gamma\} \textbf{ new } \rho \; \{\Delta \uplus \{\rho\} / \Gamma\}}$$

$$\frac{\rho \in \Psi \uplus \Delta \quad \rho' \notin \Psi \uplus \Delta}{\Psi \vdash \{\Delta / \Gamma[\rho' \mapsto \rho]\} \; \rho' := \textbf{alias } \rho \; \{\Delta \uplus \{\rho'\} / \Gamma\}}$$

$$\frac{\rho \notin \mathrm{frv}(\Gamma)}{\Psi \vdash \{\Delta \uplus \{\rho\} / \Gamma\} \textbf{ release } \rho \; \{\Delta / \Gamma\}}$$

$$\frac{\rho' \notin \Psi \uplus \Delta}{\Psi \vdash \{\Delta \uplus \{\rho\} / \Gamma\} \; \rho' := \rho \; \{\Delta \uplus \{\rho'\} / \Gamma[\rho \mapsto \rho']\}}$$

Except for $\rho \notin \mathrm{frv}(\Gamma)$ in the third rule, the side conditions on the rules correspond to the preconditions listed for each region operation. The side condition $\rho \notin \mathrm{frv}(\Gamma)$ in the third rule is meant to handle the following situation:

let $x = [\![\textbf{new } \rho]\!] \; (1_B \textbf{ at } \rho) \; [\![\textbf{release } \rho]\!]$
in $[\![\textbf{new } \rho]\!] \; (x + 1_B \textbf{ at } \rho) \textbf{ at } \rho$

This fragment should be rejected by the type system since the region created in the body of the **let** is actually a fresh region different from the region used in the binding to *x*. And indeed the side condition rejects the fragment. To type $[\![\textbf{new } \rho]\!] \; (1_B \textbf{ at } \rho) \; [\![\textbf{release } \rho]\!]$ one would have to use a subtyping step (below), which would give *x* the type $(\mathsf{int}_B, \top)$, making it inaccessible in the body of the **let** expression.

Expressions

$$\boxed{\Psi \vdash \{\Delta / \Gamma\} \; e : \mu \; \{\Delta' / \Gamma'\}}$$

Expressions with region operations

The rule for a prefixed region operation is natural:

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\}\ \mathsf{R}\ \{\Delta_2/\Gamma_2\} \qquad \Psi \vdash \{\Delta_2/\Gamma_2\}\ e : \mu\ \{\Delta_3/\Gamma_3\}}{\Psi \vdash \{\Delta_1/\Gamma_1\}\ [\![\mathsf{R}]\!]\ e : \mu\ \{\Delta_3/\Gamma_3\}}$$

but the corresponding attempt to handle a postfix:

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\}\ e : \mu\ \{\Delta_2/\Gamma_2\} \qquad \Psi \vdash \{\Delta_2/\Gamma_2\}\ \mathsf{R}\ \{\Delta_3/\Gamma_3\}}{\Psi \vdash \{\Delta_1/\Gamma_1\}\ e\ [\![\mathsf{R}]\!] : \mu\ \{\Delta_3/\Gamma_3\}}\ \text{(WRONG!)}$$

does not work! Because R is executed *after* the evaluation of e it might affect region variables that appear in μ. For example, the rule above would allow constructions such as

$$\#_1\ \ ((5,7)\ \textbf{at}\ \rho\ [\![\textbf{release}\ \rho]\!]\ [\![\textbf{new}\ \rho]\!])$$

Instead of this rule we have to thread the μ through the typing of R so the latter can rewrite it appropriately. Similar cases also arise elsewhere in the system, so we invent a general solution: An environment Γ can contain **anonymous entries**, denoted by $*{:}\mu$, which hold the region-annotated types of intermediate results that are only needed during the evaluation of the current expression/operation. The intuition is that a Γ models the contents of the value stack if the expression were to be evaluated on a stack machine. Thus we write

$$\frac{\Psi \vdash \{\Delta_1/\Gamma_1\}\ e : \mu_0\ \{\Delta_2/\Gamma_2\} \qquad \Psi \vdash \{\Delta_2/\Gamma_2, *{:}\mu_0\}\ \mathsf{R}\ \{\Delta_3/\Gamma_3, *{:}\mu\}}{\Psi \vdash \{\Delta_1/\Gamma_1\}\ e\ [\![\mathsf{R}]\!] : \mu\ \{\Delta_3/\Gamma_3\}}$$

This explains the unusual definition of ⟨*VarEnv*⟩ in Figuregrammar and the corresponding lookup operation.

Region subtyping

We define a partial order $\sqsubseteq$ on places by

$$\bot \sqsubset \rho \sqsubset \top$$

and its natural pointwise extensions to μ's (not allowing $\sqsubset$ inside function types) and Γ's. The subtyping rule

$$\frac{\begin{array}{c}\Psi \vdash \{\Delta_1/\Gamma_1'\}\ e : \mu_0\ \{\Delta_2/\Gamma_2\} \\ \Gamma_1 \sqsubseteq \Gamma_1' \qquad \mathrm{frv}(\Gamma_1') \subseteq \Psi \uplus \Delta_1 \qquad \Gamma_2, *{:}\mu_0 \sqsubseteq \Gamma_2', *{:}\mu' \qquad \mathrm{frv}(\Gamma_2', *{:}\mu') \subseteq \Psi \uplus \Delta_2\end{array}}{\Psi \vdash \{\Delta_1/\Gamma_1\}\ e : \mu'\ \{\Delta_2/\Gamma_2'\}}$$

now allows region variables inside region types to be changed to $\top$. The effect of this is that the new type does not allow the values it describes to be accessed. On the other hand, if all instances of a region variable are changed to $\top$ the variable is not free in the environment anymore; this allows it to be released.

Similarly, a place can start out as $\bot$ and be changed to a region variable at a later time when the desired region variable comes into existence. That allows constructions such as

$$\textbf{let}\ x = [\,]\ \textbf{at}\ \rho\ \textbf{in}\ (\mathit{life}\textbf{[i:}\ ..\textbf{;}\ \textbf{o:}\ \rho'\textbf{]}(..)\ \textbf{::}\ x)\ \textbf{at}\ \rho$$

where *life* creates the region that contains the list elements *after* the [] has been constructed. The type of x can start out as $((..., \bot)\ \mathsf{list}, \rho)$ and just before the cons operation it can be reinterpreted as $((..., \rho')\ \mathsf{list}, \rho)$.

Variables and variable bindings

Variables and variable bindings are standard:

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : \mu\ \{\Delta_2 / \Gamma_2\} \qquad \Psi \vdash \{\Delta_2 / \Gamma_2, x{:}\mu\}\ e' : \mu'\ \{\Delta_3 / \Gamma_3, x{:}\mu''\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ \textbf{let}\ x = e\ \textbf{in}\ e' : \mu'\ \{\Delta_3 / \Gamma_3\}}$$

$$\frac{}{\Psi \vdash \{\Delta / \Gamma\}\ x : \Gamma(x)\ \{\Delta / \Gamma\}}$$

where the environment lookup $\Gamma(x)$ ignores anonymous entries.

Operations on unboxed integers

With unboxed integers no memory management or regions are needed at all, and all we have to do is mechanically pass around the environments. Note that the two branches of a conditional must have the same net effects on the regions and region types.

$$\frac{}{\Psi \vdash \{\Delta / \Gamma\}\ n : \mathsf{int}\ \{\Delta / \Gamma\}}$$

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : \mathsf{int}\ \{\Delta_2 / \Gamma_2\} \qquad \Psi \vdash \{\Delta_2 / \Gamma_2\}\ e' : \mathsf{int}\ \{\Delta_3 / \Gamma_3\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ (e\ \mathrm{op}\ e') : \mathsf{int}\ \{\Delta_3 / \Gamma_3\}}$$

$$\frac{\begin{array}{c}\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : \mathsf{int}\ \{\Delta_2 / \Gamma_2\} \\ \Psi \vdash \{\Delta_2 / \Gamma_2\}\ e' : \mu\ \{\Delta_3 / \Gamma_3\} \qquad \Psi \vdash \{\Delta_2 / \Gamma_2\}\ e'' : \mu\ \{\Delta_3 / \Gamma_3\}\end{array}}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ \textbf{if}\ e \neq 0\ \textbf{then}\ e''\ \textbf{else}\ e'' : \mu\ \{\Delta_3 / \Gamma_3\}}$$

Operations on boxed integers

Boxed integers are our main example of a primitive data type that must be heap allocated. Each constructor expression specifies where its result should be placed, and the rules check that all involved regions are live.

$$\frac{\rho \in \Psi \uplus \Delta}{\Psi \vdash \{\Delta / \Gamma\}\ n_B\ \textbf{at}\ \rho : (\mathsf{int}_B, \rho)\ \{\Delta / \Gamma\}}$$

$$\frac{\begin{array}{c}\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : \mu_0\ \{\Delta_2 / \Gamma_2\} \\ \Psi \vdash \{\Delta_2 / \Gamma_2, *{:}\mu_0\}\ e' : (\mathsf{int}_B, \rho')\ \{\Delta_3 / \Gamma_3, *{:}(\mathsf{int}_B, \rho)\} \\ \rho, \rho', \rho'' \in \Psi \uplus \Delta_3\end{array}}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ (e\ \mathrm{op}\ e')\ \textbf{at}\ \rho'' : (\mathsf{int}_B, \rho'')\ \{\Delta_3 / \Gamma_3\}}$$

In the latter rule, the side condition $\rho, \rho' \in \Psi \uplus \Delta_3$ is, strictly speaking, redundant, because the typing rules maintain the invariant that $\mathrm{frv}(\Gamma) \subseteq \Psi \cup \Delta$ (or, respectively, $\mathrm{frv}(\Gamma', *{:}\mu) \subseteq \Psi \cup \Delta'$). We choose to leave them in for readability. The condition $\rho'' \in \Psi \uplus \Delta_3$ is essential.

Pairs

The rules for pairs are simple. Observe that none of the rules check that the pair's *components* are readable. It does no harm to put a dangling pointer into

a pair and later extract it. What matters is that the pointer does not dangle if we try to read through it; this is independent of whether it has been stored in a pair or not.

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : \mu_0\ \{\Delta_2 / \Gamma_2\} \qquad \Psi \vdash \{\Delta_2 / \Gamma_2, *{:}\mu_0\}\ e' : \mu'\ \{\Delta_3 / \Gamma_3, *{:}\mu\} \qquad \rho \in \Psi \uplus \Delta_3}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ (e, e')\ \textbf{at}\ \rho : (\mu \times \mu', \rho)\ \{\Delta_3 / \Gamma_3\}}$$

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : (\mu_1 \times \mu_2, \rho)\ \{\Delta_2 / \Gamma_2\} \qquad \rho \in \Psi \uplus \Delta_2}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ \#_i\ e : \mu_i\ \{\Delta_2 / \Gamma_2\}}$$

Lists

Lists are the prototypical example of a recursive data type. The rules contain no surprises for readers who are familiar with the handling of lists in other region systems. This does not mean, however, that lists are trivial. In a sense, recursive types such as lists are only source of imprecision in the HMN region inference: A program that does not use lists can always be annotated such that at runtime, each region is used for exactly one allocation[8].

$$\frac{\{\rho\} \cup \mathrm{frv}(\mu) \subseteq \Psi \uplus \Delta}{\Psi \vdash \{\Delta / \Gamma\}\ []\ \textbf{at}\ \rho : (\mu\ \mathsf{list}, \rho)\ \{\Delta / \Gamma\}}$$

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : \mu_0\ \{\Delta_2 / \Gamma_2\} \qquad \Psi \vdash \{\Delta_2 / \Gamma_2, *{:}\mu_0\}\ e' : (\mu\ \mathsf{list}, \rho)\ \{\Delta_3 / \Gamma_3, *{:}\mu\} \qquad \rho \in \Psi \uplus \Delta_3}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e :: e'\ \textbf{at}\ \rho : (\mu\ \mathsf{list}, \rho)\ \{\Delta_3 / \Gamma_3\}}$$

$$\frac{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : (\mu\ \mathsf{list}, \rho)\ \{\Delta_2 / \Gamma_2\} \qquad \rho \in \Psi \uplus \Delta_2 \qquad \Psi \vdash \{\Delta_2 / \Gamma_2\}\ e' : \mu_3\ \{\Delta_3 / \Gamma_3\} \qquad \Psi \vdash \{\Delta_2 / \Gamma_2, x{:}\mu, x'{:}(\mu\ \mathsf{list}, \rho)\}\ e'' : \mu_3\ \{\Delta_3 / \Gamma_3, x{:}\mu', x'{:}\mu''\}}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ \textbf{case}\ e\ \textbf{of}\ [] \Rightarrow e'\ \textbf{or}\ x :: x' \Rightarrow e'' : \mu_3\ \{\Delta_3 / \Gamma_3\}}$$

Function calls

The HMN model for function calls is illustrated by the syntax for function types:

$$\forall \vec{\rho}.(\exists \vec{\rho}'.\mu') \multimap (\exists \vec{\rho}''.\mu'')$$

where $\vec{\rho}$ are the constant region parameters, $\vec{\rho}'$ are the input region parameters, μ' is the argument type, $\vec{\rho}''$ are the output region parameters, and μ'' is the result type. We do not allow function types to have free region variables, so there are implicit well-formedness conditions

$$\mathrm{frv}(\mu') \subseteq \{\!|\rho_i'|\!\}_i \cup \{\!|\rho_i|\!\}_i \quad \text{and} \quad \mathrm{frv}(\mu'') \subseteq \{\!|\rho_i''|\!\}_i \cup \{\!|\rho_i|\!\}_i$$

The intuition behind the existential quantifiers comes from dependent types: A region-annotated type can be viewed as depending on the region variables in

[8] This property does not hold for the TT system. It depends on being able to use region renamings to reconcile the region typing of **then** and **else** branches.

it, and so the input (or output) to the function consists of an existential pair with some regions and a value whose type depends on the regions.

The peculiar function arrow "$\multimap$" comes from linear logic and supports the intuition that the input regions and argument value disappear from the caller's context in the call; that is, unless the caller explicitly takes steps to save copies of them.

Finally, the universal quantification of the constant region parameters is borrowed from the TT type system. The constant regions may be present in the argument type as well as in the result type.

The entire notation supports a type-theoretic intuition about who selects the bindings of which region variables: The ρ_i's are selected by the caller because they are universally quantified. The ρ_i'''s are selected by the called function because they are existentially quantified. The ρ_i''s are also existentially quantified, but the quantifier is in a contravariant position, so they are selected by the caller.

The purpose of constant region parameters is to allow the caller to control the bindings of some of the region variables in the result type. Without constant region parameters we would not be able to type calls such as

$$\textbf{let } x = 17_B :: [\,] \textbf{ in } (f\ 0) :: x$$

The list construction requires that the return value from f must live in the same region as the previously-constructed x. If we did not have constant region parameters, f would need to have a type like

$$(\exists.\mathsf{int}) \multimap (\exists\rho.(\mathsf{int}_B, \rho))$$

which would not give the caller any guarantee that the region produced by the function were identical to the region where x is allocated. Even if we passed ρ_x into the function

$$(\exists\rho_x.\mathsf{int}) \multimap (\exists\rho_x.(\mathsf{int}_B, \rho_x))$$

it would be allowed to release the ρ_x input and create a new region to use as output, so we could not even be sure that x's value still existed after the call. (To prevent this situation, the typing rule below explicitly forbids any of the actual input regions to occur in the caller's type environment).

With constant region parameters, however, we can give f the type

$$\forall\rho_x.(\exists.\mathsf{int}) \multimap (\exists.(\mathsf{int}_B, \rho_x))$$

which allows the caller to specify where the results should be allocated.

The rule for function calls is the most complex rule in the region type system, and we therefore introduce it with an example. Consider the function f defined by

$$\textbf{fun } f\,\textbf{[i:}\ \rho_i; \textbf{o:}\ \rho_o\textbf{]}\ x = e_f$$

where f expects a boxed integer in ρ_i and returns a boxed integer in ρ_o (f may release ρ_i and create ρ_o, we do not know). We capture this with the function type:

$$f : \forall.(\exists\rho_i.(\mathsf{int}_B, \rho_i)) \multimap (\exists\rho_o.(\mathsf{int}_B, \rho_o))$$

To type

$$f\,\mathbf{[i:}\,\rho';\mathbf{o:}\,\rho''\mathbf{]}\ (17_B\ \mathbf{at}\ \rho')$$

we first type the subexpressions using the typing rules above and get

$$\Psi \vdash \{\Delta \uplus \rho' / \Gamma\}\ f : \sigma\ \{\Delta \uplus \rho' / \Gamma\}$$
$$\Psi \vdash \{\Delta \uplus \rho' / \Gamma\}\ 17_B\ \mathbf{at}\ \rho' : (\mathsf{int}_B, \rho')\ \{\Delta \uplus \rho' / \Gamma\}$$

We next have to match the actual region parameters ρ' and ρ'' with the formal parameters ρ_i and ρ_o, resulting in substitutions θ' and θ''. Then we match argument and result: $\theta'(\mathsf{int}_B, \rho')$ and $\theta''(\mathsf{int}_B, \rho'')$ with (int_B, ρ_i) and (int_B, ρ_o). The actual input parameter ρ' loses its binding in the call, so we remove it from $\Delta \uplus \rho'$; conversely ρ'' gets bound, and we add that to get the final Δ. We have then derived

$$\Psi \vdash \{\Delta \uplus \rho' / \Gamma\}\ f\,\mathbf{[i:}\,\rho';\mathbf{o:}\,\rho''\mathbf{]}\ (17_B\ \mathbf{at}\ \rho') : (\mathsf{int}_B, \rho'')\ \{\Delta \uplus \rho'' / \Gamma\}$$

In the general case with multiple region parameters, constant regions, and potential region operations in the operand we get the following rule.

$$\frac{\begin{array}{c}\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e : \mu_0\ \{\Delta_2 / \Gamma_2\} \\ \Psi \vdash \{\Delta_2 / \Gamma_2, *{:}\mu_0\}\ e' : \mu'\ \{\Delta_3 / \Gamma_3, *{:}\sigma\} \\ \Gamma_3; \Psi \vdash \{\Delta_3 / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta_4 / \mu''\}\end{array}}{\Psi \vdash \{\Delta_1 / \Gamma_1\}\ e\ \mathbf{[}\xi\mathbf{]}\ e' : \mu''\ \{\Delta_4 / \Gamma_3\}}$$

where we have used an auxiliary relation, described below, to handle the matching of actual parameters to formal parameters.

Parameter matching

$$\boxed{\Gamma; \Psi \vdash \{\Delta' / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' / \mu''\}}$$

The first rule below does the actual matching of parameters via substitutions θ, θ', and θ''. The remaining rules allow the manipulation of the current sets of region variables to match the shape of the function type. The third rule ensures that the bindings in the type environment does not contain references to the regions passed to the function.

$$\frac{\begin{array}{c}\sigma = \forall \vec{\rho}.(\exists \vec{\rho}'.\mu') \multimap (\exists \vec{\rho}''.\mu'') \\ \mathbf{[}\xi\mathbf{]} = \mathbf{[c:}\,\theta(\vec{\rho});\mathbf{i:}\,\theta'(\vec{\rho}');\mathbf{o:}\,\theta''(\vec{\rho}'')\mathbf{]} \\ \Psi = \theta(\{\!|\rho_i|\!\}_i) \quad \Delta' = \theta'(\{\!|\rho_i'|\!\}_i) \quad \Delta'' = \theta''(\{\!|\rho_i''|\!\}_i) \\ \mu_i = (\theta \oplus \theta')(\mu') \quad \mu_o = (\theta \oplus \theta'')(\mu'')\end{array}}{\emptyset; \Psi \vdash \{\Delta' / \mu_i\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' / \mu_o\}}$$

$$\frac{\emptyset; \Psi \vdash \{\Delta' / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' / \mu''\}}{\emptyset; \Psi \uplus \{\rho\} \vdash \{\Delta' / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' / \mu''\}}$$

$$\frac{\emptyset; \Psi \vdash \{\Delta' / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' / \mu''\} \quad \mathrm{frv}(\Gamma) \subseteq \Psi}{\Gamma; \Psi \vdash \{\Delta' / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' / \mu''\}}$$

$$\frac{\Gamma; \Psi \uplus \{\rho\} \vdash \{\Delta' / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' / \mu''\}}{\Gamma; \Psi \vdash \{\Delta' \uplus \{\rho\} / \mu'\}\ \sigma\ \mathbf{[}\xi\mathbf{]}\ \{\Delta'' \uplus \{\rho\} / \mu''\}}$$

Function definitions

$$\boxed{\Gamma \vdash F \Rightarrow x : \mu}$$

Typing a function definition is simply a matter of wrapping up the typing judgment for the body as a function type:

$$\frac{\{\!|\rho_i|\!\}_i \vdash \{\{\!|\rho_i'|\!\}_i / \Gamma, x{:}\mu'\}\ e : \mu''\ \{\{\!|\rho_i''|\!\}_i / \Gamma''', x{:}\mu'''\} \qquad \forall i,j : \rho_i \neq \rho_j'}{\Gamma \vdash \textbf{fun}\ f\textbf{[c:}\ \vec{\rho}\textbf{; i:}\ \vec{\rho}'\textbf{; o:}\ \vec{\rho}''\textbf{]}\ x = e \Rightarrow f : \forall\vec{\rho}.(\exists\vec{\rho}'.\mu') \multimap (\exists\vec{\rho}''.\mu'')}$$

Declaration sequences

$$\boxed{\Gamma \vdash d \Rightarrow \Gamma}$$

Typing declarations is simply a matter of bookkeeping:

$$\frac{}{\Gamma \vdash \Rightarrow 0} \qquad\qquad \frac{\Gamma \vdash d \Rightarrow \Gamma' \qquad \Gamma \vdash F \Rightarrow x : \mu}{\Gamma \vdash d\ F \Rightarrow \Gamma', x{:}\mu}$$

Programs

$$\boxed{\vdash P}$$

The rule for programs sets up some simple boundary conditions:

$$\frac{\Gamma \vdash d \Rightarrow \Gamma \qquad \varnothing \vdash \{\varnothing/\Gamma\}\ e : \mu\ \{\varnothing/\Gamma'\}}{\vdash \textbf{let}\ d\ \textbf{in}\ e}$$

Note that we require that all regions be deallocated when the program is finished.

2.4.4 Non-optimality of region annotations

It would be nice to be able to show that, contrary to the Kit and AFL systems,

> *For a given* FUN *program, there is a set of region annotations that is optimal in the sense that no other well-typed annotations for that program will deallocate any value sooner than the optimal one.*

Unfortunately this is not true, with essentially the same counterexample as we gave for the Kit system in Section 2.2.3

```
let fun f n = let p = (5_B, 7_B) in if n = 0 then p else f (n − 1)
    fun g n = let (x, y) = f n ; z = x :: y :: [ ] in ⟨use z⟩
in let x = #1 (f 42) in ⟨space-critical section that uses x⟩
```

g must be able to call f in such a way that the components of the pair that f returns are guaranteed to be in the same region. Ignoring (for brevity) the region where the *pair* containing 5_B and 7_B is allocated, there are two fundamentally different ways to region-annotate f which allow this:

a.
```
f : ∀ρ1, ρ2.(∃ .int) −o (∃ .(int_B, ρ1) × (int_B, ρ2))
fun f[c: ρ1, ρ2] n = let p = (5_B at ρ1, 7_B at ρ2)
                     in if n = 0 then p
                        else f[c: ρ1, ρ2] (n − 1)
```

b. $f : \forall\ .(\exists\ .\mathsf{int}) \multimap (\exists \rho_3.(\mathsf{int}_B, \rho_3) \times (\mathsf{int}_B, \rho_3))$
$\quad$ **fun** f**[o:** ρ_3**]** $n =$ **let** $p = [\![$**new** $\rho_3]\!]\ (5_B$ **at** $\rho_3, 7_B$ **at** $\rho_3)$
$\quad\quad\quad\quad\quad\quad\quad\quad$ **in if** $n = 0$ **then** p
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ **else** $[\![$**release** $\rho_3]\!]\ f$**[o:** ρ_3**]** $(n - 1)$

Each of these have different advantages. Version (a) contains a space leak in f itself if n is large, because the 5_B's and 7_B's allocated by each iteration are never deallocated. In version (b), the region containing 5_B and 7_B is released before the recursive call.

However, from the main program's point of view, version (a) is desirable, because then the two components of the pair can be allocated in different regions, and the second one deallocated before the space-critical section. With version (b), the main program must keep both elements around until x can be safely deallocated.

The net result of this is that neither typing (a) nor (b) is clearly superior to the other.

One way to resolve this dilemma would be to replicate the definition of f such that call sites requiring different region separations of the return value call different implementations of f. Then the call from g could use typing (b) while the main program would call a different implementation of f with typing

c. $f : \forall\ .(\exists\ .\mathsf{int}) \multimap (\exists \rho_1, \rho_2.(\mathsf{int}_B, \rho_1) \times (\mathsf{int}_B, \rho_2))$

This strategy would be sound and terminating even in the presence of recursion, because there is only a finite number of fundamentally different region annotations of each return type. However, the number of different versions of each function might grow exponentially with the size of the return type, so the strategy is not necessarily optimal in a practical sense.

Chapter 3

The universal host language

This chapter presents the "universal host language" UHL, which together with its associated agent programming language forms the foundation of my language-independent (or perhaps rather "language-portable") theory of region inference.

The role of UHL in the overall theory was described in Section 1.4.2. Recall that UHL is supposed to be "the union of all sensible intermediate languages with everything that is irrelevant to memory management abstracted away", and that UHL programs are called "uniform mutators".

The chapter has four parts:

- In Section 3.1 we define a suitable universal host language and its ideal semantics.
- Section 3.2 introduces an agent programming language for UHL and defines its semantics formally.
- In Section 3.3 we define a criterion for an agent to be "correct", *region soundness*, and develop some general techniques for reasoning about it.
- Section 3.4 introduces the concept of "annotatable edges", which are a way for the host implementation to restrict the shape of the agent produced by the region inference.

In later chapters we will investigate the problem of finding a sound agent for a particular uniform mutator (which will in general depend on knowledge of the particular translation that produced the uniform mutator from a mutator written in a specific host language), and give algorithms for optimizing an agent such that it uses the region manager's operations more efficiently.

In Section 3.5 I will discuss a few possible further extensions to our formalism that I have not yet investigated in detail.

3.1 Uniform mutators without region annotations

The fundamental design choice is that at its heart the universal host language will be a *flowchart language*. This choice is responsible for the general applicability of our theory; the rest of this section really just develops the details necessary to make this idea work.

The flowchart idea allows us to model the vast majority of control constructs found in actual programming languages directly in a uniform mutator. One exception that we cannot model is lazy or call-by-name evaluation, where the control flow is notoriously difficult to reason about and in any case usually has little direct relation to the syntactic structure of the program. Another is call/cc and similarly strong control operators which I simply don't know how to reason about in sufficient detail to do region-based memory management.

On the other hand, we shall see that restricted control features such as exceptions and backtracking *can* be integrated with a flowchart-based universal host language.

3.1.1 A simple example

Figure 3.1 shows a flowchart for a program that computes primes using the "Sieve of Eratosthenes" algorithm that is presented as a Standard ML-ish program in the lower half of the figure. Procedures are going to be rather complex in UHL, so to give a soft introduction this example goes to some length to avoid them. Them main problem is that the inner loop in the ML formulation is not tail recursive. Therefore, the flowchart represents "nil" as a cons cell with the distinguished value 0 in its first cell. The inner loop can then construct its return list from front to back by allocating a new "nil" cell and destructively overwriting the old one (pointed to by the `F` variable) whenever it needs to append a number to the list.

The flowchart should be self-explaining, except for the nodes that are concerned with the program's use of the heap. They fall in three classes:

$\mathtt{L} := \mathsf{cons}\,\langle \mathtt{N}, \mathtt{L}\rangle$ – **Allocate**, somehow, two consecutive cells of heap memory, initialize them with the values of `N` and `L`, and then assign the address of the first cell to `L`. (The other three cons nodes in the program also follow this pattern, but initialize the newly allocated cells with zeroes instead of variable values).

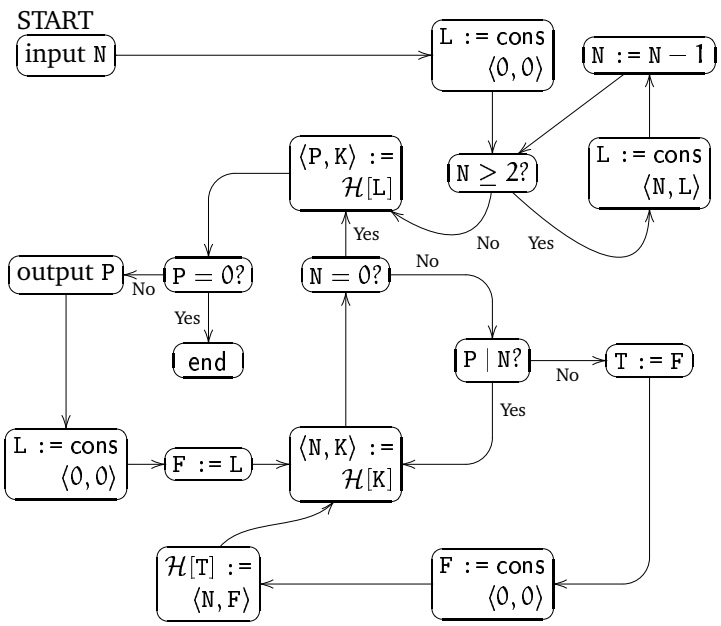
$\langle \mathtt{N}, \mathtt{K}\rangle := \mathcal{H}[\mathtt{K}]$ – **Read** the values in heap cells `K` and $\mathtt{K}+1$, and assign them to `N` and `K`, respectively. The letter $\mathcal{H}$ is part of the notation; it is meant to mnemonically symbolize the heap.

$\mathcal{H}[\mathtt{T}] := \langle \mathtt{N}, \mathtt{F}\rangle$ – Destructively **write** the current values of `N` and `F` into heap cells `T` and $\mathtt{T}+1$.

These three kinds of nodes will be a uniform mutator's only way to access the heap. They will all be handled specially in our formal treatment of uniform mutators. This is natural in the case of allocations (which is where the memory manager needs to do its work to locate a suitable block of unused heap), but we also handle reads and writes specially so that our formal semantics for uniform mutators can trap access attempts outside the currently allocated address space. This is what will allow us to prove formal statements about memory safety later.

Note that we distinguish between *input/output*, which are communication with the user, or some other entity external to the program, and *reads/writes*, which move data between the heap and the mutator's local variables.

The address of heap cells are just natural numbers. They can be arbitrarily large; Box 3.1 discusses this decision in detail.



Informal guide to variable use:
- `P` is the prime whose multiples are being eliminated currently.
- `N` is a candidate for elimination.
- `K` iterates through the old list of candidates.
- `L` points to the front of the new list of candidates.
- `F` points to the *end* of the new list as it is being built.
- `T` temporarily stores the old value of `F`.

```
fun from2to(N,L) = if N >= 2 then from2to(N-1, N::L) else L
fun mainloop(nil) = ()
  | mainloop(P::K) =
      let val _ = output P
          fun innerloop(nil) = nil
            | innerloop(N::K) = if <P divides N>
                                then innerloop K
                                else N :: innerloop K
          val L = innerloop(K)
      in mainloop(L) end
val N = input ()
val L = from2to(N,nil)
val _ = mainloop(L)
```

Figure 3.1: A flowchart for the Sieve of Eratosthenes. This notation is not yet exactly the Universal Host Language; compare Figure 3.2. To ease understanding, the lower part of the figure shows the underlying algorithm in Standard ML notation. (Note, however, that the flowchart was constructed by hand rather than systematically extracted from the ML program).

Box 3.1—Rationale for using an unbounded address space

In our formal development we allow arbitrary large numbers as heap addresses. This amounts to assuming that the available address space is infinitely large, such that we do not need to worry about running out of memory. Such a decision may seem to invalidate the entire point of discussing memory management: If memory is assumed to be infinite, then why bother freeing heap cells ever? However, the assumption actually serves to strengthen our formal results. It allows us to prove simple statements such as "If program A does something, then program B does the same thing". If we were to account for running out of memory, we would often need to say instead, "If program A does something, then program B either does the same or runs out of memory", which is weaker because it is trivially satisfied by a B that always runs out of memory.

The assumption of infinite memory does not mean that memory management has become completely pointless. It is still possible to express that program A can run without ever needing addresses larger than some N, whereas program B, which has the same I/O behavior, may need many more addresses. In fact, it is crucial that we can say that B *does* have the same extensional behavior even if our semantic model of it will need more addresses than the machine we intend to run our programs on has. Otherwise we would have learned nothing interesting about the relation between A and B.

A consequence of using natural numbers as addresses is that we have to pretend that a machine word can store an arbitrary natural number. This is, of course, as untrue of real machines as the assumption that they have infinite memory. However, it seems to be the most convenient way to model program execution at a low level without also modeling running out of memory.

Another, minor, consequence of using natural numbers for heap addresses is that a uniform mutator *knows* that addresses are just numbers. It may do strange arithmetic tricks on them, and if it can manage to get back the original number by further calculations, it may be justified in assuming that the recovered number is still good for heap accesses. This situation will not arise for high-level host languages, but if a language such as C were mapped to the universal host language, typecasts between integers and pointers would amount to such trickery. Much of our work here still be useful in this setting, though we do not present any concrete method for arriving at a sound agent in this situation from scratch.

For example, Reynolds [2002]'s "separation logic" can prove correct programs with arithmetic pointer tricks such as the XOR representation of doubly linked lists. One might imagine that this work could eventually be combined with regions, in which case it is important for the basic model to be able to handle it.

Notation 3.1. *For clarity, we will often use* $\mathbb{A}$ *as a synonym for* $\mathbb{N}$ *in contexts where it is implied that it is to be interpreted as the address space of the heap. The variable letter* $\mathfrak{a}$ *will in general range over numbers interpreted as addresses.*

3.1.2 The procedure-less fragment of UHL

In order to ease some of the formal development that is to follow, we will require uniform mutators to have a structure that is a little more fine-grained than the informal one shown in the previous section.

1. For uniformity, the values used to initialize newly-allocated heap cells

must always be given as *variables* rather than arbitrary expressions. In our example, nodes like "$\mathtt{L} := \mathsf{cons}\,\langle 0, 0\rangle$" will not be allowed, because the zeros are constants rather than variable. Such constructions must be represented using a pair of throwaway variables that are explicitly initialized to zero before the allocation takes place.

2. Heap read and write operations (but not allocation) must take place one cell at a time. That will simplify our formal treatment of memory errors, because we will not have to handle the case that a single heap operation refers to multiple cells, only some of which are allocated at the moment.

   We define single-cell reads and writes such that the variable that holds the heap address will be automatically increased by one after the heap operation has taken place. This is convenient for reading or writing a sequence of cells; and certain properties of region-based execution (Theorem 3.46) will be easier to establish if we can assume that the uniform mutator uses this auto-increment as its only form of pointer arithmetic.

   As a reminder of this auto-increment, the formal syntax for reads and writes will be "$\mathtt{Y} := \mathcal{H}[\mathtt{X}{+}{+}]$" and "$\mathcal{H}[\mathtt{X}{+}{+}] := \mathtt{Y}$". The "++" notation, of course, comes from C.

3. Heap allocations and writes *consume* the variable(s) that get written to the heap. If the uniform mutator needs the value later, it must save a copy of it itself. This decision is essentially arbitrary, but it will make the treatment of ordinary and region variables more similar.

4. No operation is allowed to give a new value to a variable that is already bound. A variable can be explicitly unbound, after which it can be assigned to again. This is also for consistency with the treatment of region variables. (And it will slightly simplify our handling of procedures, too).

The uniform mutator corresponding to Figure 3.1 is shown on Figure 3.2. The two-dimensional notation in the figure is convenient for giving examples, and for thinking about the language, but it is not well suited to mathematical treatment. Let us therefore define the following formal abstract syntax for the procedure-less fragment of UHL (recall the $\boxed{\mathtt{x}}$ notation from Section 1.6.3):

Variable names:	$\mathtt{x} ::= \mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \cdots$	
Control state names:	$\mathtt{s} ::= \mathtt{s0} \mid \mathtt{s1} \mid \mathtt{s2} \mid \cdots$	
Jumps:	$\mathfrak{Jmp} ::= \mathsf{goto}\,\mathtt{s}$	
Data states:	$\sigma \in \boxed{\mathtt{x}} \xrightarrow{\text{fin}} \mathbb{W}$	
I/O events:	$\varphi \in \{\ominus\} \cup \mathbb{W}$	
Computation steps:	$\mathfrak{St} ::= \sigma_1 \xrightarrow{\varphi} \sigma_2$	
Sets of steps:	$\mathfrak{Ss} \in \mathcal{P}(\boxed{\mathfrak{St}})$	
Mutator operations:	$\mathfrak{Mop} ::= \mathtt{x} := \mathsf{cons}\,\langle \mathtt{x}_0, \ldots, \mathtt{x}_k\rangle\,;\ \mathfrak{Jmp}$	$(k \geq 0)$
	$\mid\ \mathtt{x}_0 := \mathcal{H}[\mathtt{x}{+}{+}];\ \mathfrak{Jmp}$	
	$\mid\ \mathcal{H}[\mathtt{x}{+}{+}] := \mathtt{x}_0;\ \mathfrak{Jmp}$	
	$\mid\ \mathsf{end}$	
	$\mid\ \mathsf{misc}\ \mathfrak{Ss}_1;\ \mathfrak{Jmp}_1 \,\square\, \cdots \,\square\, \mathfrak{Ss}_k;\ \mathfrak{Jmp}_k$	$(k \geq 0)$
Uniform mutators:	$\mathcal{M} \in \boxed{\mathtt{s}} \xrightarrow{\text{fin}} \mathfrak{Mop}$	

For now, we will only give an informal discussion of how this syntax works; its formal semantics will not be separately defined but can be extracted from the

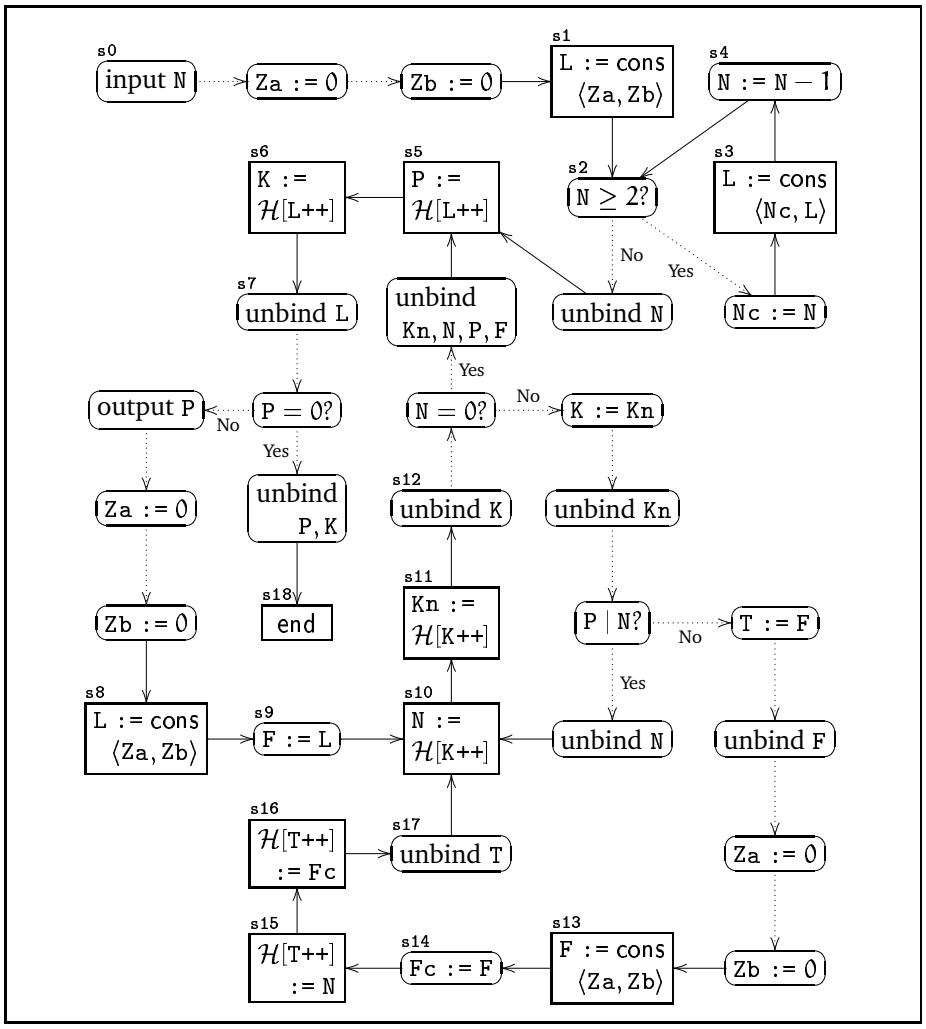


Figure 3.2: The Sieve of Eratosthenes as a uniform mutator. Compare Figure 3.1.

semantics for full UHL (Section 3.1.4) by ignoring the *IXCALL* and *IXRETURN* rules and letting the stack be empty.

Execution of the uniform mutator implicitly starts at the control state s0. We will use the terms "control state" and "node" interchangeably. ("Node" is shorter and perhaps more intuitive, but it would be confusing to use variable letter $\mathfrak{n}$ for node names, so we keep "control state" around to justify using s instead).

In the syntax, we recognize some of the nodes in Figure 3.2 – they are the ones that are drawn with *square* frames. But what about those with rounded frames? They are all handled by the "misc" case, which absorbs everything that does not have to do with the heap. It works as follows:

Before the misc node is about to be executed, some variables will already be bound to values. These bindings are recorded in a **data state** σ. Now, to execute the misc node, choose a **step** $\mathfrak{St} = \sigma_1 \xrightarrow{\varphi} \sigma_2$ from one of the $\mathfrak{Ss}$ in the misc construct, such that $\sigma = \sigma_1$. If no $\mathfrak{Ss}$ contains such a step, the program

gets stuck (this intuitively models a run-time error). If there is more than one step that matches σ, choose among them nondeterministically. Then record the **I/O event** φ and continue by executing with data state σ_2 and the control state given by the $\mathfrak{Jmp}$ corresponding to the $\mathfrak{Ss}$ where the step was found.

Wait – what is an "I/O event"? From the viewpoint of UHL, it is just a tag that we make no attempt to interpret, except for the special event $\ominus$ which marks time passing *without* any I/O. Eventually, we will model the behavior of a uniform mutator as the set of sequences of I/O events that are possible. (In general more than one sequence may be possible because of nondeterminism).

The Eratosthenes example would use things like "in.42" and "out.17" as I/O events. At the UHL level we do not make any formal distinction between input and output, but one may choose to interpret a nondeterministic choice between different "in.n" events as an actual input action.

The completely formal version of our example program would be

$$
\begin{aligned}
\mathcal{M}(\mathtt{s0}) &= \mathsf{misc}\ \left\{ \left[\right] \xrightarrow{\mathrm{in}.n} \begin{bmatrix} \mathtt{N} & \mathtt{Za} & \mathtt{Zb} \\ n & 0 & 0 \end{bmatrix} \;\middle|\; n \in \mathbb{N} \right\};\ \mathsf{goto}\ \mathtt{s1} \\
\mathcal{M}(\mathtt{s1}) &= \mathtt{L} := \mathsf{cons}\,\langle \mathtt{Za}, \mathtt{Zb} \rangle\ ;\ \mathsf{goto}\ \mathtt{s2} \\
\mathcal{M}(\mathtt{s2}) &= \mathsf{misc}\ \left\{ \begin{bmatrix} \mathtt{N} & \mathtt{L} \\ n & a \end{bmatrix} \xrightarrow{\ominus} \begin{bmatrix} \mathtt{N} & \mathtt{Nc} & \mathtt{L} \\ n & n & a \end{bmatrix} \;\middle|\; n \in \mathbb{N}, n \geq 2, a \in \mathbb{A} \right\};\ \mathsf{goto}\ \mathtt{s3} \\
&\quad \square \left\{ \begin{bmatrix} \mathtt{N} & \mathtt{L} \\ n & a \end{bmatrix} \xrightarrow{\ominus} \begin{bmatrix} \mathtt{L} \\ a \end{bmatrix} \;\middle|\; n \in \mathbb{N}, n < 2, a \in \mathbb{A} \right\};\ \mathsf{goto}\ \mathtt{s5} \\
\mathcal{M}(\mathtt{s3}) &= \mathtt{L} := \mathsf{cons}\,\langle \mathtt{Nc}, \mathtt{L} \rangle\ ;\ \mathsf{goto}\ \mathtt{s4} \\
\mathcal{M}(\mathtt{s4}) &= \mathsf{misc}\ \left\{ \begin{bmatrix} \mathtt{N} & \mathtt{L} \\ n & a \end{bmatrix} \xrightarrow{\ominus} \begin{bmatrix} \mathtt{N} & \mathtt{L} \\ n-1 & a \end{bmatrix} \;\middle|\; n \in \mathbb{N}, a \in \mathbb{A} \right\};\ \mathsf{goto}\ \mathtt{s2} \\
\mathcal{M}(\mathtt{s5}) &= \mathtt{P} := \mathcal{H}[\mathtt{L{+}{+}}];\ \mathsf{goto}\ \mathtt{s6} \\
\mathcal{M}(\mathtt{s6}) &= \mathtt{K} := \mathcal{H}[\mathtt{L{+}{+}}];\ \mathsf{goto}\ \mathtt{s7} \\
\mathcal{M}(\mathtt{s7}) &= \mathsf{misc}\ \left\{ \begin{bmatrix} \mathtt{L} & \mathtt{P} & \mathtt{K} \\ a' & n & a \end{bmatrix} \xrightarrow{\mathrm{out}.n} \begin{bmatrix} \mathtt{P} & \mathtt{K} & \mathtt{Za} & \mathtt{Zb} \\ n & a & 0 & 0 \end{bmatrix} \;\middle|\; n \in \mathbb{N}, n \neq 0, a, a' \in \mathbb{A} \right\};\ \mathsf{goto}\ \mathtt{s8} \\
&\quad \square \left\{ \begin{bmatrix} \mathtt{L} & \mathtt{P} & \mathtt{K} \\ a' & 0 & a \end{bmatrix} \xrightarrow{\ominus} \left[\right] \;\middle|\; a, a' \in \mathbb{A} \right\};\ \mathsf{goto}\ \mathtt{s18} \\
&\ldots
\end{aligned}
$$

which is well suited for mathematical treatment but quite unreadable in large doses. We shall not attempt to write down such an explicit flowchart mapping again; for giving examples, a visual notation such as the one in Figure 3.2 will usually suffice.

Observe in the example that each $\mathfrak{Ss}$ in the "misc ..." operation is generally an *infinite* set. It is important to realize that it is the infinite set itself, rather than some syntactic representation of it, that is part of the uniform mutator. This is sensible only because uniform mutators never need to exist in full on the computer – remember that actual implementations of our techniques will always be done after having mapped the general theory backwards along the translation from a real host language.

In this way, the $\mathfrak{Ss}$ has some of the flavor of denotational semantics: It is an unwieldy mathematical object that represents the abstract meaning of part of the program. And as in denotational semantics, is not inherent in our definition that it represents the meaning of an actual possible computation; even a "wild" set such as

$$
\begin{aligned}
\mathcal{M}(\mathtt{s42}) &= \mathsf{misc}\ \left\{ \begin{bmatrix} \mathtt{N} \\ n \end{bmatrix} \xrightarrow{\ominus} \left[\right] \;\middle|\; \text{Turing machine number } n \text{ halts} \right\};\ \mathsf{goto}\ \mathtt{s43} \\
&\quad \square \left\{ \begin{bmatrix} \mathtt{N} \\ n \end{bmatrix} \xrightarrow{\ominus} \left[\right] \;\middle|\; \text{Turing machine number } n \text{ diverges} \right\};\ \mathsf{goto}\ \mathtt{s44}
\end{aligned}
$$

would qualify as a well-formed mutator operation. Its appearance in a uniform mutator would constitute a promise from the person who constructed it that

if only we give him suitable region annotations, he can write machine code that solves the halting problem without needing to use the heap. He may have difficulty keeping that promise, but as long as we are concerned with region inference, that is not *our* problem.

A final observation is that a single misc state in the formal flowchart can correspond to several of the informal primitive nodes in Figure 3.2. For example, state s0 in the example simultaneously does the work of the "input N", "Za := 0", and "Zb := 0" nodes in the figure. In general, a misc state can represent an entire single-entry subgraph of the informal flowchart, as long as it consists entirely of "rounded" boxes and does not contain a path between two I/O nodes. On Figure 3.2, the edges between nodes that could be combined into a single misc are shown as dotted.

3.1.3 Subroutines

Subroutines of some kind or another are a pervasive phenomenon in all programming paradigms, save the most formal models of computability such as Turing machines. They represent the most primitive form of code genericity; very little serious programming can be done without them. Like every kind of genericity, they are likely to cause problems for program analyses in general; indeed the main complexity in many schemes for program analysis lies in preventing the model of one call context from "tainting" the model of another because both must match the model of the body of the subroutine. One widely-known example is Hindley–Milner type polymorphism, which gives an acceptable solution for this problem in the case of static type checking. It has provided a conceptual model for solving many other instances of the problem, including the original Tofte–Talpin approach to region inference.

Given the importance of good handling of subroutines, we need to model the call–return discipline in our universal host language (see Box 3.2 for discussion). The natural way to do this would be to define that a procedure should be an isolated single-entry, single-exit part of the uniform mutator's flowchart. We shall, however, go a little further and allow *multiple* entries and multiple exits.

Multiple exits is the feature that is most remarkable. The flowchart for a procedure may have more than one return state; a flowchart node representing a call must have an outgoing edge for each of the called procedure's returns. The primary envisaged use of multiple exits is for modeling exceptions. A procedure will have one return state signifying a normal exit and another signifying throwing an exception. The call site's outgoing edge for an exception throw would normally go directly to its own exceptional return, except if the call is lexically within a handler context.

It is well known that exceptions can also be modeled by having a single exit and returning a tagged value, but that model would make it more complicated to reason about an agent whose internal state satisfies a different invariant after an exceptional exit than it does after a normal exit.

Multiple entries to a procedure offer less immediate benefits, but also require almost no explicit support in the formal development. In essence, one could say that we allow them "because we can". We envisage two applications for them. One is to model partial static knowledge about the range of dynamic dispatch, as we shall see in Section 6.2.2. The other is to allow moving more freely between loops and tail recursion, as discussed in Section 7.3.8.

Box 3.2—An alternative to modeling returns directly

As obvious as the choice to model the call–return discipline directly may seem, it is not the only one possible. Subroutines *could* be reduced to pure flowchart programming by CPS-transforming the program and then converting the resulting indirect calls to continuation functions to switches on first-order tags. Afterwards all calls will be tail calls to statically known subroutines and can be replaced with ordinary jumps.

One argument for this approach would be that what machine language implements is actually "pure flowchart programming", if we ignore the possibility of errant indirect jumps to non-code. In this light the CPS strategy is just a description of how compilation of a language with subroutines traditionally works. Therefore low-level type systems such as Typed Assembly Language [Morrisett et al. 1998, 1999] usually have a strong CPS flavor.

Region-based memory management has been carried at least part way through this program; Crary, Morrisett, and Walker [Crary et al. 1999; Walker et al. 2000] have constructed a region-based "calculus of capabilities" for programs in continuation-passing style, with the goal of allowing Tofte–Talpin style agents to be CPS transformed along with their mutator and still be proved safe by a region type system. The resulting model has remarkable expressive power and actually seems to be stronger than the region type systems we will use to guide inference algorithms in the next chapter.

However, the continuation-based approach can also be seen as trading the problem of return to a not-statically-known call site away for the problem of calling a not-statically-known function (that is, the continuation). My intuitive experience is that the latter problem is intrinsically harder than the former. This feeling, formed during several years of working with region inference for first-order host languages and trying to generalize the lessons learned to higher-order ones, may or may not be correct, but it suggests that it would be valuable for a generic model of region-based memory management to handle the call–return discipline without necessarily adding in the burden of dynamic dispatch.

Furthermore, for all the power of the capability calculus, it is very technical and not easily accessible for a nonspecialist programmer. This is undesirable because strong practical experience indicates that it is important for the programmer of the mutator to be able to understand how the agent works, even if he does not need to understand the region inference process itself. However, the complexity of the capability calculus seems to be a natural consequence of working with a CPS language.

With multiple entries, one does not call a procedure *per se*; instead one just calls an entry node. The possible returns from the call are those return states that happen to be reachable from the specified entry node. Thus, we do not need to formally partition the program into individual procedures with a body flowchart each. Instead we have one big flowchart containing all the procedures as (usually) mutually unconnected components. This means that we can also handle, without changes to the formal model, completely unstructured mazes of GOSUB/GOTO/RETURN as found in bad Basic code. As direct jumps between procedure bodies are added, the precision of our region inference techniques may deteriorate, but it should do so with reasonable grace.

We can use the "`end`" mutator operation for returns from procedures, so the only new syntax we need to add to extend the universal host language with

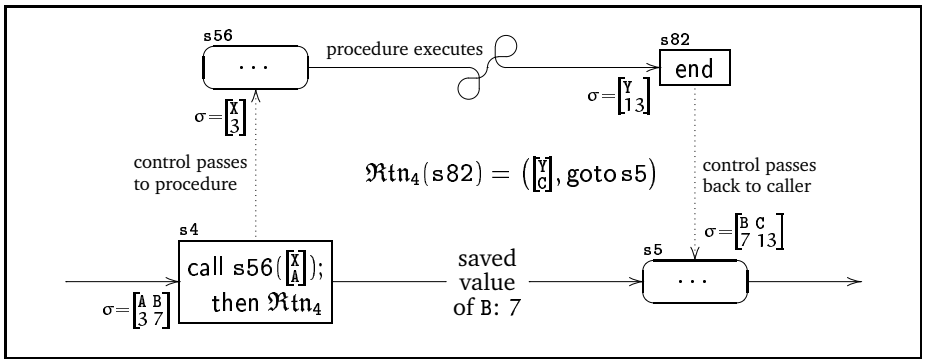


Figure 3.3: Schematic flow of parameters and return value in a procedure call. The σ annotations on each node shows the data state when the node begins execution.

procedures is a call operation. On the other hand, generality requires it to be somewhat complex:

$$\begin{array}{lll}
\text{(Mutator) argument lists:} & \mathfrak{Ma} & \in \boxed{\mathtt{X}} \xrightarrow[\text{inj}]{\text{fin}} \boxed{\mathtt{X}} \\
\text{(Mutator) return maps:} & \mathfrak{Rtn} & \in \boxed{\mathtt{S}} \xrightarrow{\text{fin}} \boxed{\mathfrak{Ma}} \times \boxed{\mathfrak{Jmp}} \\
\text{Mutator operations:} & \mathfrak{Mop} & ::= \cdots \\
 & & \quad \mid \mathsf{call}\ \mathtt{s}(\mathfrak{Ma});\ \mathsf{then}\ \mathfrak{Rtn}
\end{array}$$

To pass parameters, the caller will donate some of its local variables to the callee; they disappear from the caller's context and instead make up the callee's initial data state. Similarly, the procedure can return results by leaving one or more variables bound when it reaches the end state. (Thus our model naturally supports returning multiple values).

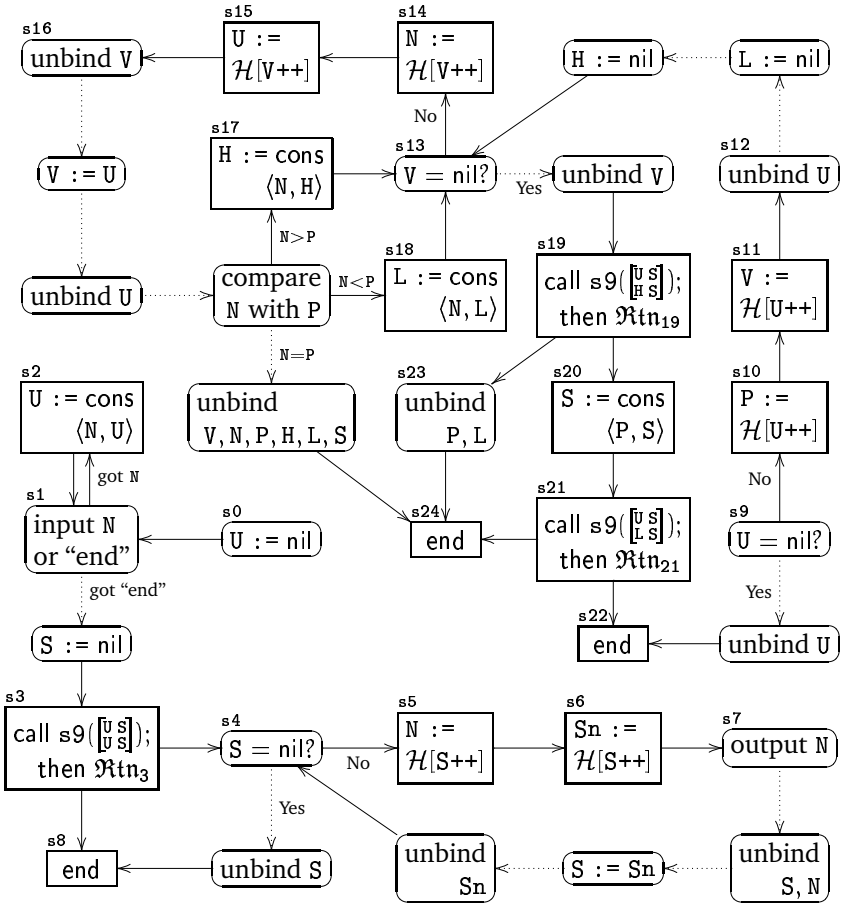
However, the caller and callee need not use the same *names* for the local variables that are used to pass data back and forth. (The caller *could* arrange to use the callee's names by surrounding its call state with misc states that rename variables appropriately, but it would be inconvenient to require that this should *always* be the case.) A (mutator) **argument list** $\mathfrak{Ma}$ is a map that describes the relation between caller-names and callee-names. We will stick to the convention that the *domain* of the map is the callee's names, and the *image* of the map is the caller's.

Those of the caller's variables that are not in the image of $\mathfrak{Ma}$ become temporarily unavailable while the procedure executes, but appear again when it returns.

As noted, the called procedure can have multiple exits; the call operation includes a **return map** that provides a separate return specification $(\mathfrak{Ma}, \mathfrak{Jmp})$ for each end state that is reachable from the the specified entry point. This pair specifies how to rename the return values, and where to continue execution afterwards.

Figure 3.3 shows an example of how all this fits together for a procedure with a single parameter and a single return value.

The reader may be worried that the syntax does not allow indirect calls where the target is not statically known. It turns out that this can be worked around, but we will defer a closer exploration of the options until we define the translation from ML to UHL in Section 4.1.



$$\mathfrak{Rtn}_3 = \{\mathtt{s22} \mapsto (\left[\begin{smallmatrix}\mathtt{S}\\ \mathtt{S}\end{smallmatrix}\right], \text{goto s4}), \mathtt{s24} \mapsto (\mathtt{[]}, \text{goto s8})\}$$

$$\mathfrak{Rtn}_{19} = \{\mathtt{s22} \mapsto (\left[\begin{smallmatrix}\mathtt{S}\\ \mathtt{S}\end{smallmatrix}\right], \text{goto s20}), \mathtt{s24} \mapsto (\mathtt{[]}, \text{goto s23})\}$$

$$\mathfrak{Rtn}_{21} = \{\mathtt{s22} \mapsto (\left[\begin{smallmatrix}\mathtt{S}\\ \mathtt{S}\end{smallmatrix}\right], \text{goto s22}), \mathtt{s24} \mapsto (\mathtt{[]}, \text{goto s24})\}$$

Informal guide to variable use:
- `P` is the pivot element.
- `N` is an element of the list.
- `U` and `V` iterate trough the unsorted list.
- `L` is the list of elements smaller than `P`.
- `H` is the list of elements larger than `P`.
- `S` is an accumulating parameter that eventually becomes the sorted list.

Figure 3.4: Quicksort as a uniform mutator.

Figure 3.4 shows everyone's favorite recursive algorithm Quicksort as a uniform mutator. This variant is semi-imperative; it treats heap data as immutable and uses lists instead of an array (because we don't model arrays and the memory management for array sorting is uninteresting anyway) but modifies the values of its local variables freely and uses an explicit loop rather than tail recursion to split its input list. In contrast to the Eratosthenes example, the quicksort program represents the empty list more conventionally, as a special value "nil" that is distinct from all pointers.

As an example of exception handling in our model, the sorting routine will throw an exception immediately if it discovers that the same number appears more than once in the list. `s22` is the normal return, which passes the sorted list by the name `S`. A return from `s24` signifies that an exception is thrown; no value is returned in that case.

Notice that if the call at `s19` results in an exception, the caller cannot pass the exception up the recursion stack immediately; it first has to unbind `P` and `L`, which were saved during the call. This corresponds to deallocating the caller's stack frame, which in a real implementation typically happens automatically as part of the exception-throwing mechanism. However, we still want to *model* this unbinding – for example, we might also want to deallocate the heap memory where the `L` list is allocated.

3.1.4 Ideal execution of uniform mutators

We now give a semantics for uniform mutators as they are **ideally** run *without* any region annotations, indeed without any memory reuse at all. This corresponds to what we can assume is the programmer's intended behavior of the program, without any thought of memory use.

The semantics describes an abstract machine that works in a series of nondeterministic transitions between configurations. The abstract machine is inherently nondeterministic – even if the uniform mutator contains no explicit nondeterminism, it is critical for our simulation results that allocations may return *any* choice of free heap cells (as opposed to, say, the one with the lowest address or the "next in line").

In addition to the various elements of UHL syntax that we have already defined, we use the following semantic objects:

$$
\begin{array}{lll}
\text{Heaps:} & H \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{W} & \\
\text{Stacks:} & \omega ::= (\mathfrak{Rtn}, \sigma) :: \omega \mid \bullet & \\
\text{Configurations:} & \mathcal{C} ::= (H, \sigma, s, \omega) & \\
 & \mid \mathsf{Stop} & \\
 & \mid \mathsf{Wrong} & \\
\text{Behaviors:} & \psi ::= \varphi_1, \ldots, \varphi_k & (k \geq 0) \\
 & \mid \varphi_1, \ldots, \varphi_k, \Downarrow & (k \geq 0) \\
 & \mid \varphi_1, \ldots, \varphi_k, \Uparrow & (k \geq 0)
\end{array}
$$

We will often abbreviate a sequence of events $\varphi_1, \cdots, \varphi_k$ by $\vec{\varphi}$. The empty sequence ($k = 0$) is written ϵ. The pseudo-events $\Downarrow$ and $\Uparrow$, which can only appear at the end of a behavior, denote ordinary program termination and divergence, respectively.

Definition 3.2. *Define the relation* $\mathcal{C} \xrightarrow[\mathcal{M}]{\varphi} \mathcal{C}'$ *by the following rules:*

$$\frac{\mathcal{M}(s) = \mathsf{misc} \cdots \square\ \mathfrak{Gs};\ \mathsf{goto}\, s' \square \cdots \qquad \sigma \xrightarrow{\varphi} \sigma' \in \mathfrak{Gs}}{(H, \sigma, s, \omega) \xrightarrow[\mathcal{M}]{\varphi} (H, \sigma', s', \omega)}\ \textit{IXMISC}$$

$$\frac{\mathcal{M}(s) = x\ :=\ \mathsf{cons}\,\langle x_0, \cdots, x_k\rangle;\ \mathsf{goto}\, s' \qquad H' = H \oplus \left[\begin{smallmatrix} a & \cdots & a+k \\ w_0 & \cdots & w_k \end{smallmatrix}\right]}{(H, \sigma \oplus \left[\begin{smallmatrix} x_0 & \cdots & x_k \\ w_0 & \cdots & w_k \end{smallmatrix}\right], s, \omega) \xrightarrow[\mathcal{M}]{\ominus} (H', \sigma \oplus \left[\begin{smallmatrix} x \\ a \end{smallmatrix}\right], s', \omega)}\ \textit{IXCONS}$$

$$\frac{\mathcal{M}(s) = x' := \mathcal{H}[x{+}{+}];\ \mathsf{goto}\, s' \qquad a \in \mathrm{Dom}\, H}{(H, \sigma \oplus \left[\begin{smallmatrix} x \\ a \end{smallmatrix}\right], s, \omega) \xrightarrow[\mathcal{M}]{\ominus} (H, \sigma \oplus \left[\begin{smallmatrix} x' & x \\ H(a) & a+1 \end{smallmatrix}\right], s', \omega)}\ \textit{IXREAD}$$

$$\frac{\mathcal{M}(s) = x' := \mathcal{H}[x{+}{+}];\ \mathsf{goto}\, s' \qquad a \in \mathbb{A} \setminus \mathrm{Dom}\, H}{(H, \sigma \oplus \left[\begin{smallmatrix} x \\ a \end{smallmatrix}\right], s, \omega) \xrightarrow[\mathcal{M}]{\ominus} \mathsf{Wrong}}\ \textit{IXREADWRONG}$$

$$\frac{\mathcal{M}(s) = \mathcal{H}[x{+}{+}] := x';\ \mathsf{goto}\, s' \qquad a \in \mathrm{Dom}\, H}{(H, \sigma \oplus \left[\begin{smallmatrix} x & x' \\ a & w \end{smallmatrix}\right], s, \omega) \xrightarrow[\mathcal{M}]{\ominus} (H \circledcirc \left[\begin{smallmatrix} a \\ w \end{smallmatrix}\right], \sigma \oplus \left[\begin{smallmatrix} x \\ a+1 \end{smallmatrix}\right], s', \omega)}\ \textit{IXWRITE}$$

$$\frac{\mathcal{M}(s) = \mathcal{H}[x{+}{+}] := x';\ \mathsf{goto}\, s' \qquad a \in \mathbb{A} \setminus \mathrm{Dom}\, H}{(H, \sigma \oplus \left[\begin{smallmatrix} x & x' \\ a & w \end{smallmatrix}\right], s, \omega) \xrightarrow[\mathcal{M}]{\ominus} \mathsf{Wrong}}\ \textit{IXWRITEWRONG}$$

$$\frac{\mathcal{M}(s) = \mathsf{call}\, s'(\mathfrak{Ma});\ \mathsf{then}\ \mathfrak{Rtn}}{(H, \sigma \oplus \sigma', s, \omega) \xrightarrow[\mathcal{M}]{\ominus} (H, \sigma \mathbin{\Diamond} \mathfrak{Ma}, s', (\mathfrak{Rtn}, \sigma') :: \omega)}\ \textit{IXCALL}$$

$$\frac{\mathcal{M}(s) = \mathsf{end} \qquad \mathfrak{Rtn}(s) = \big(\mathfrak{Ma}, \mathsf{goto}\, s'\big)}{(H, \sigma', s, (\mathfrak{Rtn}, \sigma) :: \omega) \xrightarrow[\mathcal{M}]{\ominus} (H, \sigma \oplus (\sigma' \mathbin{\Diamond} \mathfrak{Ma}^{-1}), s', \omega)}\ \textit{IXRETURN}$$

$$\frac{\mathcal{M}(s) = \mathsf{end}}{(H, [\,], s, \bullet) \xrightarrow[\mathcal{M}]{\ominus} \mathsf{Stop}}\ \textit{IXSTOP}$$

The "*IX*" in the rule names stand for "Ideal eXecution". Note how the use of $\oplus$ in *IXCONS* implies that the addresses a to $a + k$ are all fresh (*i.e.*, not in Dom H). In *IXCALL*, the use of $\Diamond$ implies that Dom σ must equal Img $\mathfrak{Ma}$, which uniquely determines how the initial data state splits into σ_1 and σ.

Definition 3.3. *A configuration $\mathcal{C}$ is **stuck** if $\mathcal{C} \neq \mathsf{Stop}$ and there is no $\mathcal{C}'$, φ such that $\mathcal{C} \xrightarrow[\mathcal{M}]{\varphi} \mathcal{C}'$.*

A stuck configuration represents some kind of internal error in the uniform mutator. The prototypical kind of stuckness is reaching a misc node with a data state σ that is not matched by any of its $\mathfrak{Gs}$; this usually corresponds to a run-time error in the human-made program that were translated to UHL. Other kinds of stuck configurations may arise with "malformed" uniform mutators; for example (H, σ, s, ω) is stuck if $\mathcal{M}(s)$ is not defined or σ does not contain the variables that are specified as operands in the mutator operation. In Section 3.1.5 we will define a simple discipline for uniform mutators that will prevent such spurious stuck configurations.

A special kind of stuck configuration, "Wrong", is reached if the program tries to access a heap cell that has not been allocated. We single out this outcome so that we can express the property that it cannot happen, which we call "memory safety".

Note that if the program tries to access the heap with something that is not an address at all (say, nil), it will get stuck at the heap operation instead of

Box 3.3—Alternatives to modeling I/O as side effects

Is it is really necessary to go to such trouble to define behaviors and expected observations? Obviously we need *some* model of program behavior if we are to speak meaningfully about it being preserved by specific memory-management strategies. But why not use a more lightweight model?

The main reason is that we want to be able to model interactive programs that do not terminate (because region-based memory management holds special promise for embedded and real-time systems), and most other behavioral models do not distinguish between nonterminating behaviors. But even for terminating programs the alternatives would still not be well suited to reasoning about region-based memory management.

A common choice in theoretical settings is ignore input and output completely. Programs (or program fragments) can still be said to "compute" specific functions, in a sense: One can simulate input by embedding the actual input as constants in the program, and for output, a program still has two distinct possible behaviors: It either halts or does not. By wrapping the body of the program in suitably chosen observation contexts, one can encode a multitude of output bits as termination behavior. (One has to be content that each output bit is *mathematically* well-defined; we can never actually observe that a wrapped program diverges).

This model has the advantage of simplicity and works well for many theoretical purposes. However, it means that region inference must be redone each time the input changes and each time one wants a different bit of the output. This is a real problem: Actual region-inference algorithms might well produce different annotations for the main program depending on which code eventually consumes its result.

A less minimalistic option is to define one's language such that a program defines a distinguished function that is applied to the program's input in order to run the program. However, this still fails our needs for speaking about region-based memory management. We surely want to allow the input and output to be composite data, as it would be ridiculous to expect the user to encode the entire input of programs that solve real-world problems as an atomic value such as an integer. But composite data would need to be heap-allocated. This is especially a problem for output, because the program would need to leave its output on the heap, which would prevent us from proving that an agent is well-behaved enough to eventually deallocate all the memory it has allocated. For input the converse problem would be how to distribute the pieces of a composite input value across regions such that the agent would be able to deallocate them once the mutator has finished inspecting them.

going Wrong. In practise both problems will be equally disastrous, of course, but the point of Wrong is to identify those errors that can reasonably blamed on the memory manager, and it is the mutator's own job to avoid following nil pointers.

Definition 3.4. *Define the relation* $\mathcal{M} \xrightarrow{\Psi\!\!\gg} \mathcal{C}$ *inductively by*

$$\frac{}{\mathcal{M} \xrightarrow{\epsilon\!\!\gg} ([\,],[\,],\mathtt{s0},\bullet)}\ \textit{IBSTART}$$

$$\frac{\mathcal{M} \xrightarrow{\vec{\varphi}\!\!\gg} \mathcal{C} \qquad \mathcal{C} \xrightarrow[\mathcal{M}]{\varphi} \mathcal{C}'}{\mathcal{M} \xrightarrow{\vec{\varphi},\varphi\!\!\gg} \mathcal{C}'}\ \textit{IBSTD} \qquad\qquad \frac{\mathcal{M} \xrightarrow{\vec{\varphi}\!\!\gg} \mathsf{Stop}}{\mathcal{M} \xrightarrow{\vec{\varphi},\Downarrow\!\!\gg} \mathsf{Stop}}\ \textit{IBTERM}$$

$$\frac{\mathcal{M} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathcal{C} \qquad \mathcal{C} \text{ is stuck}}{\mathcal{M} \xrightarrow{\vec{\varphi},\varphi}\!\!\!\!\rightarrow \mathcal{C}}\ \textit{IBSTUCK1} \qquad\qquad \frac{\mathcal{M} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathcal{C} \qquad \mathcal{C} \text{ is stuck}}{\mathcal{M} \xrightarrow{\vec{\varphi},\Downarrow}\!\!\!\!\rightarrow \mathcal{C}}\ \textit{IBSTUCK2}$$

Furthermore, let $\mathcal{M} \twoheadrightarrow \mathcal{C}$ *be an abbreviation for* $\exists \psi : \mathcal{M} \xrightarrow{\psi}\!\!\!\!\rightarrow \mathcal{C}$.

The two rules *IBSTUCK1* and *IBSTUCK2* give an "artificial" semantics to stuck states. They say that if the program gets stuck, anything can happen! This is a way for the semantics to give up; it means that the creator of the uniform mutator has not specified what is supposed to happen now, so an implementation that attempts to execute the program is at its liberty to do whatever it pleases, without risking to violate the semantics. [1]

Definition 3.5. *$\mathcal{M}$ is **memory safe** if $\mathcal{M} \not\twoheadrightarrow \mathsf{Wrong}$. It is **type safe** if there is no stuck $\mathcal{C}$ such that $\mathcal{M} \twoheadrightarrow \mathcal{C}$.*

Definition 3.6. *ψ is an **ideal behavior** of $\mathcal{M}$ if $\mathcal{M} \xrightarrow{\psi}\!\!\!\!\rightarrow \mathcal{C}$ for some $\mathcal{C}$. The set of $\mathcal{M}$'s ideal behaviors is notated $[\![\mathcal{M}]\!]$.*

(The "*IB*" in the rule names stand for "Ideal Behavior").

$[\![\mathcal{M}]\!]$ will be our primary model of $\mathcal{M}$'s intended semantics (see Box 3.3 for rationale).

The reader may have noticed that ideal behaviors never have the form $\vec{\varphi}, \Uparrow$, which according to the explanation we gave previously ought to signify divergence. That is deliberate, because we want to keep the definition of ideal behaviors clean of the co-inductive reasoning that would be necessary for speaking about divergence. However, because $[\![\mathcal{M}]\!]$ contains "prefixes" of the form $\vec{\varphi}$ in addition to completed computations of the form $\vec{\varphi}, \Downarrow$, we have enough information to recover a description of diverging behaviors:

Definition 3.7. *Given a set Ψ of behaviors, its **global abstraction** $\mathcal{A}bs(\Psi)$ is*

$$\begin{aligned} \mathcal{A}bs(\Psi) \quad = \quad & \{\, \|\vec{\varphi}\| \mid \vec{\varphi} \in \Psi \,\} \\ \cup \quad & \{\, \|\vec{\varphi}\|, \Downarrow \mid \vec{\varphi}, \Downarrow \in \Psi \,\} \\ \cup \quad & \{\, \vec{\varphi}', \Uparrow \mid \forall n \geq 0\ \exists \vec{\varphi} : \vec{\varphi}, \ominus^n \in \Psi \wedge \|\vec{\varphi}\| = \vec{\varphi}' \,\} \end{aligned}$$

where, for a sequence $\vec{\varphi}$ of events, $\|\vec{\varphi}\|$ denotes the same sequence with every occurrence of $\ominus$ removed.

Besides making divergence explicit, the global abstraction also hides differences in how many $\ominus$ ticks the mutator takes to decide on I/O actions. Thus, we can consider two mutators $\mathcal{M}_1$ and $\mathcal{M}_2$ to be "observationally equivalent" if $\mathcal{A}bs([\![\mathcal{M}_1]\!]) = \mathcal{A}bs([\![\mathcal{M}_2]\!])$.

[1] This viewpoint can also be explained by viewing the semantics as a "theory" according to the Popperian philosophy of science. In this philosophy the purpose of a theory is not so much to explain what *does* happen, as to define a large class of things that the theory asserts will *not* happen. If one of them happens nevertheless, the theory (or the observation) must be wrong. This view is especially convenient when we deal with nondeterministic programs. A semantics that says "this program *may* output 42" does not tell us much, whereas one that says "this program will *never* output 43" does: Then we know that the implementation (or the semantics) must be wrong if we see the program output 43. Therefore, when the semantics "gives up", the conservative choice is to say that "anything can happen now" rather than predicting an observable program crash.

Compare also the proverbial explanation of the treatment of certain program errors in the ISO standard for C: If the standard explicitly declares the meaning meaning of some construction to be "undefined", compilers are entitled to translate it to code that "causes demons to fly out your nose".

When $\vec{\varphi}, \Uparrow$ is in $\mathcal{A}bs([\![\mathcal{M}]\!])$ it means that it is possible for the mutator to do the I/O events $\vec{\varphi}$ and then run silently for an arbitrarily long time. That is almost the same as running *forever*, but not quite: A program that nondeterministically chooses a natural number $\mathfrak{n}$, then outputs a beep and finally counts to $\mathfrak{n}$ before terminating, will have "$\mathsf{beep}, \Uparrow$" as an expected observation even though it always terminates if it gets as far as to actually beep.

3.1.5 A scoping discipline for uniform mutators

In Section 3.1.2 we mentioned the rule that a bound variable cannot be bound to a new value until its value has been consumed either by a heap operation (or a procedure call) or explicitly in a misc node. The transition relation in the previous sections enforces such rules at run time; for example, *IXREAD* uses $\oplus$ to add x to the data state, so the program gets stuck if x is already bound.

We now give *static* rules to guard against such run-time errors. The rules will have the flavor of a very primitive "type system", but they do not really track the type of values, only which variables have some value bound to them.

Definition 3.8. *A uniform mutator $\mathcal{M}$ induces an* ***edge relation*** $\rhd_{\mathcal{M}}$ *in* $\operatorname{Dom}\mathcal{M}$ *by:*

$$s_1 \rhd_{\mathcal{M}} s_2 \quad \textit{iff} \quad \mathsf{goto}\, s_1 \in \mathcal{J}\big(\mathcal{M}(s_1)\big)$$

where the function $\mathcal{J}$ from mutator operations to sets of jumps is defined by

$$\begin{array}{ll}
\mathcal{J}\big(x := \mathsf{cons}\,\langle..\rangle;\ \mathfrak{Jmp}\big) & = \{\mathfrak{Jmp}\} \\
\mathcal{J}\big(x' := \mathcal{H}[x{+}{+}];\ \mathfrak{Jmp}\big) & = \{\mathfrak{Jmp}\} \\
\mathcal{J}\big(\mathcal{H}[x{+}{+}] := x';\ \mathfrak{Jmp}\big) & = \{\mathfrak{Jmp}\} \\
\mathcal{J}\big(\mathsf{misc}\ \mathfrak{Ss}_1; \mathfrak{Jmp}_1 \,\square\, \cdots \,\square\, \mathfrak{Ss}_k; \mathfrak{Jmp}_k\big) & = \{\mathfrak{Jmp}_1, ..., \mathfrak{Jmp}_k\} \\
\mathcal{J}\big(\mathsf{call}\ s(\mathfrak{Ma});\ \mathsf{then}\ \mathfrak{Rtn}\big) & = \{\mathfrak{Jmp} \mid (\mathfrak{Ma}', \mathfrak{Jmp}) \in \operatorname{Img}\mathfrak{Rtn}\} \\
\mathcal{J}\big(\mathsf{end}\big) & = \varnothing
\end{array}$$

$\rhd^*_{\mathcal{M}}$ is the reflexive, transitive closure of $\rhd_{\mathcal{M}}$.

Definition 3.9. *For each state s, let $\mathcal{E}_{\mathcal{M}}(s)$ be the set of* end *states that are reachable from s:*

$$\mathcal{E}_{\mathcal{M}}(s) = \big\{\, s' \;\big|\; s \rhd^*_{\mathcal{M}} s',\ \mathcal{M}(s') = \mathsf{end} \,\big\}$$

Note that, because we require flowcharts to be finite, $\mathcal{E}_{\mathcal{M}}(s)$ is finite and computable.

Definition 3.10. *Let $\mathcal{M}$ be a uniform mutator, and let the* ***scope invariant*** $\mathcal{S}$ *be a mapping from* $\operatorname{Dom}\mathcal{M}$ *to sets of variable names. $\mathcal{M}$ is* ***well-formed by*** $\mathcal{S}$ *if the following conditions are satisfied:*

1. *When $\mathcal{M}(s) = x := \mathsf{cons}\,\langle x_1, ..., x_n\rangle;\ \mathsf{goto}\, s'$, there must be a $\Sigma \subseteq \boxed{x}$ such that $\mathcal{S}(s) = \Sigma \uplus \{x_1, ..., x_n\}$ and $\mathcal{S}(s') = \Sigma \uplus \{x\}$.*
2. *When $\mathcal{M}(s) = x' := \mathcal{H}[x{+}{+}];\ \mathsf{goto}\, s'$, it must hold that $x \in \mathcal{S}(s)$ and $\mathcal{S}(s) \uplus \{x'\} = \mathcal{S}(s')$.*
3. *When $\mathcal{M}(s) = \mathcal{H}[x{+}{+}] := x';\ \mathsf{goto}\, s'$, it must hold that $x \in \mathcal{S}(s')$ and $\mathcal{S}(s) = \mathcal{S}(s') \uplus \{x'\}$.*
4. *When $\mathcal{M}(s) = \mathsf{misc}\ \mathfrak{Ss}_1;\ \mathsf{goto}\, s_1 \,\square\, \cdots \,\square\, \mathfrak{Ss}_k;\ \mathsf{goto}\, s_k$, then for each i and each $\sigma \xrightarrow{\varphi} \sigma' \in \mathfrak{Ss}_i$ it must hold that $\operatorname{Dom}\sigma = \mathcal{S}(s)$ and $\operatorname{Dom}\sigma' = \mathcal{S}(s_i)$.*

5. *When* $\mathcal{M}(s) = \mathsf{call}\ s'(\mathfrak{Ma});\ \mathsf{then}\ \{s_1 \mapsto (\mathfrak{Ma}_1, \mathsf{goto}\, s'_1), ..., s_n \mapsto (\mathfrak{Ma}_n, \mathsf{goto}\, s'_n)\}$, *then there must be a* Σ *such that*
   a. $\mathcal{S}(s) = \Sigma \uplus \operatorname{Img} \mathfrak{Ma}$
   b. $\mathcal{S}(s') = \operatorname{Dom} \mathfrak{Ma}$
   c. $\mathcal{E}_{\mathcal{M}}(s') = \{s_1, ..., s_n\}$
   d. *For each* i, $\mathcal{S}(s_i) = \operatorname{Dom} \mathfrak{Ma}_i$
   e. *For each* i, $\mathcal{S}(s'_i) = \Sigma \uplus \operatorname{Img} \mathfrak{Ma}_i$
6. $\mathcal{S}(\mathfrak{s}0) = \varnothing$.
7. *For each* $s \in \mathcal{E}_{\mathcal{M}}(\mathfrak{s}0)$, $\mathcal{S}(s) = \varnothing$.

Theorem 3.11. *Let* $\mathcal{M}$ *be well-formed by* $\mathcal{S}$. *If* $\mathcal{M} \twoheadrightarrow \mathcal{C}$ *and* $\mathcal{C}$ *is stuck, then it has one of the following forms:*

1. Wrong.
2. (H, σ, s, ω) *where* $\mathcal{M}(s) = \mathsf{misc}\ \mathfrak{Gs}_1;\ \mathfrak{Jmp}_1 \square \cdots \square \mathfrak{Gs}_k;\ \mathfrak{Jmp}_k$ *such that no* $\mathfrak{Gs}_i$ *contains a step of the form* $\sigma \xrightarrow{\varphi} \sigma'$.
3. (H, σ, s, ω) *where* $\mathcal{M}(s) = x' := \mathcal{H}[x{+}{+}]$ *and* $\sigma(x) \notin \mathbb{A}$.
4. (H, σ, s, ω) *where* $\mathcal{M}(s) = \mathcal{H}[x{+}{+}] := x'$ *and* $\sigma(x) \notin \mathbb{A}$.

The proof uses the following auxiliary relations:

Definition 3.12. *Define the relations* $\mathcal{S}, \mathcal{M} \vdash \mathcal{C}$ *and* $\mathcal{S}, \mathcal{M} \vdash \omega \rhd s$ *by*

$$\frac{}{\mathcal{S}, \mathcal{M} \vdash \mathsf{Stop}}\ \textit{ISCSTOP} \qquad \frac{}{\mathcal{S}, \mathcal{M} \vdash \mathsf{Wrong}}\ \textit{ISCWRONG}$$

$$\frac{\operatorname{Dom}\sigma = \mathcal{S}(s) \qquad \mathcal{S}, \mathcal{M} \vdash \omega \rhd s}{\mathcal{S}, \mathcal{M} \vdash (H, \sigma, s, \omega)}\ \textit{ISCSTD} \qquad \frac{\forall s' \in \mathcal{E}_{\mathcal{M}}(s) : \mathcal{S}(s') = \varnothing}{\mathcal{S}, \mathcal{M} \vdash \bullet \rhd s}\ \textit{ISSEMPTY}$$

$$\frac{\forall s' \in \mathcal{E}_{\mathcal{M}}(s) : \begin{cases} \mathcal{S}(s') = \operatorname{Dom} \mathfrak{Ma} \\ \mathcal{S}(s'') = \operatorname{Dom}\sigma \uplus \operatorname{Img} \mathfrak{Ma} \\ \mathcal{S}, \mathcal{M} \vdash \omega \rhd s'' \\ \quad \text{where } (\mathfrak{Ma}, \mathsf{goto}\, s'') = \mathfrak{Rtn}(s') \end{cases}}{\mathcal{S}, \mathcal{M} \vdash (\mathfrak{Rtn}, \sigma) :: \omega \rhd s}\ \textit{ISSFRAME}$$

The "*ISC*" and "*ISS*" in the rule names stand for "Ideal Scope-invariant for Configurations/Stacks".

Lemma 3.13. *If* $\mathcal{S}, \mathcal{M} \vdash \omega \rhd s$ *and* $s \rhd_{\mathcal{M}} s'$, *then* $\mathcal{S}, \mathcal{M} \vdash \omega \rhd s'$.

Proof. Immediate from the definitions of $\rhd^*_{\mathcal{M}}$ and $\mathcal{E}_{\mathcal{M}}(\cdot)$. □

Proposition 3.14 (Subject reduction, level 0). *If* $\mathcal{S}, \mathcal{M} \vdash \mathcal{C}$ *and* $\mathcal{C} \xrightarrow[\mathcal{M}]{\varphi} \mathcal{C}'$. *Then* $\mathcal{S}, \mathcal{M} \vdash \mathcal{C}'$.

Proof. Easy, by comparing each case in the definition of $\xrightarrow[\mathcal{M}]{\varphi}$ with the corresponding case in Definition 3.10 and then applying Lemma 3.13 for the stack part of the configuration. □

Proof of Theorem 3.11. First, prove $\mathcal{S}, \mathcal{M} \vdash \mathcal{C}$ by rule induction on $\mathcal{M} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}$. The base case *IBSTART* is guaranteed by Definition 3.10(6,7). For *IBSTD*, use the induction hypothesis and Proposition 3.14, and for the other rules the induction hypothesis directly.

By definition, Stop is not stuck, so assume that $\mathcal{C} = (H, \sigma, s, \omega)$. If $\mathcal{M}(s) =$ end, then the premise that $\mathcal{S}, \mathcal{M} \vdash \omega \rhd s$ guarantees that either *IXRETURN* or *IXSTOP* will apply. Otherwise, inspect the appropriate case in Definition 3.10 and combine it with the respective reduction rules. □

3.2 Region annotations for universal mutators

We will now define a generic scheme for adding region annotations to a uniform mutator. As noted earlier, the region annotations will essentially be the same as the "HMN model" of Section 2.4. Indeed, annotated UHL will be a very close relative of the REGWHILE calculus in chapter 6 of Niss [2002]. Our notation here does not completely equal that used by Niss, due to technical issues with our handling of unstructured flow graphs and multiple-entry–multiple-exit procedures.

The basic ideas of the agent programming language are:

- Each cons state in the uniform mutator is annotated with a **region variable** that defines which region to do the allocation in.
- Edges in the flowchart may be annotated with **region operations** that specify the creation and deallocation of regions at run time, and maintain the bindings between regions and region variables.
- A region on the heap can be referred to by different region variables at different times: Region operations can move region references between region variables.
- A region on the heap can be referred to by different region variables at *the same* time. Each region maintains a **reference count** at runtime and gets deallocated when the reference count eventually reaches zero.
- When the mutator calls a procedure, the agent can pass a number of region references as **region parameters** to the agent code for the procedure in parallel with the mutator's ordinary parameters. When the procedure returns, a number of region references may be returned too, in parallel with the mutator's ordinary return values.
- **Uncounted region variables** are a special restricted form of region variables. They can only be assigned to as part of a procedure call (and disappear silently at returns). This restriction will allow them to be safely created and forgotten *without* adjusting the reference count of the region they point to.

Figure 3.5 shows the syntax of region-annotated UHL. A (possibly empty) list of region annotations has been added to each $\mathfrak{Jmp}$, which represents the edges in the flowchart (compare Definition 3.8) – which is the reason why we defined $\mathfrak{Jmp}$ as a syntactic class in UHL in the first place.

The syntax for allocation has been extended with an "at annotation" which is normally at ρ, giving the region in which the heap cells should be allocated. The alternative annotation nowhere means that the allocation should be omitted completely. It is there for technical reasons; sometimes the region annotation

• **History of notation:** "Uncounted region variables" are the same as the "constant region parameters" of the HMN system. My experience is that the "constant" terminology has been difficult to explain, so I chose here to base my terminology on the reference counts instead of the constancy.

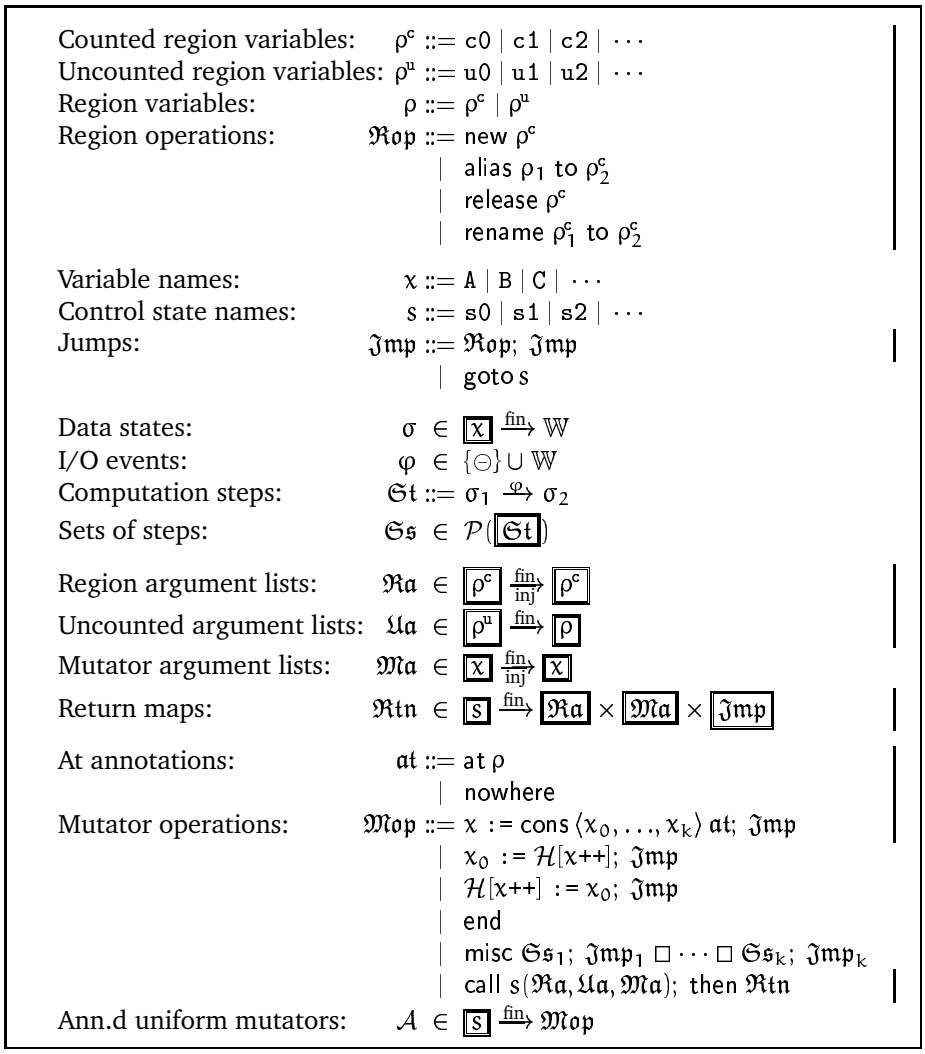


Figure 3.5: Syntax of annotated uniform mutators. The part of the syntax that differ from that of plain uniform mutators are marked with a vertical line on the right.

algorithm will depend on being able to generate it in the unlikely case that the program never uses the allocated value.

We recognize four different region operations:

1. new allocates a new region with reference count 1 and binds a reference to it to a region variable.
2. alias increases the reference count of a region by one, and binds a new reference to it to a region variable.
3. release unbinds a region variable and decreases the reference count of the region it was bound to by one. If the count reaches zero, the region is deallocated.

4. rename moves region references between different region variables. It is not strictly necessary from a purist's viewpoint, because its effects can always be simulated by the other operations: "rename ρ_1^c to ρ_2^c" is equivalent to "alias ρ_1^c to ρ_2^c; release ρ_1^c". However, the rename notation makes it explicit that there is no net reference-count manipulation going on. This makes it slightly easier to reason about a rename than an alias–release combination.

None of the region operations can bind or unbind an uncounted region variable. Instead, uncounted region variables can be bound as part of a procedure call, and disappear again when the procedure returns. We enforce this rule in the syntax by letting uncounted region variables be lexically different from counted ones. (In contrast, the original HMN system had only one kind of region variables, and the difference between "constant region parameters" and other region variables was enforced by side conditions in the type system. Later we have come to the conclusion that it is cleaner to distinguish them syntactically.)

Note that the source of an alias operation *can* be an uncounted variable, as it is neither bound nor unbound by the operation.

Passing region references into and out of procedures is somewhat complex, but we have already mastered the formal idiosyncrasies, because the notation for parameter-passing in plain UHL was cleverly chosen to parallel the features we need for regions.

Each ordinary argument-renaming map $\mathfrak{Ma}$ (for either calls or returns) is paralleled by a $\mathfrak{Ra}$ that does the same job for region variables. Everything that holds for $\mathfrak{Ma}$ and ordinary variables is true for $\mathfrak{Ra}$ and (counted) region variables, too. In particular, the region variables in Img $\mathfrak{Ma}$ disappear from the caller's context as part of the call, so passing a region parameter entails no reference-count manipulation.

A third argument mapping, $\mathfrak{Ua}$ is given only at the call and not at the return. It serves to initialize the ρ^u bindings for the callee's agents. Because an uncounted region variable is, well, uncounted, passing this kind of parameter does not involve manipulating reference counts. Instead it will be kept alive by the reference count of the caller-variable it was initialized from – the variables in Img $\mathfrak{Ua}$ do *not* disappear from the caller's context. A single caller-variable can be used to initialize several ρ^us: The grammar does not require $\mathfrak{Ua}$ to be injective. (This is the reason for our convention that parameter-list maps map callee-variables to caller-variables). However Img $\mathfrak{Ua}$ must be disjoint from Img $\mathfrak{Ra}$ – otherwise the callee would risk releasing one of the reference counts that its ρ^us depend on.

It may be instructive to compare our notation for procedure calls with the one in the HMN system. Take the following call to a single-exit procedure:

$$\text{call s56}\left(\begin{bmatrix}\text{c1}\\ \text{c0}\end{bmatrix}, \begin{bmatrix}\text{u2 u5}\\ \text{c7 u8}\end{bmatrix}, \begin{bmatrix}\text{X}\\ \text{A}\end{bmatrix}\right);\ \text{then}\ \{\text{s82}(\ \begin{bmatrix}\text{c3 c6}\\ \text{c9 c4}\end{bmatrix}, \begin{bmatrix}\text{Y}\\ \text{C}\end{bmatrix}, \text{release c7; goto s5})\}$$

(which might be an annotated version of the call in Figure 3.3). In the ML-based HMN system (Section 2.4), the call would be notated

$$\big(\text{S56 } \mathbf{[c:}\ \rho_7, \rho_8; \mathbf{i:}\ \rho_0; \mathbf{o:}\ \rho_9, \rho_4\mathbf{]}\ \text{A}\big)\ [\![\mathbf{release}\ \rho_7]\!]$$

• **History of notation:** In the HMN system, renaming of regions is written simply as "ρ_2 := ρ_1". I changed the notation to "rename ρ_1 to ρ_2" to emphasize that the operation unbinds ρ_1, in contrast to the ":=" nodes on Figure 3.2 etc. The *meaning* of the operation has not changed.

where the callee's names $\mathtt{X}$, ρ_1, ρ_2, ρ_3, ρ_5, and ρ_6 are given in the definition of the procedure instead of in the call, and the ordinary return value implicitly becomes the value of the expression instead of being assigned to $\mathtt{C}$.

The REGWHILE notation of Niss [2002, chapter 6] is a little closer to the UHL model. It writes the same call as

$$(\rho_9, \rho_4; \mathtt{C}) \;:=\; \textbf{call}\ \mathtt{S56}[\rho_7, \rho_8](\rho_0; \mathtt{A})\ ;\ \textbf{release}\ \rho_7$$

3.2.1 What is an agent?

It is easy to imagine how one can recover the "underlying" uniform mutator from an annotated one. One simply removes all of the region-related syntax:

$$\begin{aligned}
\|\mathfrak{Rop};\ \mathfrak{Jmp}\|_{\text{Jmp}} &= \|\mathfrak{Jmp}\|_{\text{Jmp}} \\
\|\mathsf{goto}\ \mathtt{s}\|_{\text{Jmp}} &= \mathsf{goto}\ \mathtt{s} \\
\left\|\{\ldots, \mathtt{s}_i \mapsto (\mathfrak{Ra}_i, \mathfrak{Ma}_i, \mathfrak{Jmp}_i), \ldots\}\right\|_{\text{Rtn}} &= \{\ldots, \mathtt{s}_i \mapsto (\mathfrak{Ma}_i, \|\mathfrak{Jmp}_i\|_{\text{Jmp}}), \ldots\} \\
\|\mathtt{x} := \mathsf{cons}\,\langle \mathtt{x}_0, \ldots, \mathtt{x}_k\rangle\ \mathsf{at}\ \rho;\ \mathfrak{Jmp}\|_{\text{Mop}} &= \mathtt{x} := \mathsf{cons}\,\langle \mathtt{x}_0, \ldots, \mathtt{x}_k\rangle\ ;\ \|\mathfrak{Jmp}\|_{\text{Jmp}} \\
\|\mathtt{x}' := \mathcal{H}[\mathtt{x}{+}{+}];\ \mathfrak{Jmp}\|_{\text{Mop}} &= \mathtt{x}' := \mathcal{H}[\mathtt{x}{+}{+}];\ \|\mathfrak{Jmp}\|_{\text{Jmp}} \\
\|\mathcal{H}[\mathtt{x}{+}{+}] := \mathtt{x}';\ \mathfrak{Jmp}\|_{\text{Mop}} &= \mathcal{H}[\mathtt{x}{+}{+}] := \mathtt{x}';\ \|\mathfrak{Jmp}\|_{\text{Jmp}} \\
\|\mathsf{end}\|_{\text{Mop}} &= \mathsf{end} \\
\|\mathsf{misc} \cdots \square\ \mathfrak{Ss}_i;\ \mathfrak{Jmp}_i \square \cdots\|_{\text{Mop}} &= \mathsf{misc} \cdots \square\ \mathfrak{Ss}_i;\ \|\mathfrak{Jmp}_i\|_{\text{Jmp}} \square \cdots \\
\|\mathsf{call}\ \mathtt{s}(\mathfrak{Ra}, \mathfrak{Ua}, \mathfrak{Ma});\ \mathsf{then}\ \mathfrak{Rtn}\|_{\text{Mop}} &= \mathsf{call}\ \mathtt{s}(\mathfrak{Ma});\ \mathsf{then}\ \|\mathfrak{Rtn}\|_{\text{Rtn}} \\
\|\mathcal{A}\| &= \|\cdot\|_{\text{Mop}} \circ \mathcal{A}
\end{aligned}$$

Conversely, an annotated uniform mutator is completely known if we have

a) The underlying uniform mutator $\mathcal{M} = \|\mathcal{A}\|$.

b) A (possibly empty) sequence of region operations for each flowchart edge in $\mathcal{M}$.

c) An at annotation for each cons node in $\mathcal{M}$.

d) A $\mathfrak{Ra}$ and $\mathfrak{Ua}$ for each call node in $\mathcal{M}$ and a $\mathfrak{Ra}$ for each of its returns.

Intuitively, the **agent** comprises items (b), (c), and (d) in this list. We could define some kind of syntax for "naked" agents and define formally how to combine an agent with a mutator to get an annotated mutator, but doing so would offer little or no insight in return for the symbolism it would require. Instead, for the purpose of our *formal* development, we simply define

Definition 3.15. *An **agent** for a uniform mutator $\mathcal{M}$ is given by an annotated uniform mutator $\mathcal{A}$ such that $\|\mathcal{A}\| = \mathcal{M}$.*

In informal discussions we will usually stick to the intuition that the agent "really" consists of the *difference* between $\mathcal{A}$ and $\|\mathcal{A}\|$.

• **History of notation:** In the HMN syntax for call annotations, the uncounted region parameters come before the counted inputs, because that matches the type-theoretic description from the callee's point of view, where the quantifiers for the uncounted ("constant") regions need to come before the ones for the counted one. However, this reasoning is not really appropriate here, where we are developing a universal agent language without reference to a specific region type system. Therefore I have chosen to notate the counted region parameters first, which matches the operational intuition at the caller's end: First the counted region parameters disappear from the caller's context, and afterwards values for the uncounted ones may be selected from the region variables that still remain.

3.2.2 Managed execution of uniform mutators

We now give a semantics for the combination of a uniform mutator and an agent for it – we shall say that the mutator's execution is **managed** by the agent. The structure of the semantics will be close to the ideal semantics of Section 3.1.4, but of course extended with a model for region operations.

The major difference between managed and ideal semantics is that cells are sometimes *deallocated* from the heap in the managed semantics (in the ideal semantics the domain of the heap is always non-decreasing). Thus, if the mutator attempts to access a deallocated heap cell it will go Wrong instead of reading whatever was in the cell the last time it was in use. This is important for reasoning about soundness, but may look slightly unrealistic – usually region managers just reuse memory internally and never give it back to the operating system – but remember (Definition 3.7) that the semantic interpretation of Wrong is actually just that "anything may happen now", including getting the previous value of the cell. So our choice of going Wrong for deallocated cells actually just makes the semantics slightly imprecise, not incorrect.

The semantic objects used in the semantics are

Region names:	$r ::= \mathtt{r0} \mid \mathtt{r1} \mid \mathtt{r2} \mid \cdots$	
Heaps:	$H \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{W}$	
Region-manager states:	$L \in \boxed{r} \xrightarrow{\text{fin}} \mathbb{N}_+ \times \mathcal{P}_{\text{fin}}(\mathbb{A})$	
Region environments:	$R \in \boxed{\rho} \xrightarrow{\text{fin}} \boxed{r}$	
Stacks:	$\omega ::= (\mathfrak{Rtn}, R, \sigma) :: \omega$	
	$\mid \bullet$	
Configurations:	$\mathcal{C} ::= (H, L, R, \sigma, \mathfrak{Jmp}, \omega)$	
	$\mid$ Stop	
	$\mid$ Wrong	
Behaviors:	$\psi ::= \varphi_1, \cdots, \varphi_k$	$(k \geq 0)$
	$\mid \varphi_1, \cdots, \varphi_k, \Downarrow$	$(k \geq 0)$
	$\mid \varphi_1, \cdots, \varphi_k, \Uparrow$	$(k \geq 0)$

We do not model the inner workings of the region manager, only its extensional behavior. An L simply records our expectation about how the region manager will behave in the future – the fact that $L(r) = (n, A)$ means that the region manager is supposed not to reuse any of the addresses in A until n release operations have been done on r.

Definition 3.16. *The following rules for the relation* $L, R \xrightarrow[\mathfrak{Rop}]{} L', R'$ *define the semantics of region operations:*

$$\frac{}{L, R \xrightarrow[\text{new } \rho^c]{} L \oplus \{r \mapsto (1, \varnothing)\}, R \oplus \left[\begin{smallmatrix}\rho^c \\ r\end{smallmatrix}\right]} \textit{MXNEW}$$

$$\frac{}{L \oplus \{\not{r} \mapsto (n, A)\}, R \quad \oplus \left[\begin{smallmatrix}\rho \\ r\end{smallmatrix}\right] \xrightarrow[\text{alias } \rho \text{ to } \rho^c]{} L \oplus \{\not{r} \mapsto (n+1, A)\}, R \quad \oplus \left[\begin{smallmatrix}\rho\ \rho^c \\ r\ r\end{smallmatrix}\right]} \textit{MXALIAS}$$

$$\frac{n \geq 2}{L \oplus \{\not{r} \mapsto (n, A)\}, R \quad \oplus \left[\begin{smallmatrix}\rho^c \\ r\end{smallmatrix}\right] \xrightarrow[\text{release } \rho^c]{} L \oplus \{\not{r} \mapsto (n-1, A)\}, R} \textit{MXRELEASEN}$$

$$\frac{}{L \oplus \{\not{r} \mapsto (1, A)\}, R \quad \oplus \left[\begin{smallmatrix}\rho^c \\ r\end{smallmatrix}\right] \xrightarrow[\text{release } \rho^c]{} L, R} \textit{MXRELEASE1}$$

$$\frac{}{L, R \oplus \left[\begin{smallmatrix}\rho_1^c\\ r\end{smallmatrix}\right] \xrightarrow[\text{rename } \rho_1^c \text{ to } \rho_2^c]{} L, R \oplus \left[\begin{smallmatrix}\rho_2^c\\ r\end{smallmatrix}\right]}\ \textit{MXRENAME}$$

The "*MX*" in the rule names stand for "Managed eXecution".

One aspect of the region operations' semantics that is not captured by these rules is the actual deallocation of heap cells when a region's reference count reaches zero. This is in keeping with our general principle of not modeling the internals of the region manager. Instead, the definitions that follow will simply assert that the region manager, somehow, deallocates cells once it has no obligation to keep them around. Therefore the cells will be deallocated *implicitly* when *MXRELEASE1* removes the region from the manager state. (Rule *MXROP* below).

Definition 3.17. *The* ***footprint*** *of a region-manager state* L *is the union of the second components of* $\operatorname{Img} L$.

Definition 3.18. *The* ***restriction*** *of a heap* H *to a region-manager state* L *means* H *with bindings outside* L*'s footprint discarded. The restriction is notated* $H|_L$.

Definition 3.19. *Define the relations* $\mathcal{C} \xrightarrow[\mathcal{A}]{} \mathcal{C}$ *and* $\mathcal{C} \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}'$ *by the following rules:*

$$\frac{L, R \xrightarrow[\mathfrak{Rop}]{} L', R'}{(H, L, R, \sigma, \mathfrak{Rop}; \mathfrak{Jmp}, \omega) \xrightarrow[\mathcal{A}]{} (H|_{L'}, L', R', \sigma, \mathfrak{Jmp}, \omega)}\ \textit{MXROP}$$

$$\frac{\mathcal{A}(s) = \text{misc} \cdots \square\ \mathfrak{Ss}; \mathfrak{Jmp} \square \cdots \qquad \sigma \xrightarrow{\varphi} \sigma' \in \mathfrak{Ss}}{(H, L, R, \sigma, \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\varphi} (H, L, R, \sigma, \mathfrak{Jmp}, \omega)}\ \textit{MXMISC}$$

$$\frac{\begin{array}{c}\mathcal{A}(s) = x := \text{cons}\langle x_0, \cdots, x_k\rangle \text{ at } \rho;\ \mathfrak{Jmp} \qquad H' = H \oplus \left[\begin{smallmatrix}a & \cdots & a+k\\ w_0 & \cdots & w_k\end{smallmatrix}\right]\\ R(\rho) = r \qquad L(r) = (n, A) \qquad L' = L \circledcirc \{r \mapsto (n, A \cup \{a, ..., a+k\})\}\end{array}}{(H, L, R, \sigma \oplus \left[\begin{smallmatrix}x_0 & \cdots & x_k\\ w_0 & \cdots & w_k\end{smallmatrix}\right], \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\ominus} (H', L', R, \sigma \oplus \left[\begin{smallmatrix}x\\ a\end{smallmatrix}\right], \mathfrak{Jmp}, \omega)}\ \textit{MXCONS}$$

$$\frac{\mathcal{A}(s) = x := \text{cons}\langle x_0, \cdots, x_k\rangle \text{ nowhere};\ \mathfrak{Jmp} \qquad a \in \mathbb{A}}{(H, L, R, \sigma \oplus \left[\begin{smallmatrix}x_0 & \cdots & x_k\\ w_0 & \cdots & w_k\end{smallmatrix}\right], \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\ominus} (H, L, R, \sigma \oplus \left[\begin{smallmatrix}x\\ a\end{smallmatrix}\right], \mathfrak{Jmp}, \omega)}\ \textit{MXNOWHERE}$$

$$\frac{\mathcal{A}(s) = x' := \mathcal{H}[x{+}{+}];\ \mathfrak{Jmp} \qquad a \in \operatorname{Dom} H}{(H, L, R, \sigma \oplus \left[\begin{smallmatrix}x\\ a\end{smallmatrix}\right], \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\ominus} (H, L, R, \sigma \oplus \left[\begin{smallmatrix}x' & x\\ H(a) & a+1\end{smallmatrix}\right], \mathfrak{Jmp}, \omega)}\ \textit{MXREAD}$$

$$\frac{\mathcal{A}(s) = x' := \mathcal{H}[x{+}{+}];\ \mathfrak{Jmp} \qquad a \in \mathbb{A} \setminus \operatorname{Dom} H}{(H, L, R, \sigma \oplus \left[\begin{smallmatrix}x\\ a\end{smallmatrix}\right], \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\ominus} \text{Wrong}}\ \textit{MXREADWRONG}$$

$$\frac{\mathcal{A}(s) = \mathcal{H}[x{+}{+}] := x';\ \mathfrak{Jmp} \qquad a \in \operatorname{Dom} H}{(H, L, R, \sigma \oplus \left[\begin{smallmatrix}x & x'\\ a & w\end{smallmatrix}\right], \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\ominus} (H \circledcirc \left[\begin{smallmatrix}a\\ w\end{smallmatrix}\right], L, R, \sigma \oplus \left[\begin{smallmatrix}x\\ a+1\end{smallmatrix}\right], \mathfrak{Jmp}, \omega)}\ \textit{MXWRITE}$$

$$\frac{\mathcal{A}(s) = \mathcal{H}[x{+}{+}] := x';\ \mathfrak{Jmp} \qquad a \in \mathbb{A} \setminus \operatorname{Dom} H}{(H, L, R, \sigma \oplus \left[\begin{smallmatrix}x & x'\\ a & w\end{smallmatrix}\right], \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\ominus} \text{Wrong}}\ \textit{MXWRITEWRONG}$$

$$\frac{\begin{array}{c}\mathcal{A}(s) = \text{call } s'(\mathfrak{Ra}, \mathfrak{Ua}, \mathfrak{Ma});\ \text{then } \mathfrak{Rtn}\\ R'' = (R \dagger \mathfrak{Ra}) \oplus (R' \circ \mathfrak{Ua}) \qquad \omega' = (\mathfrak{Rtn}, R', \sigma') :: \omega\end{array}}{(H, L, R \oplus R', \sigma \oplus \sigma', \text{goto } s, \omega) \xrightarrow[\mathcal{A}]{\ominus} (H, L, R'', \sigma \dagger \mathfrak{Ma}, \text{goto } s', \omega')}\ \textit{MXCALL}$$

Box 3.4—Different semantic models of the region manager

My semantic model of the interaction between the region manager and the heap is more abstract and indirect than the models commonly used in the literature.

Tofte and Talpin [1994, 1997] let addresses have the form (r, o), where r is a region name and o is an "offset" within the region. They then work with stores of type $\boxed{r} \xrightarrow{\text{fin}} \boxed{o} \xrightarrow{\text{fin}} \mathbb{W}$ but implicitly reinterpret the store as $\boxed{r} \times \boxed{o} \xrightarrow{\text{fin}} \mathbb{W}$ for actual heap accesses. The region operations work mostly in terms of the curried interpretation.

This representation makes it notationally easy to deallocate a region – one just removes r from the domain of the uncurried store – and allows one to represent the set of existing regions implicitly as the domain of the store (there is a difference between an r that maps to the the empty map from offsets to data and an r that is not in the store at all). However, the address representation means that in cases like

```
new c1
new c2
X := cons ⟨0, 1, 2⟩ at c2
release c2
Y := cons ⟨3, 4, 5⟩ at c1
```

the semantics will claim that it is impossible for `X` and `Y` to equal each other. (They must have different r parts). However, with the intended implementation of the region manager, it is quite possible for `Y` to be allocated in the same addresses from where `X`'s tuple was just deallocated. This difference could become crucial for reasoning about region-based memory management for languages with pointer comparison, which is why I do not use the (r, o) representation.

Fritz Henglein has suggested [personal communication] to use the (r, o) model as the basis for an alternative implementation of the region manager. In this implementations, regions are *continuous* pieces of memory that the region manager is allowed to move around in memory, because any access to the heap goes through the region manager: The agent supplies r and o, and the region manager replies with the *current* address of the indicated cell. The mutator itself actually manipulates only the o parts, and the agent provides the correct r whenever it is needed. Thus the model requires region annotations on read and write operations in addition to `cons` operations.

An entirely different technique is employed by Calcagno et al. [2002]; Calcagno [2001]; Helsen and Thiemann [2000]. They use a syntactic reduction semantics without a store. Heap-allocated values are represented by special terms that include a region variable. Thus the model does not use a separate concept of region names, instead region variables double as the representation of actually existing regions.

This scheme has the advantage that it allows very simple syntactic proofs of the safety of the region type system. On the other hand, it somewhat removed from the operational intuition, and it is difficult to extend to systems with destructive or other features that require a separate store, because the technique depends on the lexical scoping for **letregion**-bound variables, which would break down if values with embedded region variables get written to an external store.

$$\frac{\mathcal{A}(s) = \mathsf{end} \qquad \mathfrak{Rtn}(s) = (\mathfrak{Ra}, \mathfrak{Ma}, \mathfrak{Jmp}) \qquad \mathrm{Img}\, R'_u \subseteq \boxed{\rho^u} \qquad R'' = R \oplus (R'_c \multimap \mathfrak{Ra}^{-1}) \qquad \sigma'' = \sigma \oplus (\sigma' \multimap \mathfrak{Ma}^{-1})}{(H, L, R'_c \oplus R'_u, \sigma', \mathsf{goto}\, s, (\mathfrak{Rtn}, R, \sigma) :: \omega) \xrightarrow[\mathcal{A}]{\ominus} (H, L, R'', \sigma'', \mathfrak{Jmp}, \omega)}\ \textit{MXRETURN}$$

$$\frac{\mathcal{A}(s) = \mathsf{end}}{(H, L, [\,], [\,], \mathsf{goto}\, s, \bullet) \xrightarrow[\mathcal{A}]{\ominus} \mathsf{Stop}}\ \textit{MXSTOP}$$

Note that the region annotation on cons operations does not constrain *where* in the heap the new cells are allocated; the $\mathfrak{a}$ in *MXCONS* is still chosen nondeterministically. The semantics merely update the model of the region manager to the effect that the new cells will be deallocated together with the specified region. In an actual implementation, of course, it will be the region manager's job to choose an $\mathfrak{a}$, but it is important that our semantics does not constrain how it does it.

Definition 3.20. *Define the relation* $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\rightarrow \mathcal{C}$ *by*

$$\frac{}{\mathcal{A} \xrightarrow{\epsilon}\!\!\!\!\rightarrow ([\,], [\,], [\,], [\,], \mathsf{goto}\, s0, \bullet)}\ \textit{MBSTART} \qquad \frac{\mathcal{A} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathcal{C} \qquad \mathcal{C} \xrightarrow[\mathcal{A}]{} \mathcal{C}'}{\mathcal{A} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathcal{C}'}\ \textit{MBRGN}$$

$$\frac{\mathcal{A} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathcal{C} \qquad \mathcal{C} \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}'}{\mathcal{A} \xrightarrow{\vec{\varphi},\varphi}\!\!\!\!\rightarrow \mathcal{C}'}\ \textit{MBSTD} \qquad \frac{\mathcal{A} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathsf{Stop}}{\mathcal{A} \xrightarrow{\vec{\varphi},\Downarrow}\!\!\!\!\rightarrow \mathsf{Stop}}\ \textit{MBTERM}$$

$$\frac{\mathcal{A} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathcal{C} \qquad \mathcal{C} \text{ is stuck}}{\mathcal{A} \xrightarrow{\vec{\varphi},\varphi}\!\!\!\!\rightarrow \mathcal{C}}\ \textit{MBSTUCK1} \qquad \frac{\mathcal{A} \xrightarrow{\vec{\varphi}}\!\!\!\!\rightarrow \mathcal{C} \qquad \mathcal{C} \text{ is stuck}}{\mathcal{A} \xrightarrow{\vec{\varphi},\Downarrow}\!\!\!\!\rightarrow \mathcal{C}}\ \textit{MBSTUCK2}$$

and let $\mathcal{A} \twoheadrightarrow \mathcal{C}$ *abbreviate* $\exists \psi : \mathcal{A} \xrightarrow{\Psi}\!\!\!\!\rightarrow \mathcal{C}$.

Definition 3.21. *An agent* $\mathcal{A}$ *is* ***region safe*** *if* $\mathcal{A} \not\twoheadrightarrow \mathsf{Wrong}$.

Definition 3.22. ψ *is a* ***managed behavior*** *of* $\mathcal{A}$ *if* $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\rightarrow \mathcal{C}$ *for some* $\mathcal{C}$. *The set of* $\mathcal{A}$*'s managed behaviors is notated* $[\![\mathcal{A}]\!]$.

(The "*MB*" in the rule names stand for "Managed Behavior").

In the rest of this subsection, we will state some basic invariants of the semantics. First we will relate H to L:

Proposition 3.23. *Assume that* $\mathcal{A} \twoheadrightarrow (H, L, R, \sigma, \mathfrak{Jmp}, \omega)$.

a. The footprint of L *is exactly* Dom H.

b. If $L(r_1) = (\mathfrak{n}_1, A_1)$, $L(r_2) = (\mathfrak{n}_2, A_2)$ *and* $r_1 \neq r_2$, *then* $A_1 \cap A_2 = \varnothing$.

Proof. By an easy rule induction on $\twoheadrightarrow$. □

This lemma tells us tell us that the intuitive model for deallocation in region is valid: Instead of explicitly restricting the heap after each region operation, it is enough to deallocate the addresses associated with the region when a release operation decreases the reference count to 0.

(The reason for the roundabout definition of deallocation is that originally I had plans of investigating a non-standard semantics for region operations where Proposition 3.23(b) is not true. When it later became clear that time was too short for that, it was also too late to change the semantics to work more straightforwardly).

Next, to relate L to R, we will prove

Proposition 3.24. *Assume that* $\mathcal{A} \twoheadrightarrow (H, L, R, \sigma, \mathfrak{Jmp}, \omega)$. *Then* $\mathrm{Img}\, R \subseteq \mathrm{Dom}\, L$.

This basically says that the reference counting of regions works: Each region name that is bound to a region variable will also be known to the region manager. The statement cannot be proved by direct induction, because it does not say anything about the dormant region references in the stack ω.

As a helper concept, let us define a notation for counting the number of references to a region name:

$$R \# r \quad = \quad \#\{\rho^c \mid R(\rho^c) = r\}$$

and extend it to stacks in the natural way:

$$\begin{aligned} (\mathfrak{Rtn}, R, \sigma) :: \omega \ \# \ r &= R \# r + \omega \# r \\ \bullet \# r &= 0 \end{aligned}$$

Note that we are only counting references from ρ^cs, not from ρ^us. That is why the latter are *uncounted* region variables.

Lemma 3.25. *Assume that* $\mathcal{A} \twoheadrightarrow (H, L, R, \sigma, \mathfrak{Jmp}, \omega)$. *Then for all* r*: If* $\#r = R \# r + \omega \# r > 0$ *then* $L(r) = (\#r, A)$. *Otherwise* $L(r)$ *is not defined.*

Proof. By an easy rule induction on $\twoheadrightarrow$. □

Lemma 3.25 proves Proposition 3.24 as regards counted region variables. For uncounted variables we combine it with

Lemma 3.26. *Say that a stack* ω ***supports*** *a region environment* R *if for all* $\rho^u \in \mathrm{Dom}\, R$ *it holds that* $\omega \# R(\rho^u) > 0$, *and if* ω *has the form* $(\mathfrak{Rtn}, R', \sigma) :: \omega'$, *then* ω' *supports* R'.

Now assume that $\mathcal{A} \twoheadrightarrow (H, L, R, \sigma, \mathfrak{Jmp}, \omega)$. *Then* ω *supports* R.

Proof. Again, by rule induction on $\twoheadrightarrow$. □

This completes the proof of Proposition 3.24: For any $r = R(\rho^u)$, we now know that

$$R \# r + \omega \# r \geq \omega \# r > 0$$

and therefore by Lemma 3.25, $L(r)$ is defined.

We also have, by Lemma 3.25 and Proposition 3.23(a):

Corollary 3.27. *If* $\mathcal{A} \twoheadrightarrow (H, L, \sigma, [\,], \mathfrak{Jmp}, \bullet)$, *then* $L = [\,]$ *and* $H = [\,]$.

Thus, if the annotated uniform mutator eventually ends at Stop, then before the last transition, it will have deallocated every region and every heap cell it ever allocated. We will not make direct use of this property, but it is nice to know that our agents clean up after themselves nicely.

In an implementation this would mean that we can safely launch a new region-based application afterwards without bothering to reinitialize the region manager. Such a property would be especially important in an embedded or sandboxed setting where several independent mutators share a pool of memory managed by a common region manager.

3.2.3 A scoping discipline for agents

Proposition 3.24 tells us that an annotated uniform mutator will never get stuck because $\mathsf{L}(r)$ is not defined in a cons, release, or alias operation. However, there still exists the possibility of getting stuck because $\mathsf{R}(\rho)$ is not defined. To guard against this, we need to define a static discipline in the style of Section 3.1.5. Like in Section 3.1.5, our discipline will not guarantee that the agent is *safe*. It does prevent the agent from shooting *itself* in the foot (and getting stuck *within* memory-management code), but it does nothing about the possibility of shooting the mutator in the foot by deallocating its data prematurely.

Definition 3.28. *Let $\mathcal{D}$ be a finite map from node names to sets of region variable names and $\mathcal{S}$ be a finite map from node names to sets of ordinary variables. Define by the following rules the relations $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta, \Sigma\}\ \mathfrak{Mop}$ and $\mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma\}\ \mathfrak{Jmp}$, where $\Sigma \subseteq \boxed{\chi}$ and $\Delta \subseteq \boxed{\rho}$:*

$$\frac{\mathcal{D}(s) = \Delta \qquad \mathcal{S}(s) = \Sigma \qquad \Delta' \subseteq \boxed{\rho^{\mu}}}{\mathcal{D}, \mathcal{S} \vdash \{\Delta \cup \Delta', \Sigma\}\ \mathsf{goto}\ s}\ \textit{ASGOTO}$$

$$\frac{\mathcal{D}, \mathcal{S} \vdash \{\Delta \uplus \{\rho^c\}, \Sigma\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma\}\ \mathsf{new}\ \rho^c; \mathfrak{Jmp}}\ \textit{ASNEW}$$

$$\frac{\rho \in \Delta \qquad \mathcal{D}, \mathcal{S} \vdash \{\Delta \uplus \{\rho^c\}, \Sigma\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma\}\ \mathsf{alias}\ \rho\ \mathsf{to}\ \rho^c; \mathfrak{Jmp}}\ \textit{ASALIAS}$$

$$\frac{\mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S} \vdash \{\Delta \uplus \{\rho^c\}, \Sigma\}\ \mathsf{release}\ \rho^c; \mathfrak{Jmp}}\ \textit{ASRELEASE}$$

$$\frac{\mathcal{D}, \mathcal{S} \vdash \{\Delta \uplus \{\rho_2^c\}, \Sigma\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S} \vdash \{\Delta \uplus \{\rho_1^c\}, \Sigma\}\ \mathsf{rename}\ \rho_1^c\ \mathsf{to}\ \rho_2^c; \mathfrak{Jmp}}\ \textit{ASRENAME}$$

$$\frac{\forall i : \forall \sigma \xrightarrow{\varphi} \sigma' \in \mathfrak{Ss}_i : \begin{cases} \mathrm{Dom}\,\sigma = \Sigma \\ \mathcal{D}, \mathcal{S} \vdash \{\mathrm{Dom}\,\sigma', \Delta\}\ \mathfrak{Jmp}_i \end{cases}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta, \Sigma\}\ \mathsf{misc}\ \mathfrak{Ss}_1; \mathfrak{Jmp}_1 \,\square\, \cdots \,\square\, \mathfrak{Ss}_k; \mathfrak{Jmp}_k}\ \textit{ASMISC}$$

$$\frac{\rho \in \Delta \qquad \mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma \uplus \{x\}\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta, \Sigma \uplus \{x_0, \ldots, x_k\}\}\ x\ :=\ \mathsf{cons}\ \langle x_0, \ldots, x_k\rangle\ \mathsf{at}\ \rho;\ \mathfrak{Jmp}}\ \textit{ASCONS}$$

$$\frac{\mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma \uplus \{x\}\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta, \Sigma \uplus \{x_0, \ldots, x_k\}\}\ x\ :=\ \mathsf{cons}\ \langle x_0, \ldots, x_k\rangle\ \mathsf{nowhere};\ \mathfrak{Jmp}}\ \textit{ASNOWHERE}$$

• **History of notation:** There is an "underground tradition" of using the letter Δ to stand for the set of live region variables. The earliest such use I know of is Christiansen and Velschow [1998], from which the practise propagated to Henglein et al. [2001]. Wang [2001] also, apparently independently, uses Δ for region assumptions.

The punctuation I use for the relations in Definition 3.28 is inspired by the typing judgment "$\Psi \vdash \{\Delta, \Gamma\}$ *expr* $\{\Delta, \Gamma\}$" in the HMN region type system, which itself mimicked the traditional notation "{P} *stmt* {Q}" for Hoare triples. In the UHL framework, the continuation of a $\mathfrak{Mop}$ is given by its embedded next-state jump rather than by its context, so the judgments do not need a postcondition. (See Niss [2002] for a thorough development of the connections between Hoare logic and the HMN region type system).

$$\frac{x \in \Sigma \qquad \mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma \uplus \{x'\}\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta, \Sigma\}\ x' := \mathcal{H}[x{+}{+}];\ \mathfrak{Jmp}}\ \textit{asRead}$$

$$\frac{x \in \Sigma \qquad \mathcal{D}, \mathcal{S} \vdash \{\Delta, \Sigma\}\ \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta, \Sigma \uplus \{x'\}\}\ \mathcal{H}[x{+}{+}] := x';\ \mathfrak{Jmp}}\ \textit{asWrite}$$

$$\frac{\begin{array}{c}\operatorname{Img}\mathfrak{Ra} = \Delta \qquad \operatorname{Img}\mathfrak{Ua} \subseteq \Delta' \qquad \operatorname{Img}\mathfrak{Ma} = \Sigma \\ \operatorname{Dom}\mathfrak{Ra} \uplus \operatorname{Dom}\mathfrak{Ua} = \mathcal{D}(s) \qquad \operatorname{Dom}\mathfrak{Ma} = \mathcal{S}(s) \\ \forall s' \in \mathcal{E}_{\|\mathcal{A}\|}(s) : \begin{cases} \mathcal{D}(s') \cap \boxed{\boxed{\rho^c}} = \operatorname{Dom}\mathfrak{Ra}' \\ \mathcal{S}(s') = \operatorname{Dom}\mathfrak{Ma}' \\ \mathcal{D}, \mathcal{S} \vdash \{\Delta'', \Sigma''\}\ \mathfrak{Jmp} \\ \quad \text{where } \Delta'' = \Delta' \uplus \operatorname{Img}\mathfrak{Ra}' \\ \qquad\quad \Sigma'' = \Sigma' \uplus \operatorname{Img}\mathfrak{Ma}' \\ \qquad\quad (\mathfrak{Ra}', \mathfrak{Ma}', \mathfrak{Jmp}) = \mathfrak{Rtn}(s') \end{cases}\end{array}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta \uplus \Delta', \Sigma \uplus \Sigma'\}\ \mathsf{call}\ s(\mathfrak{Ra}, \mathfrak{Ua}, \mathfrak{Ma});\ \mathsf{then}\ \mathfrak{Rtn}}\ \textit{asCall}$$

$$\frac{}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\Delta, \Sigma\}\ \mathsf{end}}\ \textit{asEnd}$$

The "*AS*" in the rule names stand for "Agent Scoping".

The only really interesting one of these rules is *asGoto*, which says that *uncounted* region variables may disappear silently from Δ. The point of this is that a procedure with multiple entries may have *different* sets of uncounted parameters for each entry, as long as a particular ρ^u is defined for each entry from which a region annotation that mentions it is reachable. (However, this possibility is not a central one in our theory. It will only be useful in quite specialized cases, and always as an optimization rather than an essential feature. There is just no reason *not* to allow it).

Definition 3.29. *The annotated mutator $\mathcal{A}$ is **well-formed** by $\mathcal{D}$ and $\mathcal{S}$ iff*

a. $\operatorname{Dom}\mathcal{D} = \operatorname{Dom}\mathcal{S} = \operatorname{Dom}\mathcal{A}$.

b. For each s *in* $\operatorname{Dom}\mathcal{A}$, $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\mathcal{D}(s), \mathcal{S}(s)\}\ \mathcal{A}(s)$.

c. $\mathcal{D}(\mathtt{s0}) = \mathcal{S}(\mathtt{s0}) = \varnothing$.

d. For each $s \in \mathcal{E}_{\|\mathcal{A}\|}(\mathtt{s0})$, $\mathcal{D}(s) = \mathcal{S}(s) = \varnothing$.

Notice that the system given here is a refinement of the one in Section 3.1.5:

Fact 3.30. *If $\mathcal{A}$ is well-formed by $\mathcal{D}$ and $\mathcal{S}$, then $\|\mathcal{A}\|$ is well-formed by $\mathcal{S}$.*

Theorem 3.31. *Let $\mathcal{M}$ be well-formed by $\mathcal{S}$. If $\mathcal{A} \twoheadrightarrow \mathcal{C}$ and $\mathcal{C}$ is stuck, then it has one of the following forms:*

1. Wrong.
2. $(H, L, R, \sigma, s, \omega)$ *where* $\mathcal{A}(s) = \mathsf{misc}\ \mathfrak{Gs}_1; \mathfrak{Jmp}_1 \,\square\, \cdots \,\square\, \mathfrak{Gs}_k; \mathfrak{Jmp}_k$ *such that no* $\mathfrak{Gs}_i$ *contains a step of the form* $\sigma \xrightarrow{\varphi} \sigma'$.
3. $(H, L, R, \sigma, s, \omega)$ *where* $\mathcal{A}(s) = x' := \mathcal{H}[x{+}{+}]$ *and* $\sigma(x) \notin \mathbb{A}$.
4. $(H, L, R, \sigma, s, \omega)$ *where* $\mathcal{A}(s) = \mathcal{H}[x{+}{+}] := x'$ *and* $\sigma(x) \notin \mathbb{A}$.

Notice that Theorem 3.31 corresponds exactly to Theorem 3.11. The proof proceeds in much the same way, except that the handling of region operations is, of course, new:

Definition 3.32. *Define the relations* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}$ *and* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp}$ *by*

$$\frac{}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathsf{Stop}}\ \textit{ASCSTOP} \qquad \frac{}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathsf{Wrong}}\ \textit{ASCWRONG}$$

$$\frac{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\mathrm{Dom}\,\mathsf{R}, \mathrm{Dom}\,\sigma\right\}\ \mathfrak{Jmp} \qquad \mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash (\mathsf{H}, \mathsf{L}, \mathsf{R}, \sigma, \mathfrak{Jmp}, \omega)}\ \textit{ASCSTD}$$

$$\frac{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Rop}; \mathfrak{Jmp}}\ \textit{ASSROP}$$

$$\frac{\forall s' \in \mathcal{E}_{\|\mathcal{A}\|}(s) : \mathcal{D}(s') = \mathcal{S}(s') = \varnothing}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \bullet \rhd \mathsf{goto}\, s}\ \textit{ASSEMPTY}$$

$$\frac{\forall s' \in \mathcal{E}_{\|\mathcal{A}\|}(s) : \begin{cases} \mathcal{D}(s') \cap \boxed{\rho^c} = \mathrm{Dom}\,\mathfrak{Ra} \\ \mathcal{S}(s') = \mathrm{Dom}\,\mathfrak{Ma} \\ \mathcal{D}, \mathcal{S} \vdash \left\{\Delta'', \Sigma''\right\}\ \mathfrak{Jmp} \\ \mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp} \\ \quad \text{where } \Delta'' = \mathrm{Dom}\,\mathsf{R} \uplus \mathrm{Img}\,\mathfrak{Ra}' \\ \qquad\quad \Sigma'' = \mathrm{Dom}\,\sigma \uplus \mathrm{Img}\,\mathfrak{Ma}' \\ \qquad\quad \left(\mathfrak{Ra}, \mathfrak{Ma}, \mathfrak{Jmp}\right) = \mathfrak{Rtn}(s') \end{cases}}{\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash (\mathfrak{Rtn}, \mathsf{R}, \sigma) :: \omega \rhd \mathsf{goto}\, s}\ \textit{ASSFRAME}$$

The "*ASC*" and "*ASS*" in the rule names stand for "Agent-Scoping invariant for Configurations/Stacks".

Lemma 3.33. *Let* $\Delta' \subseteq \boxed{\rho^u}$.
a. If $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\Delta, \Sigma\right\}\ \mathfrak{Jmp}$ *then* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\Delta \cup \Delta', \Sigma\right\}\ \mathfrak{Jmp}$.
b. If $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\Delta, \Sigma\right\}\ \mathfrak{Mop}$ *then* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\Delta \cup \Delta', \Sigma\right\}\ \mathfrak{Mop}$.

Proof. By rule induction on $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\Delta, \Sigma\right\}\ \mathfrak{Jmp}/\mathfrak{Mop}$. The cases for *ASGOTO* and *ASEND* are immediate.

For *ASRELEASE*, *ASMISC*, *ASCONS*, *ASNOWHERE*, *ASREAD* and *ASWRITE*, simply apply the induction hypothesis. Similarly for *ASNEW*, *ASALIAS*, and *ASRENAME*, because the freshly bound variable is uncounted and therefore cannot be in Δ'.

For *ASCALL*, observe that Δ' and $\mathrm{Img}\,\mathfrak{Ra}'$ are by disjoint by definition, so the induction hypothesis can be applied to each of the "$\mathcal{D}, \mathcal{S} \vdash \left\{\Delta'', \Sigma''\right\}\ \mathfrak{Jmp}$" premises in turn. □

Lemma 3.34. *Assume* $\mathcal{D}, \mathcal{S} \vdash \left\{\Delta, \Sigma\right\}\ \mathsf{goto}\, s$. *Then* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\Delta, \Sigma\right\}\ \mathcal{A}(s)$.

Proof. By *ASGOTO* we have that $\Sigma = \mathcal{S}(s)$ and $\Delta = \mathcal{D}(s) \cup \Delta'$ where Δ' consists of uncounted variables. Because $\mathcal{A}$ is well-formed, $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \left\{\mathcal{D}(s), \mathcal{S}(s)\right\}\ \mathcal{A}(s)$. Now apply Lemma 3.33(b). □

Lemma 3.35. *If* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathsf{goto}\, s$ *and* $\mathcal{A}(s)$ *contains* $\mathfrak{Jmp}$, *then* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp}$.

Proof. Let $\mathfrak{Jmp} = \mathfrak{Rop}_1; \cdots; \mathfrak{Rop}_n; \mathsf{goto}\, s'$. If $\mathcal{A}s$ contains $\mathfrak{Jmp}$, then $\|\mathcal{A}(s)\|_{\mathrm{Mop}} = \|\mathcal{A}\|(s)$ contains $\|\mathfrak{Jmp}\|_{\mathrm{Jmp}} = \mathsf{goto}\, s'$. Therefore $s \rhd_{\|\mathcal{A}\|} s'$, so $\mathcal{E}_{\|\mathcal{A}\|}(s') \subseteq \mathcal{E}_{\|\mathcal{A}\|}(s)$. $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathsf{goto}\, s$ implies $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathsf{goto}\, s'$, from which $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp}$ is easily derived. □

Proposition 3.36 (Subject reduction, level 1). *Assume* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}$.
a. If $\mathcal{C} \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}'$, *then* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}'$.
b. If $\mathcal{C} \xrightarrow[\mathcal{A}]{} \mathcal{C}'$, *then* $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}'$.

Proof. Part (a): By inspection of each of the rules for $\xrightarrow[\mathcal{A}]{\varphi}$, we see that $\mathcal{C}$ must have the form $(H, L, R, \sigma, \mathsf{goto}\, s, \omega)$. Then the derivation of $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}$ must be by *ASCSTD* and *ASGOTO*, so by Lemma 3.33 we get $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\mathrm{Dom}\, R, \mathrm{Dom}\, \sigma\}\ \mathcal{A}(s)$. Now proceed by case analysis on the derivation of $\mathcal{C} \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}'$.

MXMISC $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\mathrm{Dom}\, R, \mathrm{Dom}\, \sigma\}\ \mathcal{A}(s)$ must be by *ASMISC*, which gives us $\mathcal{D}, \mathcal{S} \vdash \{\mathrm{Dom}\, R, \mathrm{Dom}\, \sigma'\}\ \mathfrak{Jmp}'$ for the new σ' and $\mathfrak{Jmp}$. By Lemma 3.35 we also have $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp}'$, so by *ASCSTD* we can conclude $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}'$, as required.

MXCONS The updates of σ in *MXCONS* exactly match the updates of Σ in *ASCONS*, so we have $\mathcal{D}, \mathcal{S} \vdash \{\mathrm{Dom}\, R, \mathrm{Dom}\, \sigma'\}\ \mathfrak{Jmp}'$ for the new σ' and $\mathfrak{Jmp}$. Now apply Lemma 3.35 as for *MXMISC*.

MXNOWHERE, *MXREAD*, and *MXWRITE* Similar to *MXCONS*.

MXREADWRONG and *MXWRITEWRONG* Immediate by *ASCWRONG*.

MXCALL From the premises of *ASCALL* and the assumption that $\mathcal{A}$ is well-formed we get the first premise of *ASCSTD* for $\mathcal{C}'$. The second one follows by *ASSFRAME*; half of the premises come from the *ASCALL*, the other half from Lemma 3.35 applied to the second premise of the original *ASCSTD*.

MXRETURN The right premise of *ASCSTD* must be derived by *ASSFRAME*. We have $\mathcal{E}_{\|\mathcal{A}\|}(s) = \{s\}$, so *ASSFRAME* has exactly two premises which happen to be just what we need for *ASCSTD* for the new configuration.

MXSTOP Immediate by *ASCSTOP*.

Part (b). $\mathcal{C} \xrightarrow[\mathcal{A}]{} \mathcal{C}'$ must be derived by *MXROP*. By comparison of each of the rules for $\xrightarrow[\mathfrak{Rop}]{}$ with the corresponding *ASXXX* rule we find that $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\mathrm{Dom}\, R', \mathrm{Dom}\, \sigma\}\ \mathfrak{Jmp}$. We also have $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Rop}; \mathfrak{Jmp}$, hence $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathfrak{Jmp}$. Thus $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash (H|_{L'}, L', R, \sigma, \mathfrak{Jmp}, \omega)$, since the heap part of the configuration is ignored by the *ASCXXX* rules. □

Proof of Theorem 3.31. First, prove $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}$ by rule induction on $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\rightarrow \mathcal{C}$. The base case *MBSTART* is guaranteed by Definition 3.29(c,d). For *MBRGN* and *MBSTD*, use the induction hypothesis and Proposition 3.36, and for the other rules the induction hypothesis directly.

Now do case analysis on the derivation of $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}$. The case *ASCSTOP* is immediate (by definition Stop is not stuck), as is *ASCWRONG*, which is explicitly allowed by the statement of the theorem. So we have $\mathcal{C} = (H, L, R, \sigma, \mathfrak{Jmp}, \omega)$ and $\mathcal{D}, \mathcal{S} \vdash \{\mathrm{Dom}\, R, \mathrm{Dom}\, \sigma\}\ \mathfrak{Jmp}$. Continue the case analysis on the latter judgment.

ASNEW $\mathrm{Dom}\, R$ does not contain ρ^c, so *MXNEW* and *MXROP* applies and $\mathcal{C}$ was not stuck after all.

ASALIAS $\mathrm{Dom}\, R$ contains ρ but not ρ^c. Also, by Proposition 3.24, $L(R(\rho))$ is defined, so *MXALIAS* and *MXROP* applies and $\mathcal{C}$ was not stuck after all.

asRelease $\mathrm{Dom}\,\mathsf{R}$ contains ρ^c, and Proposition 3.24 tells us that $\mathsf{L}(\mathsf{R}(\rho))$ is defined. Therefore either *mxReleaseN* or *mxRelease1* applies with *mxRop*, and $\mathcal{C}$ was not stuck after all.

asRename $\mathrm{Dom}\,\mathsf{R}$ contains ρ_1^c but not ρ_2^c. Therefore *mxRename* and *mxRop* applies and $\mathcal{C}$ was not stuck after all.

asGoto $\mathfrak{Jmp}$ is $\mathtt{goto}\,\mathsf{s}$ for some s, and by the assumption that $\mathcal{A}$ is well-formed, plus Lemma 3.33, we have $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \{\mathrm{Dom}\,\mathsf{R}, \mathrm{Dom}\,\sigma\}\ \mathcal{A}(\mathsf{s})$. Continue the case analysis by the derivation of that:

asMisc Either *mxMisc* will apply or we are in the situation described by prong 2 in the theorem.

asCons We have $\rho \in \mathrm{Dom}\,\mathsf{R}$ and χ_0, ..., $\chi_k \in \mathrm{Dom}\,\sigma$. By Proposition 3.24 this implies that $\mathsf{L}(\mathsf{R}(\rho))$ is defined. Therefore, *mxCons* applies and $\mathcal{C}$ was not stuck after all.

asNowhere Similar to *asCons*.

asRead We know that $\sigma(\chi)$ is defined. According to its value, either prong 3 in the theorem applies, or one of *mxRead* or *asReadWrong* will apply.

asWrite Similar to *asRead*.

asCall The first three premises of *asCall* guarantee that *mxCall* will apply, and $\mathcal{C}$ was not stuck after all.

asEnd Continue the case analysis on the derivation of $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \omega \rhd \mathtt{goto}\,\mathsf{s}$. If *assEmpty*, then *mxStop* will apply. Conversely, if *assFrame*, then its premises guarantees the *mxReturn* will apply. In either case $\mathcal{C}$ was not stuck after all. □

3.3 Region soundness

In Section 3.2.2, we defined an agent to be "region safe" if it makes sure that none of the mutator's heap accesses will be outside the currently allocated part of the heap. It is intuitively obvious that this is what we want to require of agents that our region inference produces. Or is it?

Certainly, region safety is what most published proofs of "correctness" of the Tofte–Talpin calculus [Banerjee et al. 1999; Calcagno et al. 2002; Calcagno 2001; Dal Zilio and Gordon 2000; Helsen and Thiemann 2000] or other region systems [Aiken et al. 1995; Christiansen and Velschow 1998; Niss 2002; Walker et al. 2000; Wang 2001] aim to establish. But on closer inspection, it is really not all we need. When our goal is region *inference*, we are interested in region-based memory management as an implementation technique for programming languages with implicit memory management (whereas some of the cited works view regions as a programmer-visible concept that just happens to be statically checked). And in general, we expect an implementation to do more than simply not crash: We expect it to implement the semantics of the original program says it should do. In the present context, it means that the region-annotated mutator must have the same observable behavior as the original (non-annotated) mutator had. We define:

Definition 3.37. *The agent $\mathcal{A}$ is called* ***region sound*** *iff $Abs([\![\mathcal{A}]\!]) = Abs([\![\mathcal{M}]\!])$, where $\mathcal{M} = \|\mathcal{A}\|$.*

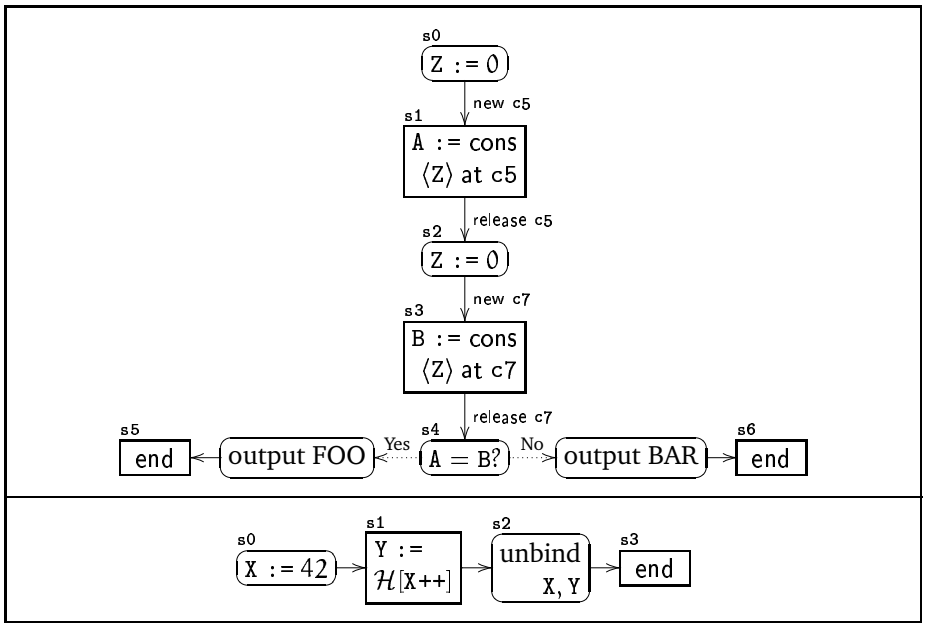


Figure 3.6: Examples of the difference between safety and soundness. The annotated mutator in the top part of the figure is region safe but not region sound. The one in the bottom part is sound but not safe.

Soundness is (with some variations in terminology) what the original articles by Tofte and Talpin [1994, 1997] proved. In the subsequent literature, Calcagno [2001] seems to be alone in mentioning soundness, and that only in a brief remark to the effect that the safety that he proves obviously implies soundness. And this implication is indeed more or less obvious in Calcagno's context, but *it is no longer true in our more general setting!*

Indeed, consider the annotated mutator shown on Figure 3.6(a). It is trivially region *safe* (not containing any read or write operations, it cannot possibly end up in state Wrong), but it is not region sound: If we ignore the region annotations, we find that the global abstraction of its ideal behavior contains only $(\text{out.BAR}, \Downarrow)$ (and its prefixes), because the test at `s4` will always come out false. (In the ideal semantics, all allocations in a single execution trace will be at different addresses). However, in the *managed* execution the cell pointed to by `A` gets deallocated before the second allocation, so the region manager is allowed to reuse the same address for both cons nodes. Therefore, $(\ominus, \ominus, \ominus, \ominus, \text{out.FOO}, \ominus, \Downarrow)$ is a managed behavior, so $(\text{out.FOO}, \Downarrow)$ is in the global abstraction – which proves the lack of soundness.

Incidentally, the converse implication is false, too: Soundness does not imply safety. The uniform mutator on Figure 3.6(b) is an example of this. It consistently tries to read from an unknown heap address, and has as its only execution trace

$$([\,],[\,],\mathtt{s0},\bullet) \xrightarrow[\mathcal{M}]{\ominus} ([\,],\left[\begin{smallmatrix}\mathtt{X}\\42\end{smallmatrix}\right],\mathtt{s1},\bullet) \xrightarrow[\mathcal{M}]{\ominus} \mathsf{Wrong}$$

Nothing an agent can do will prevent this, so any agent will be region unsafe. On the other hand, this also mean that an arbitrary agent will actually preserve

the ideal behavior; thus the agent is sound.

3.3.1 One half of soundness is trivial

We shall adopt region soundness as our general standard of correctness for agents. However, our definition of soundness is quite inconvenient to reason about. In general it is easier to prove safety, so we want to develop a criterion allows us to infer soundness from safety and excludes cases like Figure 3.6(a). We will do that below, after we prove a general theorem that simplifies reasoning about region soundness:

Theorem 3.38. *Let $\mathcal{A}$ be any well-formed annotated uniform mutator, and let $\mathcal{M} = \|\mathcal{A}\|$. If $[\![\mathcal{A}]\!] \subseteq [\![\mathcal{M}]\!]$, then $\mathcal{A}$ is region sound.*

Proof. Below we will show (Proposition 3.43) that $[\![\mathcal{M}]\!] \subseteq [\![\mathcal{A}]\!]$ holds in general. Therefore $[\![\mathcal{M}]\!] = [\![\mathcal{A}]\!]$, so their global abstractions are obviously equal, which proves that $\mathcal{A}$ is region sound. □

We will prove that every ideal behavior is also a managed behavior directly by simulation. The idea of the proof is to take an arbitrary ideal execution trace and show that the annotated mutator can duplicate it exactly, writing and reading the same values in the same heap cells. This is possible because the managed semantics merely *allows* but does not *force* the reuse of heap cells. The only way this can fail is if the agent has already deallocated a cell when the ideal execution accesses it – but in that case the managed execution will get stuck, whereupon *MBSTUCK1* allows it to keep mimicking the I/O events that the the ideal execution does, nevertheless.

Define, for the duration of the proof, the relation $\sim$ between managed configurations and ideal configurations (with the helper relation $\sim'$ between managed and ideal stacks):

$$\frac{}{\bullet \sim' \bullet}\ \textit{SIMEMPTY} \qquad \frac{\omega \sim' \omega'}{(\mathfrak{Rtn}, \mathsf{R}, \sigma) :: \omega \sim' (\|\mathfrak{Rtn}\|_{\mathrm{Rtn}}, \sigma) :: \omega'}\ \textit{SIMFRAME}$$

$$\frac{\mathsf{H} \subseteq \mathsf{H}' \qquad \|\mathfrak{Jmp}\|_{\mathrm{Jmp}} = \mathsf{goto}\, \mathsf{s} \qquad \omega \sim' \omega'}{(\mathsf{H}, \mathsf{L}, \mathsf{R}, \sigma, \mathfrak{Jmp}, \omega) \sim (\mathsf{H}', \sigma, \mathsf{s}, \omega')}\ \textit{SIMSTD}$$

$$\frac{}{\mathsf{Stop} \sim \mathsf{Stop}}\ \textit{SIMSTOP} \qquad \frac{\mathcal{C} \text{ is stuck}}{\mathcal{C} \sim \mathcal{C}'}\ \textit{SIMSTUCK}$$

Lemma 3.39. *Assume $\mathcal{A} \xrightarrow{\vec{\varphi}\!\!\rightarrow} \mathcal{C} = (\mathsf{H}, \mathsf{L}, \mathsf{R}, \sigma, \mathfrak{Jmp}, \omega)$ and that $\|\mathfrak{Jmp}\|_{\mathrm{Jmp}} = \mathsf{goto}\, \mathsf{s}$. Then there exists $\mathsf{H}' \subseteq \mathsf{H}$, L', R' such that $\mathcal{A} \xrightarrow{\vec{\varphi}\!\!\rightarrow} (\mathsf{H}', \mathsf{L}', \mathsf{R}', \sigma, \mathsf{goto}\, \mathsf{s}, \omega)$.*

Proof. By structural induction on $\mathfrak{Jmp}$. The case $\mathfrak{Jmp} = \mathsf{goto}\, \mathsf{s}$ is trivial. If, on the other hand, $\mathfrak{Jmp} = \mathfrak{Rop}; \mathfrak{Jmp}'$, then by Theorem 3.31 $\mathcal{C}$ cannot be stuck, so $\mathcal{C} \xrightarrow[\mathcal{A}]{} \mathcal{C}'(\mathsf{H}|_{\mathsf{L}'}, \mathsf{L}', \mathsf{R}', \sigma, \mathfrak{Jmp}', \omega)$. Therefore $\mathcal{A} \xrightarrow{\Psi\!\!\rightarrow} \mathcal{C}'$. Now apply the induction hypothesis to $\mathcal{C}'$, and we find $\mathcal{A} \xrightarrow{\Psi\!\!\rightarrow} (\mathsf{H}'', \mathsf{L}'', \mathsf{R}'', \sigma, \mathsf{goto}\, \mathsf{s}, \omega)$, where $\mathsf{H}'' \subseteq \mathsf{H}|_{\mathsf{L}'} \subseteq \mathsf{H}$, which completes the proof. □

Lemma 3.40. *Assume $\mathcal{A} \twoheadrightarrow \mathcal{C}$, $\mathcal{C} \sim \mathcal{C}'$, and $\mathcal{C}' \xrightarrow[\mathcal{M}]{\varphi} \mathcal{C}'_*$. If $\mathcal{C} = (\mathsf{H}, \mathsf{L}, \mathsf{R}, \sigma, \mathsf{goto}\, \mathsf{s}, \omega)$ is not stuck, then $\mathcal{C} \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}_*$ for some $\mathcal{C}_*$ with $\mathcal{C}_* \sim \mathcal{C}'_*$.*

Proof. $\mathcal{C} \sim \mathcal{C}'$ must be derived by *SIMSTD*; we have

$$\mathcal{C}' = (\mathsf{H}', \sigma, \mathsf{s}, \omega) \quad \text{with } \mathsf{H} \subseteq \mathsf{H}' \text{ and } \omega \sim' \omega'$$

Now proceed by case analysis on the derivation of $\mathcal{C}' \xrightarrow[\mathcal{M}]{\varphi} \mathcal{C}'_*$.

IXMISC We get $\mathcal{C}'_* = (\mathsf{H}, \sigma', \mathsf{s}', \omega')$ with a σ' such that by *SXMISC* we have $\mathcal{C} \xrightarrow[\mathcal{A}]{\varphi} (\mathsf{H}, \mathsf{L}, \mathsf{R}, \sigma', \mathfrak{Jmp}, \omega)$ with $\|\mathfrak{Jmp}\|_{\mathrm{Jmp}} = \mathsf{goto}\, \mathsf{s}'$. Thus $\mathcal{C}_* \sim \mathcal{C}'_*$, as required.

IXCONS Assume first that the at annotation in $\mathcal{A}(\mathsf{s})$ is $\mathsf{at}\ \rho$ rather than $\mathsf{nowhere}$. Since by assumption $\mathcal{C}$ is not stuck, we know that $\mathsf{R}(\rho)$ and $\mathsf{L}(\mathsf{R}(\rho))$ are both defined. Because the $\mathfrak{a}$ chosen in *IXCONS* is not in $\mathrm{Dom}\,\mathsf{H}'$, it is not in $\mathrm{Dom}\,\mathsf{H}$ either, so the same $\mathfrak{a}$ can be used in *MXCONS*

On the other hand, if the at annotation is $\mathsf{nowhere}$, then the $\mathfrak{a}$ from *IXCONS* can trivially be reused in *MXNOWHERE* (which does not restrict $\mathfrak{a}$ at all). H will not grow as H' does, but that is OK because H_* is only supposed to be a subset of H'_* anyway.

IXREAD Either *MXREAD* or *MXREADWRONG* will apply. In the latter case *SIMSTUCK* completes the proof, so assume the former. Because $\mathsf{H} \subseteq \mathsf{H}'$, we have $\mathsf{H}'(\mathfrak{a}) = \mathsf{H}(\mathfrak{a})$, so the σ' in *IXREAD* and *MXREAD* will be the same.

IXREADWRONG Because $\sigma(\mathsf{x}) \notin \mathrm{Dom}\,\mathsf{H}'$, it cannot be in $\mathrm{Dom}\,\mathsf{H}$ either. So *MXREADWRONG* applies, and *SIMSTUCK* completes the proof.

IXWRITE As for *IXREAD*, assume that *MXWRITE* will apply. Because *IXWRITE* and *MXWRITE* both update the heap with $\left[\begin{smallmatrix}\mathfrak{a}\\ \sigma(\mathsf{x}')\end{smallmatrix}\right]$, the subset relation between them will be preserved.

IXWRITEWRONG Similar to *IXREADWRONG*.

IXCALL Because $\mathcal{C}$ is not stuck, *MXCALL* will apply; only one $\mathcal{C}_*$ is possible. It is obvious that $\mathcal{C}_* \sim \mathcal{C}'_*$ by *SIMSTD* and *SIMFRAME*.

IXRETURN Because $\mathcal{C}$ is not stuck and $\sim'$ respects the overall shape of stacks, *MXRETURN* will apply.

IXSTOP $\omega \sim' \omega'$ implies $\omega = \bullet$, so the only rule that can apply is *MXSTOP*. As $\mathcal{C}$ is assumed non-stuck it *will* apply. □

Lemma 3.41. *Assume $\mathcal{M} \twoheadrightarrow \mathcal{C}'$ and $\mathcal{C} \sim \mathcal{C}'$. If $\mathcal{C}'$ is stuck, then so is $\mathcal{C}$.*

Proof. By case analysis on $\sim$. *SIMSTOP* is impossible: Stop is by definition not stuck. The case for *SIMSTUCK* is trivial. For *SIMSTD*, apply Theorem 3.11. Its first possibility is inconsistent with the shape of $\mathcal{C}'$ in *SIMSTUCK*; the other three immediately imply that $\mathcal{C}$ is also stuck, as required. □

Lemma 3.42. *Assume $\mathcal{M} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}'$. Then there exists $\mathcal{C}_0$ such that $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}_0$ and $\mathcal{C}_0 \sim \mathcal{C}'$.*

Proof. By rule induction.

IBSTART Trivial, by *MBSTART*.

IBSTD Apply the induction hypothesis to $\mathcal{M} \xrightarrow{\vec{\varphi}}\!\!\!\!\twoheadrightarrow \mathcal{C}$; we get a $\mathcal{C}_0$ such that $\mathcal{A} \xrightarrow{\vec{\varphi}}\!\!\!\!\twoheadrightarrow \mathcal{C}$. If $\mathcal{C}_0$ is stuck, then use *MBSTUCK1*. Otherwise $\mathcal{C}_0$ must have the form $(\mathsf{H}, \mathsf{L}, \mathsf{R}, \sigma, \mathfrak{Jmp}, \omega)$. By Lemma 3.39 we can assume that $\mathfrak{Jmp}$ has the form $\mathsf{goto}\, \mathsf{s}$; so apply Lemma 3.40 and *MBSTD*.

IBTERM Apply the induction hypothesis to $\mathcal{M} \xrightarrow{\vec{\varphi}\!\!\rightarrow} \mathsf{Stop}$; we get $\mathcal{C}_1$ such that $\mathcal{C}_1 \sim \mathsf{Stop}$. This means that $\mathcal{C}_1$ must either be Stop, in which case we can use *MBTERM*, or stuck, in which case we can use *MBSTUCK2*.

IBSTUCK1(2) Apply the induction hypothesis to the $\mathcal{M} \xrightarrow{\vec{\varphi}\!\!\rightarrow} \mathcal{C}'$. We get a $\mathcal{C}$ such that $\mathcal{C} \sim \mathcal{C}'$. By Lemma 3.41, $\mathcal{C}$ is stuck, so *MBSTUCK1(2)* will apply. □

Proposition 3.43. *Let $\mathcal{A}$ be a well-formed annotated uniform mutator, and let ψ be a managed behavior of it. Then ψ is also an ideal behavior of $\|\mathcal{A}\|$.*

Proof. This follows directly from Lemma 3.42. □

3.3.2 Soundness from safety: Pointer-blind programs

We now turn our attention back to the problem of when soundness is implied by safety. As explained, the challenge is to find a condition which will exclude programs like the one in in Figure 3.6(a) which is safe but not sound.

What goes "wrong" in Figure 3.6(a) is that the program compares pointers after what they point to has been deallocated. So if we disallow pointer comparisons altogether, we should at least be on our way to having safety imply soundness. (Later, we will allow a controlled form of pointer comparison). This plan also has the appealing feature that we can infer soundness from safety plus a property of the underlying uniform mutator rather than of the agent.

How can we formally specify the absence of pointer comparisons in a uniform mutator when such comparisons can be hidden within a misc node? Intuitively, we can test it by taking each $\mathfrak{Gs}$ set in the uniform mutator and check that it is closed under the operation of "replacing some pointers by other pointers" on both sides of each $\sigma \xrightarrow{\varphi} \sigma'$ step. If that is the case, everything that the mutator can do with equal pointers, it can also do with different pointers, and vice versa.

The bit about "replacing some pointers by other pointers" in the previous paragraph was rather fuzzy, so we hasten to give a formal definition:

Definition 3.44. *Let $\mathcal{M}$ be a uniform mutator, and let the **displacement** δ be some map $\mathbb{W} \to \mathbb{W}$. A set $\mathfrak{Gs}$ of steps **commutes with** δ if, for all σ_1 and σ_2 with $\sigma_1 = \delta \circ \sigma_2$:*

a. If $\sigma_1 \xrightarrow{\varphi} \sigma_1' \in \mathfrak{Gs}$ then $\sigma_2 \xrightarrow{\varphi} \sigma_2' \in \mathfrak{Gs}$ for some σ_2' such that $\sigma_1' = \delta \circ \sigma_2'$.

b. If $\sigma_2 \xrightarrow{\varphi} \sigma_2' \in \mathfrak{Gs}$ then $\sigma_1 \xrightarrow{\varphi} \sigma_1' \in \mathfrak{Gs}$ for some σ_1' such that $\sigma_1' = \delta \circ \sigma_2'$.

$$\begin{array}{ccc} \sigma_1 & \xrightarrow{\varphi} & \sigma_1' \quad \in \mathfrak{Gs} \\ {\scriptstyle \delta\circ\_}\uparrow & & \uparrow{\scriptstyle \delta\circ\_} \\ \sigma_2 & \xrightarrow{\varphi} & \sigma_2' \quad \in \mathfrak{Gs} \end{array}$$

Definition 3.45. *The uniform mutator $\mathcal{M}$ is **pointer blind** if every $\delta : \mathbb{A} \to \mathbb{A}$ has an extension $\hat{\delta} : \mathbb{W} \to \mathbb{W} \supseteq \delta$ such that $\hat{\delta}^{-1}(\mathbb{A}) \subseteq \mathbb{A}$ and each $\mathfrak{Gs}$ in $\mathcal{M}$ commutes with $\hat{\delta}$.*

This definition turns out to be the right one (or at least strong enough to prove what we want):

Theorem 3.46. *Let $\mathcal{A}$ be a well-formed annotated uniform mutator. If $\|\mathcal{A}\|$ is pointer blind and $\mathcal{A}$ is region safe, then $\mathcal{A}$ is region sound.*

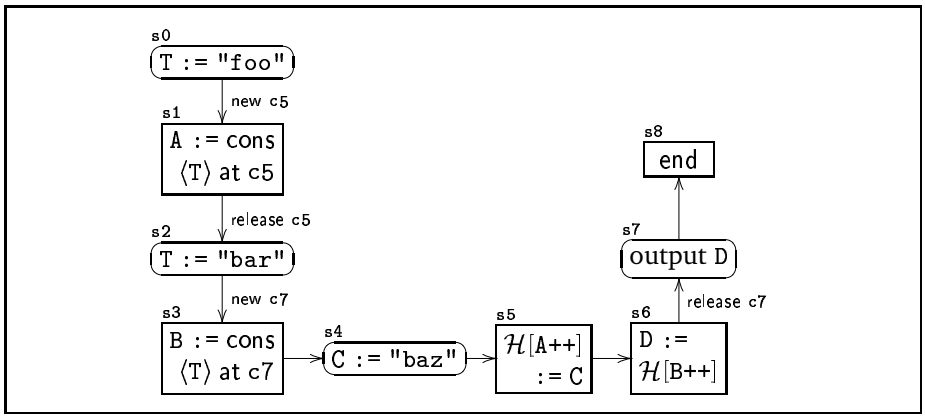


Figure 3.7: A counterexample to Proposition 3.47.

Some general remarks are in order before we prove this theorem. A uniform mutator can only be pointer blind if it represents arithmetic numbers by elements of $\mathbb{W}$ that are distinct from the addresses in $\mathbb{A}$ ($\subsetneq \mathbb{W}$): Definition 3.44 must hold for displacements with *arbitrary* effects on $\mathbb{A}$, which will fail if the misc node in question tries to do arithmetic on something in $\mathbb{A}$. This implies that a pointer-blind program must use a tagged representation to distinguish between pointers and data.

This may seem problematic, because one might like to use region-based memory management with a language where a strong type systems subsumes the need for run-time tags on pointers. However, notice that the uniform mutator is but a model of the actual execution. We can prove soundness of a tagless annotated program by showing[2] that it is observationally equivalent to a version with tags and then applying Theorem 3.46 to the latter one.

The reason why we do not simply fix $\hat{\delta}(w) = w$ for $w \notin \mathbb{A}$ is that we want the theorem to be useful for implementations that sometimes combine a pointer with some tag bits in a single word (thereby setting up a part of $\mathbb{W} \setminus \mathbb{A}$ as a "shadow copy" of $\mathbb{A}$), or for applications where the uniform mutator sometimes pack several atomic values into one UHL variable (such as the representation of the call stack in the second half of Section 6.1.6, or when representing a unification operation in Prolog, in which case the unification stack can conveniently be represented by a single UHL variable).

Now, to prove the theorem, a natural first attempt would be to derive it as a corollary of

Proposition 3.47 (FALSE!). *Let $\mathcal{A}$ be a well-formed uniform mutator such that $\mathcal{M} = \|\mathcal{A}\|$ is pointer blind. If $\mathcal{A} \xrightarrow{\vec{\varphi}\!\!\rightarrow} \mathcal{C}$ such that $\mathcal{C} \neq$ Wrong, then $\mathcal{M} \xrightarrow{\vec{\varphi}\!\!\rightarrow} \mathcal{C}'$ for some $\mathcal{C}'$ (with $\mathcal{C} \sim \mathcal{C}'$ in some way).*

Unfortunately, this proposition is not true. Figure 3.7 shows a counterexample. Because the two cons nodes may end up allocating the same heap cell, there is a risk that the heap write in s5 will not go Wrong but instead update the same cell that B points to, in which case s7 will output "baz" instead of the expected "bar". Therefore Proposition 3.47 is not true for this example.

[2]Such a proof could be done with standard techniques for type soundness. Here it helps that the "infinite word length" of our model means that there is always room for another tag bit.

However, Theorem 3.46 is not necessarily lost, because even though the uniform mutator in the counterexample is clearly pointer blind, the agent is not safe – it *can* go wrong at `s5`. The unexpected fact is that this possibility of going wrong seems to harm the soundness even of execution histories where it does not actually happen.

Back to the drawing board, then. Theorem 3.46 apparently cannot be proved by the standard procedure of attempting the "obvious" induction proof and then adding material to the invariant as it becomes clear that it is needed. Instead we need a trick, and here is one that works:

Take an arbitrary managed execution trace and use it to construct not one but two simulations:

a. An ideal execution trace that simulates the original one, but with all pointers displaced sufficiently far from each other that all cons nodes in the trace allocate different cells. At the end of the day, this is what we need to conclude Theorem 3.46.

b. A *managed* execution trace that has had its pointers displaced in the same way as the ideal trace from (a). The purpose of this one is to end up at Wrong in all situations where the ideal trace cannot keep simulating the original one. Because the agent is assumed to be region safe, this cannot happen, so by contradiction the simulation (a) *will* keep working.

Definition 3.48. *Let an $\mathcal{A}$ with a pointer-blind underlying mutator be given, and fix the displacement $\delta : \mathbb{A} \to \mathbb{A}$ as*

$$\delta(n) = n - \left\lfloor \sqrt{n} \right\rfloor^2$$

Let $\hat{\delta}$ be its (mutator-specific) extension to $\mathbb{W}$ given by Definition 3.45.

The displacement maps new "ideal" addresses to the original "managed" ones. The point of the exact definition is to make sure that each contiguous sequences of "managed" addresses is hit sufficiently often by δ that we can always find a suitable "ideal" address at a cons node.

Definition 3.49. *Define the relation $\mathcal{C}_1 \frown \mathcal{C}_2 \smile \mathcal{C}_3$, where $\mathcal{C}_1$, $\mathcal{C}_2$ are managed configurations and $\mathcal{C}_3$ is an ideal one, by*

- $\mathsf{Stop} \frown \mathsf{Stop} \smile \mathsf{Stop}$.
- $(H_1, L_1, R, \sigma_1, \mathfrak{Jmp}, \omega_1) \frown (H_2, L_2, R, \sigma_{23}, \mathfrak{Jmp}, \omega_2) \smile (H_3, \sigma_{23}, s, \omega_3)$ *iff*
  1. $H_1 \dagger \delta' = \hat{\delta} \circ H_2$, *where δ' is a finite, injective restriction of δ.*
  2. *For all $n \in \operatorname{Dom} H_2$ it holds that $\delta(n+1) = \delta(n) + 1$.*
  3. $H_2 \subseteq H_3$
  4. $L_1 = \big(\lambda(n, A).(n, \delta(A))\big) \circ L_2$.
  5. $\sigma_1 = \hat{\delta} \circ \sigma_{23}$.
  6. $\|\mathfrak{Jmp}\|_{\mathrm{Jmp}} = \mathsf{goto}\, s$.
  7. ω_1 *is* ω_2 *with every local state σ composed with $\hat{\delta}$.*
  8. ω_3 *is* ω_2 *with every* R *removed and every $\mathfrak{Rtn}$ replaced by $\|\mathfrak{Rtn}\|_{\mathrm{Rtn}}$.*

Lemma 3.50. *Assume $\mathcal{C}_1 \frown \mathcal{C}_2 \smile \mathcal{C}_3$. If $\mathcal{C}_1 \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}_1'$ then either $\mathcal{C}_2 \xrightarrow[\mathcal{A}]{\varphi}$ Wrong or there exist $\mathcal{C}_2'$ and $\mathcal{C}_3'$ such that $\mathcal{C}_2 \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}_2'$ and $\mathcal{C}_3 \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}_3'$ with $\mathcal{C}_1' \frown \mathcal{C}_2' \smile \mathcal{C}_3'$.*

Proof. By case analysis on the derivation of $\mathcal{C}_1 \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}_1'$.

MXMISC Because $\mathcal{A}$ is pointer blind, and by Definition 3.44(a), the chosen $\mathfrak{Gs}$ also contains $\sigma_{23} \xrightarrow{\varphi} \sigma_{23}'$ for some σ_{23}' such that $\sigma_1' = \hat{\delta} \circ \sigma_{23}'$. This allows us to construct the required $\mathcal{C}_2'$ and $\mathcal{C}_3'$ by *MXMISC* and *IXMISC*, respectively.

MXCONS The original execution allocates $k + 1$ words beginning at address a. Choose n such that $2n + 1 > a + k + 1$ and $n^2 > \max \operatorname{Dom} H_3$, and let $a_2 = n^2 + a$. Then, for $0 \le i \le k + 1$ it holds that $n^2 \le a_2 + i < (n + 1)^2$, so $\delta(a_2 + i) = a + i$ and for $a_2 \le a' \le a_2 + k$ we have $\delta(a' + 1) = \delta(a') + 1$.

Construct the appropriate instances of *MXCONS* and *IXCONS* for $\mathcal{C}_2$ and $\mathcal{C}_3$, where in each case the new heap cells are allocated at address a_2. Now $\mathcal{C}_1' \frown \mathcal{C}_2' \smile \mathcal{C}_3'$ for the new states: Clause 1 of the definition of $\frown\smile$ holds by extending δ' with values for $\{a_2, ..., a_2 + k\}$; it will stay injective because by assumption $\{a, ..., a + k\}$ is disjoint from $\operatorname{Dom} H_1$. Clause 2 continues holding by explicit construction of a_2. Clause 3 keeps holding because the same addresses $\{a_2, ..., a_2 + k\}$ get added to H_2 and H_3. Clause 4 keeps holding because the new addresses map to the old ones. Clause 5 keeps holding because x_0 through x_k are uniformly removed from both data states, and $\hat{\delta}(a_2) = a$ by construction for the new value of x. Clause 6 holds by definition of $\mathcal{M} = \|\mathcal{A}\|$, and clauses 7 and 8 are not touched.

MXNOWHERE Select a_2 as for *MXCONS*. Then apply *MXNOWHERE* to $\mathcal{C}_2$ and *MXCONS* to $\mathcal{C}_3$. Definition 3.49(3,5) obviously keep holding, ant the other clauses are unchanged.

MXREAD We know that $\hat{\delta}(\sigma_{23}(x)) = \sigma_1(x) \in \mathbb{A}$. Therefore, by definition of pointer blindness, $\sigma_{23}(x) \in \mathbb{A}$ too, so either *MXREAD* or *MXREADWRONG* will apply for $\mathcal{C}_2$. In the latter case we are done, so assume *MXREAD*.

Then $\sigma_{23}(x) \in \operatorname{Dom} H_2$, so we have $\hat{\delta}(H_2(\sigma_{23}(x))) = H_1(\sigma_1(x))$. Therefore, the required relation between $\sigma_1'(x')$ and $\sigma_2' 3(x')$ will hold after *MXREAD*. Also, because $\sigma_{23}(x) \in \operatorname{Dom} H$ we have

$$\hat{\delta}(\sigma_{23}'(x)) = \hat{\delta}(\sigma_{23}(x) + 1) = \hat{\delta}(\sigma_{23}(x)) + 1 = \sigma_1(x) + 1 = \sigma_1'(x),$$

so the value of x will also stay related between $\mathcal{C}_1'$ and $\mathcal{C}_2'$.

It is clear that $\mathcal{C}_3$ can execute by *IXREAD* in parallel with $\mathcal{C}_2$ because they share the same σ_{23} and $H_3 \supseteq H_2$.

MXREADWRONG As for *MXREAD*, we must have $\sigma_{23}(x) \in \mathbb{A}$, but if $\sigma_{23}(x) \in \operatorname{Dom} H_2$ then $\hat{\delta}(\sigma_{23}(x)) = \sigma_1(x) \in \operatorname{Dom} H_1$, a contradiction. Therefore $\mathcal{C}_2 \xrightarrow[\mathcal{A}]{\ominus}$ Wrong and we are done.

MXWRITE As for *MXREAD*, we find $\mathcal{C}_2 \xrightarrow[\mathcal{A}]{\ominus}$ Wrong or $\sigma_{23}(x) \in \operatorname{Dom} H_2$. Updating $H_2(\sigma_{23}(x))$ with $\sigma_{23}(x')$ in parallel with updating $H_1(\sigma_1(x))$ with $\sigma_1(x') = \hat{\delta}(\sigma_{23}(x))$ will not break the synchronization, and the new value of x will also match according to the same computation as for *MXREAD*.

Likewise, $\mathcal{C}_3$ can execute by *IXWRITE* in parallel to $\mathcal{C}_2$ by *MXWRITE*.

MXCALL, *MXRETURN* All three traces execute calls and returns in synchrony. There is a lot of uninteresting plumbing to take care of, but in the end the fact that the call does not get stuck in $\mathcal{C}_1$ ensures that it will not do in one of the other states either, and the $\frown\smile$ relation is preserved by design.

MXSTOP Similarly. □

Lemma 3.51. *Assume $\mathcal{C}_1 \frown \mathcal{C}_2 \smile \mathcal{C}_3$ with $\mathcal{A} \twoheadrightarrow \mathcal{C}_1$ and $\mathcal{A} \twoheadrightarrow \mathcal{C}_2$. If $\mathcal{C}_1 \xrightarrow[\mathcal{A}]{} \mathcal{C}_1'$ then there exists $\mathcal{C}_2'$ such that $\mathcal{C}_2 \xrightarrow[\mathcal{A}]{} \mathcal{C}_2'$ with $\mathcal{C}_1' \frown \mathcal{C}_2' \smile \mathcal{C}_3$.*

Proof. $\mathcal{C}_1 \xrightarrow[\mathcal{A}]{} \mathcal{C}_1'$ can only be derived by *MXROP*. The premise $L_1, R \xrightarrow[\mathfrak{Rop}]{} L_1', R$ easily implies to $L_2, R \xrightarrow[\mathfrak{Rop}]{} L_2', R$ because none of the $\xrightarrow[\mathfrak{Rop}]{}$ rules care about the address sets in L.

Thus $\mathcal{C}_2'$ can be constructed by its own application of *MXROP*. Proposition 3.23 ensures that the heap restrictions for $\mathcal{C}_1'$ and $\mathcal{C}_2'$ proceed in parallel. □

Proposition 3.52. *Assume that $\mathcal{A}$ is pointer blind (by δ, $\hat{\delta}$) and region safe. If $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}_1$, then there exist $\mathcal{C}_2$ and $\mathcal{C}_3$ such that $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}_2$ and $\mathcal{M} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}_3$ with $\mathcal{C}_1 \frown \mathcal{C}_2 \smile \mathcal{C}_3$.*

Proof. By rule induction on $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}_1$.

MBSTART Easy:

$$([],[],[],[],\mathsf{goto}\,s0,\bullet) \frown ([],[],[],[],\mathsf{goto}\,s0,\bullet) \smile ([],[],s0,\bullet)$$

MBRGN Use the induction hypothesis and Lemma 3.51.

MBSTD Use the induction hypothesis and then Lemma 3.50. The outcome $\mathcal{C}_2 \xrightarrow[\mathcal{A}]{\varphi}$ Wrong is impossible because $\mathcal{A}$ was assumed to be region safe.

MBSTOP Trivial.

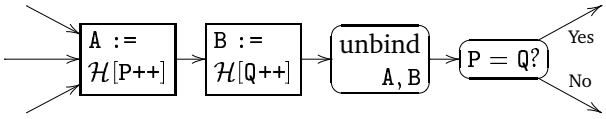
MBSTUCK1/2 If $\mathcal{C}_1$ is stuck, then one of the four cases in Theorem 3.31 applies. Case (1) contradicts the definition of $\frown\!\smile$. For case (2), apply Definition 3.44(b) to see that $\mathcal{C}_2$ and $\mathcal{C}_3$ must also be stuck. For case (3) and (4), $\hat{\delta}(\sigma_{23}(x)) = \sigma_1(x) \notin \mathbb{A}$ implies $\sigma_{23}(x) \notin \mathbb{A}$ by definition of $\hat{\delta}$. In every case $\mathcal{C}_2$ and $\mathcal{C}_3$ are both stuck, so set $\mathcal{C}_{2/3}' = \mathcal{C}_{2/3}$ and apply *MBSTUCK1/2* and *IBSTUCK1/2*. □

Proof of Theorem 3.46. The theorem now follows from Proposition 3.52 □

As it stands Theorem 3.46 is not completely helpful, because one can seldom be sure that uniform mutators will *always* be pointer blind. Pointer comparison is an often-used primitive in many object-oriented language, and even ML offers pointer comparison for references.

One way around this dilemma would be to instrument the uniform mutator such that each allocated block contains a unique (non-pointer) sequence number that can be compared instead of the pointers themselves. Of course this is not a *good* solution – it is somewhat impractical and inelegant and breaks the direct relationship between the heap object layout in our formal model and the intended low-level implementation of the language.

But the bad solution hints at a better one: If we can be sure that what the two pointers point to is still there, it will give the same results to compare the pointers than to read the sequence numbers and compare them. So in that case it will not hurt soundness to compare pointers. If we can arrange for *all* pointer comparisons to have this property, we never need to read sequence tags, so we don't have to write them in the first place. This could be done by giving each pointer comparison a context like



The two reads ensure that the program goes Wrong if the comparison would be unsound. Because the values read are discarded, no actual "load" instructions need to be used when the annotated program is eventually compiled to machine code – the job of the read nodes is only to force the region inference to avoid unsound agents, by pretending that they would be un*safe*.

This strategy can be integrated in the proof of Theorem 3.46: If $\mathcal{A}(s)$ is a comparison between variables x_1 and x_2, and we can prove by some method that $\mathcal{A} \twoheadrightarrow (H, L, R, \sigma, s, \omega)$ implies $\sigma(x_i) - 1 \in \operatorname{Dom} H$ then we can exempt s from Definition 3.44 and still have the theorem hold. The *MXMISC* case in Lemma 3.50 would need a special case for s: $\sigma_{23}(x_{1,2}) - 1$ will both be in $\operatorname{Dom} H_2$; because δ's restriction to $\operatorname{Dom} H_2$ is injective this means that

$$\sigma_{23}(x_1) - 1 = \sigma_{23}(x_2) - 1 \iff \delta(\sigma_{23}(x_1) - 1) = \delta(\sigma_{23}(x_2) - 1)$$

Because of Definition 3.49(2) we also have $\delta(\sigma_{23}(x_i) - 1) = \sigma_1(x_i) - 1$, so

$$\sigma_{23}(x_1) = \sigma_{23}(x_2) \iff \sigma_1(x_1) = \sigma_2(x_2)$$

which means that $\mathcal{C}_2$ and $\mathcal{C}_3$ will end up at the same side of the conditional as $\mathcal{C}_1$, as required.

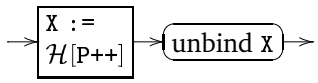
This proof tactic assumes that the edges between the dummy reads and the comparison do not contain any region operations that may deallocate the cells pointed to by P and Q. However, it is clear that such region operations would have no bearing on the values of the pointers themselves, so they cannot influence the outcome of the test.

The proof can also be extended to deal with the situation that we compare pointers not only for equality but also for the arithmetic order of different pointers (for example, to implement Prolog's pointer comparison operators @>, @<, etc.). We shall just sketch the basic principles: Definition 3.49(1) would need to be amended with a condition that δ's restriction to $\operatorname{Dom} H_2$ should be monotonically increasing. Then the proof case for the comparison operation would be easy, but maintaining this invariant in the *MXCONS* case is not trivial. If the original execution path starts by allocating something at addresses 23 and 25, there is no knowing in advance how many times it will allocate something at address 24 and expect it to lie between those two first allocations.[3] So we need to parameterize the induction property 3.52 with some extra information that ensures that enough unused space is left in the final H_3'. I think it is intuitively clear that this can always be done for any given entire original execution trace, but the details of formalizing it get messy no matter how it is done, so let us leave the matter at this sketchy level.

Another language feature that could prevent uniform mutators from being completely pointer blind is pointer arithmetic. The pointer-blindness restriction

[3] "Expect" in the sense that we need to prove that whatever the program does when the all of the subsequent allocations lie between the first two ones, it will be something that is allowed by its ideal semantics.

does allow a limited form of pointer arithmetic, namely the auto-increment behavior of the read and write operations. The combination



has the net effect of stepping the pointer in P one address forward[4], and a sequence of such stepping operations will suffice for modeling access to a non-initial field in a heap structure.

However, one can imagine situations where it would be desirable to perform some pointer arithmetic *without* immediately accessing the heap through the resulting pointer – for example, if one wanted to represent a

```
p1 = &(p2->fieldname);
```

statement in C. Such an offsetting operation done by dummy reads would go Wrong if p2 is a dangling pointer, which – though intuitively unnecessary – would force agents to ensure that the pointed-to structure is still allocated. It is doubtful whether this anomaly would do much harm to the precision of region inference (after all, programs rarely do something like this unless they intend to actually access the structure later anyway), but it would be nice to have a theoretical justification for not enforcing this artificial requirement in the region inference.

Unfortunately the proof of Theorem 3.46 relies directly on the guarantee that a safe program can do pointer arithmetic only on still-allocated addresses. This assumption allows Definition 3.49(2) to restrict its attention to $n \in \mathrm{Dom}\, H_2$; in default of it δ could have no discontinuities at all and the proof would break down.

The proof *can* be fixed to allow Wrong-free pointer increments, but as with comparing pointers for ordering, the details get messy and are left out. Again, we must let the details of the simulation depend on the future of the computation, and simulate one execution trace at a time. Definition 3.49(2) would need to be replaced with a statement that every pointer known to $\mathcal{C}_2$ can be incremented sufficiently far without running into a discontinuity of δ. An adequate interpretation of "sufficiently far" could be the length of the rest of the computation (here it comes in handy that Proposition 3.52 needs to speak about finite execution traces only), but it would need some finesse to define "every pointer known to $\mathcal{C}_2$" robustly if the extension from δ to $\hat{\delta}$ is not trivial.

3.3.3 Preservation of soundness

Theorem 3.46 will be our preferred way of showing from scratch that an agent is region sound[5], but we can get by easier if we already know another region-sound agent:

[4]Backward-stepping is rarely useful without C-like explicit pointer arithmetic and arrays, and we do not handle arrays at all in this thesis. However, if one wanted to model an object layout where some fields have negative offsets with respect to the pointers that the program passes around, one could easily imagine adding a primitive backwards-stepping instruction to UHL.

[5]However, we would need something stronger if we wanted to extend region inference to languages where the unannotated uniform mutator is not necessarily memory safe, such as C. It follows from the proof of Theorem 3.38 (more precisely, from Lemma 3.42), that any agent for a memory-unsafe mutator must be region-unsafe. I leave further exploration of this problem for further work.

Theorem 3.53. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be agents for the same mutator $\mathcal{M}$. Say that $\mathcal{A}_2$ is **conservative** with respect to $\mathcal{A}_1$ iff $[\![\mathcal{A}_2]\!] \subseteq [\![\mathcal{A}_1]\!]$.*

If $\mathcal{A}_2$ is conservative with respect to $\mathcal{A}_1$ and $\mathcal{A}_1$ is region sound, then $\mathcal{A}_2$ is also region sound.

Proof. Proposition 3.43 tells us that $[\![\mathcal{M}]\!] \subseteq [\![\mathcal{A}_2]\!]$. Because $\mathcal{A}bs(\cdot)$ is a monotonic operator (which is clear from its definition), this implies $\mathcal{A}bs([\![\mathcal{M}]\!]) \subseteq \mathcal{A}bs([\![\mathcal{A}_2]\!])$. For the opposite inclusion, the premise $[\![\mathcal{A}_2]\!] \subseteq [\![\mathcal{A}_1]\!]$ similarly implies $\mathcal{A}bs([\![\mathcal{A}_2]\!]) \subseteq \mathcal{A}bs([\![\mathcal{A}_1]\!]) = \mathcal{A}bs([\![\mathcal{M}]\!])$ where the last equality is from the assumption that $\mathcal{A}_1$ is region sound. □

This theorem will be our principal way to show that the output of an agent transformation preserves region soundness. Note that soundness is preserved even if the agents are not region *safe*!

(It is an intuitive fact that Theorem 3.53 is also true with "region safe" instead of "region sound", but it is not completely trivial to prove this with our definitions, because $[\![\cdot]\!]$ hides the difference between Wrong and other kinds of stuckness. We're not going to need this alternative result, so we shall spare the time it would take to prove it).

3.4 Annotatable edges and flowchart chunks

Sections 3.1 and 3.2 have presented the uniform host and agent languages as generally as possible without making the proofs of their properties unnecessarily complex. In practise one usually wishes to place some restrictions on the shape of mutators and agents.

First, it is not always convenient that *every* edge in the flowchart can be annotated by region operations. Some edges may only be put there for the sake of modeling the mutator's operations within the UHL framework but do not actually correspond to places in the machine code that the host implementation intends to generate for the mutator. In other cases it may be that the translation of a host-language construct into UHL simply introduces more edges than the number of different places in the source host-language syntax where it would be convenient to display region annotations. And in general, it is desirable, for the sake of the efficiency of the region inference algorithms, to reduce the number of possible places where region annotations can be put.

Therefore, in general we will let the producer of a uniform mutator also produce a set of edges that are "suited" to be annotated with region operations. We will refer to this set as the set of **annotatable** edges for the uniform mutator.

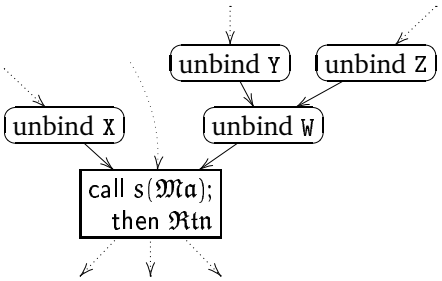
An alternative way of thinking of this is to look at the dual concept: Consider the flowchart of the uniform mutator with all of the annotatable edges removed. If there are sufficiently many annotatable edges (as there ought to be), the flowchart falls apart into (weakly) connected components. Let us call these components **chunks**. If two nodes in the flowchart are connected by an edge that is *not* annotatable, they are in the same chunk.

We can imagine a "simplified" flowchart consisting of the chunks as nodes plus the annotatable edges of the original uniform mutator. Most of the region inference process works on this simplified graph rather than the uniform mutator itself. The internal structure of the chunks will be mostly ignored – this is part of the reason why we could claim in Section 1.4.2 that uniform mutators

need not exist in full on the computer. Only the simplified flowchart actually needs to be constructed, and representations of the chunk internals need only be handled briefly during the construction of it.

3.4.1 Call chunks

Every chunk that contains a call node *must* have a shape like



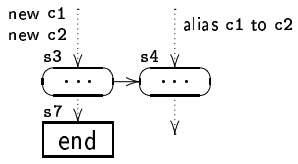
That is: Every edge *from* a call node *must* be annotatable. Every edge *to* a call node must be annotatable unless it comes from a misc node that just unbinds variables, in which case the same conditions apply to that misc node. (Neither a call node or the misc nodes that fall under this rule may be procedure entries).

Such chunks are called **call chunks** and are treated specially by most of the region inference algorithms. The presence of a fan of unbinding nodes in the chunk is somewhat inconvenient notationally; it would be nicer to be able to have a chunk consisting only of the call node. The reader is free to imagine that call chunks have this simpler form, if he is also willing to imagine that the definitions of how to map actual intermediate languages to UHL use slightly more complicated principles for positioning "unbind" nodes than the one I give in Chapter 4.

3.4.2 Guidelines for chunk size

Because most of the region-algorithms work on the chunk level, their running time of the region inference algorithms will to a large degree be affected by the number of chunks rather than the size of the mutator before dividing it into chunks. Therefore it is worthwhile to strive to have relatively large chunks. But if the chunks get too large, the precision of chunk-based transformations and analyses may suffer.

Imagine, for example, a region-annotated mutator containing



where dotted arrows stand for annotatable edges and `s3` and `s4` constitute a chunk. Now imagine a region optimization that depends on doing some sort of alias analysis on region variables. If the analysis treats each entire chunk as an atomic unit, it will conclude that it is possible for `c1` and `c2` to be aliased in

the chunk containing `s7`, even though `s7` is not actually reachable from the alias operation.

To prevent such problems, and to make sure that reasonable agents are possible, we will define a set of general rules for the shape of chunks. One of the requirements will reject the chunk above as "too big" – it should be split into two smaller chunks by making the edge from `s3` to `s4` annotatable, thereby breaking the apparent causal chain between `s4` and `s7`.

The rules are not absolutes – nothing will break if they are not followed, but in general better agents will be possible when they are observed.

Let a **split-node** mean a node in the flowchart with more than one outgoing edge, and a **join-node** be one with more than one incoming edge. A call node counts as "proxy" join-node (because there may be several calls of the same procedure) as well as a split node (because a return in the called procedure can go to either of several call nodes).

Now, there should be at least one annotatable edge on any path

a. from a read (or write) node to a cons node. The read may be the last use of a heap cell, and we want the agent to have the possibility to deallocate it and let it be reused in the cons. The same is true of a write node – the last access to a heap cell could easily be a write (for example if each read from the cell is consistently followed by a write, which can be a useful programming invariant for some uses of updateable locations).

b. from a read (or write) node to a join-node. The read may be the last access to a region that need exists only in one of the paths to the join-node; it should be possible to release the region before the join, lest other paths need to invent a binding for the region variable simply to keep the agent well-formed.

c. from a read (or write) node to an end node. It may be sound to deallocate some region after the read, and it would be wasteful to force it to be passed back to the caller and deallocated only after the return.

d. from a split-node to a cons node. The fact that the mutator chooses a particular direction at the split-node may be what the agent needs to conclude at some heap cells will never be accessed again; it should have the chance to let them be reused in the cons.

e. from a split-node to a join-node. Otherwise the agent will be unable to exploit the knowledge that the mutator takes this particular path rather than another one.

f. from a split-node to a end node. Again, the fact that the a particular direction is chosen may enable the agent to deallocate some region; it should be possible for this to happen within the procedure.

g. from a procedure entry to a cons node. It would be wasteful to force all callers to pass a "scratchpad" region as a parameter if the allocation turns out to be short-lived.

h. from a procedure entry to a join node. Some set-up region operations may be necessary for compatibility with the other predecessors of the join; it should be possible to do them locally rather than distribute across them all call sites.

We can summarize all these rules simply: There should be an annotatable edge on any path

$$\textbf{from} \left\{\begin{array}{c}\text{a read or write node}\\ \text{a split-node}\\ \text{a call node}\\ \text{a procedure entry}\end{array}\right\} \textbf{to} \left\{\begin{array}{c}\text{a cons node}\\ \text{a join-node}\\ \text{a call node}\\ \text{an end node}\end{array}\right\}$$

The only of these combinations that has not been argued for above is a path from a procedure entry to an end node without any intervening splits, joins, reads, writes, conses and calls. Such a procedure consists of just a linear sequence of misc nodes and could be ignored completely by the agent. The most efficient way of handling it would be to inline it[6] at all call sites. In default of that, it will not hurt much to award an annotatable edge somewhere in it.

The reasoning behind the "from procedure entry to something" cases also apply to the degenerate case that the procedure entry *is* a something. We will generally assume that the uniform mutator is constructed such that no procedure entry is a cons, call, end or join-node. This can be easily achieved by adding a no-op misc node and an annotatable edge, if necessary.

One further goal of this kind may be imagined: to have an annotatable edge on any path from one cons to another. If the allocated cells turns out never to be used, it should be possible to deallocate them and reuse the memory for the next cons. However, such a rule would be formally awkward and would also prevent the efficiency benefit that could be reaped from treating a complex sequence of term constructions as a unit in most of the region inference (which is a sound principle in general). This is why we added the nowhere possibility for the region annotation on a cons node. With nowhere there will never be any reason to put region operations between cons nodes (at least not because they are cons).

3.4.3 Pruning the annotatable edges

It will often be the case that the "natural" way of selecting annotatable edges during the generation of a uniform mutator will lead to many more annotatable edges than necessary for satisfying the rules of the previous subsections. Because many phases of the region inference work at the chunk level, having too many annotatable edges will harm the performance of the region inference. Let us therefore present a general algorithm for pruning the set of annotatable edges down to a minimum.

The basic idea is to consider the annotatable edges one at a time, and only leaving an edge annotatable if removing it would cause a violation of one of the principles and rules in the previous subsections. In principle it is immaterial which order we consider the rules in, but it will be easier to spot violations if we consider the edges in topological order. For this, we need the flowchart to be acyclic; if it contains cycles we need to identify, in some other way, at least one annotatable edge in each cycle and declare it to be immune to pruning. The last annotatable edge before an arbitrary join-node in the cycle will usually be a

[6] In the UHL representation, that is. The implementation is still free to *implement* it as a procedure call in its final object code if it is too long to *actually* be inlined.

good choice. [7] (It will not harm if there are cycles entirely without annotatable edges).

Assuming now that the flowchart is acyclic, consider the annotatable edges in backwards topological order. (A corresponding algorithm that works in forwards order is also possible; I leave it for the reader to imagine how it would work). For each edge, divide into three cases:

1. The edge leads, directly or indirectly through unbinding misc nodes and non-annotatable edges, to a call node.
2. The edge leads, directly or indirectly through non-annotatable edges, to a join-node or a cons or end node.
3. None of the above.

The topological order means that the "indirectly through non-annotatable edges" part of the specifications can be determined quickly by caching the outcome for previously considered edges.

In case (3) it is always possible to make the edge unannotatable. In case (2) it is possible unless the edge's "from" node is, or is reachable through edges that were non-annotatable initially, a read or write node, a split-node, a call node, or a procedure entry. In case (1) the edge can only be made unannotatable if the "from" node is an unbinding misc node whose in-edges are themselves (yet) annotatable.

We make no claim that this algorithm is "optimal" in the sense that it always removes the largest possible number of annotatable edges. It does seem intuitively plausible that this is the case, but since the entire pruning operation is pragmatically motivated rather than essential for correctness, we will not bother to try to prove its (assumed) optimality formally.

The pruning algorithm can often be integrated into the initial generation of the flowchart.

3.5 Possible extensions of the UHL model

Here are a couple of loose ideas that I have not had time to develop in full:

3.5.1 Region closures as in the TT model

As described in Section 2.1.1, the Tofte–Talpin region model (and ones building on top of it) allows a closure to contain one or more region handles in addition to ordinary mutator data. This allows higher-order programs to be region-annotated in ways that the UHL model does not support.

Regions within closures might be added to the UHL model by equipping each cons operation with an extra region annotation in the form of a list of region variables whose handles will be stored in the heap cells following the ones visible to the mutator. These embedded region variables could be read again by a new kind of region operation that initializes reads a region handle from the cell a certain offset from the pointer in a given UHL variable.

[7] If the cycle contains no join-nodes at all, it will be dead code and can be eliminated from the flowchart completely.

However, such an extension is far from unproblematic – for example, it breaks the consistency of region reference counts, and a result such as Proposition 3.24 would not be provable without reference to a region type system (as it stands currently, it needs not even assume that the agent is well-formed).

Nevertheless it would be interesting to try to add a feature such as this (and associated region inference techniques) such that we could get a combined system that truly subsumes the TT system as well as HMN.

3.5.2 Concurrency

The UHL model seems to be excellently suited to be used in concurrent settings based on message-passing between threads (as in Erlang). All thread could share a single region-based heap. When a thread has created a region it will be the only thread that knows the region, so it can do allocations in it and create and release aliases to it without synchronizing with other threads – until an alias for the region gets passed to another thread along with a message that is allocated partly in the region. After that, synchronization would need to be used.

Each thread could have its own list of free cards and exchange cards with a master free-list (requiring global synchronization) only when it runs out of free cards or accumulates them beyond a certain threshold.

3.5.3 Finite regions

The ML Kit includes a special analysis called **multiplicity analysis** [Birkedal et al. 1996] which identifies regions where at most one cons operation is ever done. Such **finite** regions have space for them allocated directly on the call stack instead of from the list of free region cards.

It would not be difficult to adapt the multiplicity analysis to work in the UHL model, but it is not clear which kind optimized representation could be used for finite regions here. The ML Kit can allocate them on the call stack, because the **letregion** invariant means that the lifetime of a region will always be a "nice" interval of the lifetime of its creating function's stack frame. In the UHL model things are not always that simple, though a static analysis might be able to show that a finite region behaves well enough to be stack-allocated.

Chapter 4

Application to ML

In this chapter, I give a primary motivating motivating example of how to use the UHL framework; namely how to derive a region system similar to HMN (Section 2.4) for a Standard ML subset. We will go a little step further than HMN and handle exceptions and lexical closures too.

We will not derive the HMN *region type system* just yet; that will be done in Chapter 5 and Section 7.2. Also, for the time being, we will ignore ML's native type system and treat it more or less as a typeless language. (The native types will come into play once we start doing region inference, but they are not necessary to derive the region-based execution model, which is the purpose of this section).

As argued in Section 1.4, in a real application one would derive a region model for the intermediate language in an existing implementation. Doing so here would entail a lot of detailed exposition of the details and idiosyncrasies of a real-world intermediate language, which would distract from the points that are relevant to region inference. So instead, we will follow the simpler but slightly less realistic path of translating from source code directly to UHL.

Variables:	z	$\in$ (variables)
Operators:	op	$::= + \mid - \mid * \mid \texttt{div} \mid \texttt{mod} \mid \cdots$
Exception names:	Ex	$::= Ex_1 \mid Ex_2 \mid \cdots \mid Ex_\kappa$
Expressions:	e	$::= z$
		$\mid \mathsf{fix}\, z_0.\lambda z_1.e \mid e_1\, e_2$
		$\mid \mathsf{let}\, z = e_0\, \mathsf{in}\, e_1\, \mathsf{end}$
		$\mid n \mid e_1\, op\, e_2$
		$\mid \mathsf{if}\, e_0 <> 0\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$
		$\mid \mathsf{input} \mid \mathsf{output}\, e$
		$\mid \mathsf{nil} \mid e_1 :: e_2 \mid \mathsf{case}\, e_0\, \mathsf{of}\, \mathsf{nil} \Rightarrow e_1 \mid z_1 :: z_2 \Rightarrow e_2$
		$\mid \mathsf{raise}\, Ex(e)$
		$\mid e_0\, \mathsf{handle}\, Ex(z) \Rightarrow e_1$

Figure 4.1: Syntax of the ML-subset host language.

Figure 4.1 shows the syntax of a representative subset of Standard ML, chosen to exhibit the essential subset of our procedure without being too large. We will discuss ways to extend the procedure to the full language later.

The subset is almost the same as HMN's host language FUN with region annotations removed. The differences consist of the addition of exceptions, I/O and function definitions within expressions. Also, we ignore HMN's "boxed integers", which were there only to be used in examples.

We notate the ML-level variables as z to prevent confusion with UHL variables x.

The data objects supported by the subset are functions, integers and lists. For simplicity we have no booleans but instead a conditional primitive that tests integers for zeroness.

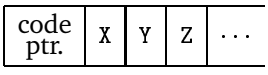
Unlike Standard ML, construction of recursive functions is an expression, not a declaration. (This is purely for convenience; it means that we won't have to define a separate syntactic class for declarations). "$\mathsf{fix}\, z_0.\lambda z_1.e$" corresponds to the Standard ML expression "$\mathsf{let\ fun}\ \ z_0\, z_1 = e\ \mathsf{in}\ z_0\ \mathsf{end}$".

The input and output e expressions are side-effecting I/O constructions; they input and output integers on some implicit communication channel. input evaluates to a freshly input number; output e evaluates e to an integer which is output and also becomes the value of the output expression. A program consists of a closed expression that is evaluated for its I/O side effects (its final value is discarded).

We assume that each program uses a finite number κ of exception names Ex; each exception takes a single parameter. Unlike in Standard ML, exceptions are not first-class values; one must be constructed explicitly at each raise expression and destroyed at the handle expression that catches it. Unhandled exceptions cause the program to terminate silently.

4.1 Indirect calls

The major problem with representing ML in the UHL model is what to do with indirect function calls where it is not statically known which function abstraction is actually called. We will assume that the intended implementation represent closures naively as heap objects laid out as



where X, Y, Z, ... are the values of the function's free variables. All the usual variations (such as indirect access links to closures for enclosing abstractions, displays, storing more variables than just the ones that actually appear, etc.) can be implemented with only minor changes to generated UHL, but let us stick to the simple model for now.

The problem, of course, is the code pointer in the first cell. Our imagined implementation translates application simply to an indirect call in the target machine's language, but UHL does not contain an equivalent of an indirect call. This omission is by design; the region-inference algorithms depend on the target of calls being explicitly given in the mutator's UHL implementation.

Our answer is that region inference must be preceded by some kind of *control-flow analysis* that approximates the set of (lexical) function abstractions that

may be the target of each application expression in the program. Control-flow analysis is a large topic in itself – see, for example, Mossin [1997] – but our requirements here will be modest. We will need the flow analysis to divide the function abstractions in the program into **procedure groups**, each with a distinguished label ℓ. Each application expression in the program will be tagged with one label ℓ such that the call will always end up going to one of the abstractions in the named procedure group:

$$\begin{aligned} \text{Expressions: } e ::= {} & \cdots \\ & \mid \ \mathsf{fix}\, z_0.\lambda^{\ell} z_1.e \\ & \mid \ e_1 \ @^{\ell} \ e_2 \end{aligned}$$

Functions would then have a (small-step) semantics like

$$\overline{\left(\mathsf{fix}\, z_0.\lambda^{\ell} z_1.e\right) \ @^{\ell} \ v \to e\left[z_0 \mapsto (\mathsf{fix}\, z_0.\lambda^{\ell} z_1.e), z_1 \mapsto v\right]}$$

such that $(\mathsf{fix}\, z_0.\lambda^{\ell} z_1.e)@^{\ell'} v$ with $\ell \neq \ell'$ is a run-time error (and the control-flow analysis being correct means that this kind of error will never occur).

Given the control-flow information, our strategy will be to construct the uniform mutator with one common entry node $\mathtt{s}^{\ell}$ for each procedure group. Each $@^{\ell}$ call is translated to a call of $\mathtt{s}^{\ell}$ with two parameters: One (say, `A`) for the actual function argument, and one (say, `C`) that is a pointer to the closure being applied. The entry node then reads the code-pointer part of the closure and jumps to the appropriate function body, as sketched on Figure 4.2.

The point of this translation is that it can be seen as a UHL representation of an ordinary indirect call: It can be used as the basis for region inference for an implementation where the caller reads the code pointer from the closure and makes an indirect call directly (!) through it. This will make sense for the agent as long as long as the edge marked with a star on the figure do not have any region annotations – all other edges belong unambiguously to a single one of the abstractions, so any region annotations on them can be compiled into a specific function body in the machine code. And the starred edge can easily be made unannotatable without violating any of the rules in Section 3.4.

We do not need a very complex control-flow analysis, although it is (or will be) clear that the precision of the region-inference improves if the control-flow analysis can find a finer partitioning of function abstractions into procedure groups. The HMN region type system of Section 2.4.3 implies a simple control-flow analysis – each region-annotated function type corresponds to a procedure group label. This analysis is equivalent to the simply-typed CFA that Mossin [1997, chapter 2] presents as "the simplest imaginable nontrival flow analysis", and proved to be perfectly adequate in our experiments. That does not necessarily tell much, because the host language in those experiments had a monomorphic type system and did not support higher-order programming very well, so the control flow of our benchmark programs was usually simple. I conjecture, however, that the simple CFA will continue being useful for all *monomorphic* programs.

Things are less clear when the source program contains higher-order functions that are used polymorphically. Mossin's formulation of the CFA generalizes easily to a polymorphic type system, but it is not clear that the resulting

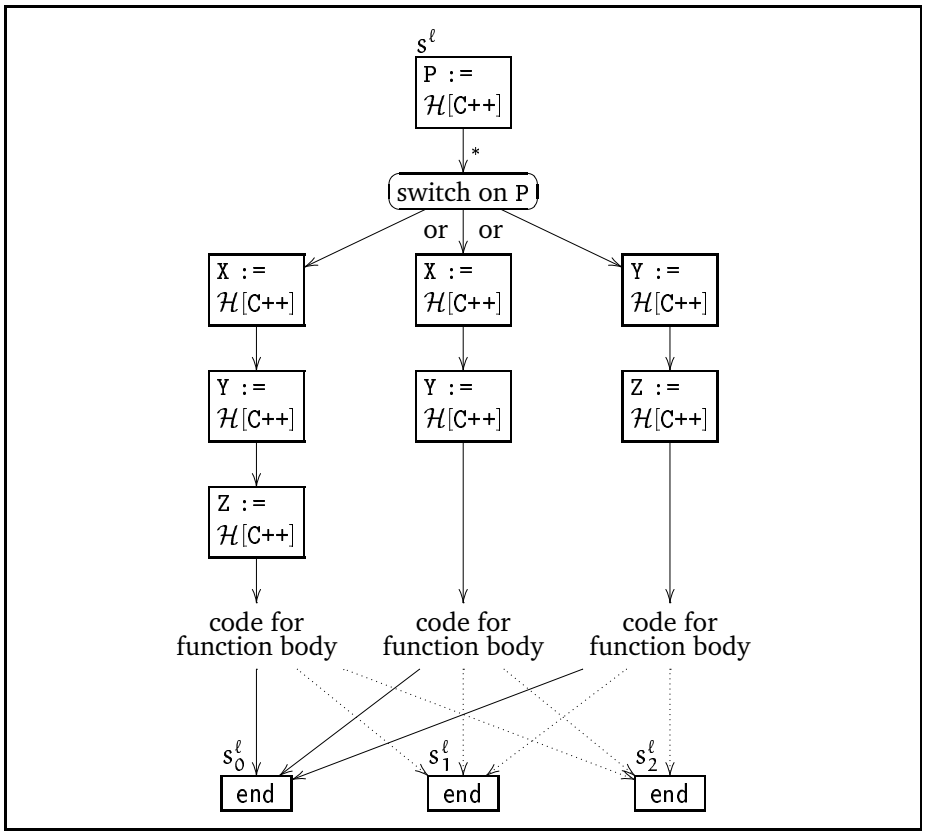


Figure 4.2: Sketch of a procedure group with three function definitions in it. There is also, coincidentally, three different returns. s_0^ℓ *is the normal return, and* s_1^ℓ *and* s_2^ℓ *correspond to exceptions being thrown. The dotted arrows hint that an exceptional return may not be reachable from all three abstractions. For a more detailed look at the code for function definitions, refer to Figure 4.8.*

uniform mutator will be able to support an efficient agent at all – it may be necessary to add novel features to our agent sublanguage before such programs can be handled well. In any case, my region inference techniques do not currently extend to polymorphically typed programs, so I will leave such extensions of the model for future work.[1]

An interesting question is whether it would be useful to employ a CFA that provides results with greater granularity, such that it can be expressed that the possible targets of one call is a proper subset of the possible targets of another. It is certainly possible to add secondary entry nodes (with their own, limited, switches on P) to the UHL flowchart for the procedure group, but it is not clear how good use the region-inference algorithms would be able to make of such extra precision, unless they were allowed to generate region annotations that

[1] One clear direction for such future work is to try to import features from the TT system, which does handle higher-order functions seamlessly (but at the cost of conceptual and algorithmic complications in the region-inference process, related to the "effect variables" that the TT region system uses to reason about higher-order functions); see Section 3.5.1 for further discussion.

were specific to the *combination* of a call site and a callee rather than just one or the other. And this would make the simple indirect-call implementation of function calls impossible.

One interpretation of the "one label per call expression" is that the label (as far as it corresponds to one entry node in the uniform mutator) is used to select the protocol used for communicating between the agent for the caller and the agent for the callee: How many regions references are passed in and out of the procedure, and which of the in-regions are uncounted? The more precise paradigm of "one (unique) label per abstraction and several labels per call expression" would correspond to a requirement that the caller's agent adheres to one of several protocols, which one being statically unknown.

4.2 The translation

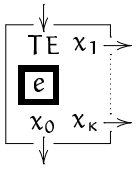
We are now in a position to describe a translation from our ML subset to UHL in some detail. This may seem like a waste of time – surely every reader who has but a glancing knowledge of elementary compiler construction will be able to imagine for himself approximately how such a translation would go. Nevertheless, the exercise will carry some rewards. First, the translation will serve as a practical example of how some of the finer points of UHL will be used in practise. Second, we get to survey in detail which kinds of misc nodes will be necessary to express ML programs as uniform mutators. This knowledge will be useful when we extend the translation with a region inference algorithm in the next chapter. Third, we get a chance to point out some features of the translation that are not necessarily direct models of what happens in a native intermediate language.

The bulk of the translation is a recursive translation of expressions to flowchart fragments. Each step in the translation takes the following arguments:

- e: The expression to translate.
- TE: The "translation environment", a finite map from $\boxed{z}$ to $\boxed{x}$. Its domain should consist of the variables that are lexically in scope at e.
- s_0: If e is evaluated without throwing an exception, it must jump to this control state.
- x_0: The value of the expression must be stored in this UHL variable at the jump to s_0.
- $s_1, ..., s_\kappa$: If the expression throws Ex_i, execution will continue at s_i.
- $x_1, ..., x_\kappa$: The counterparts of x_0 for exception throwing

The output of the translation consists of some new nodes in the flowchart, with a distinguished entry node s.

We will use a graphical notation for the translation of one expression:



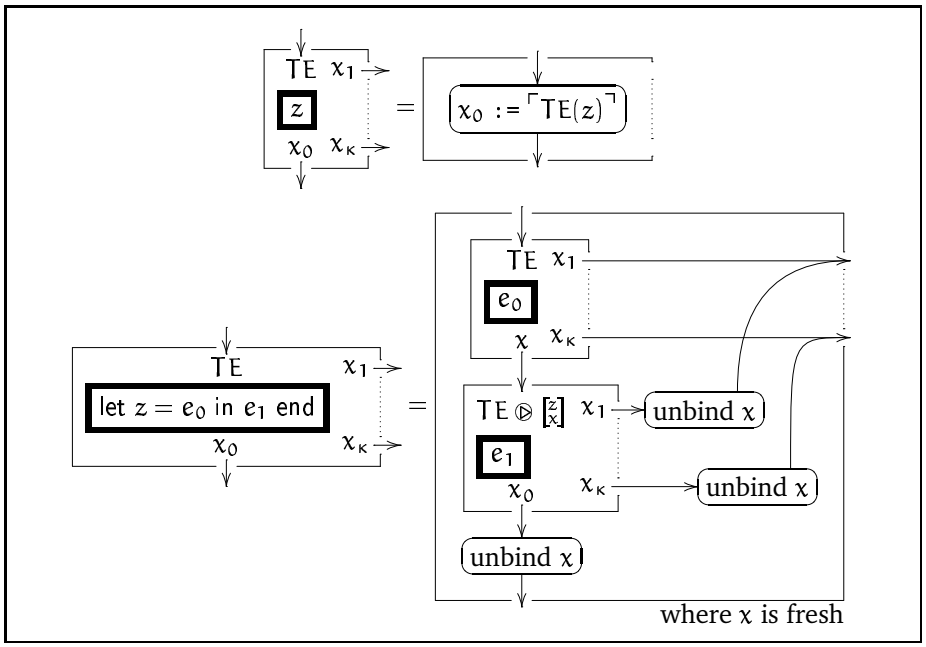
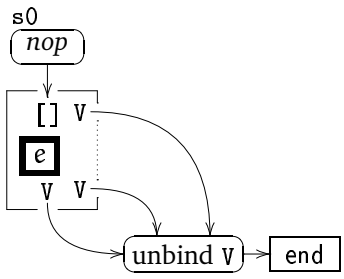


Figure 4.3: Translation for variables and let bindings.

The box symbolizes the flowchart fragment that corresponds to the expression. The arrows that pierce the box stand for the s_i inputs and the s output from the translation. Control enters at the top and leaves at the bottom, or to the right if the expression throws an exception.

An entire program is simply a closed expression e. Its translation to UHL just consists of giving the expression translation a suitable context, ignoring the value it evaluates to and prefixing it with a nop (so that the region inference will get a chance to insert region operations at the very beginning of the program):



4.2.1 Variables and let bindings

We can now begin to define the individual cases in the translation. Typical examples are the ones for variables and let bindings, shown on Figure 4.3. The case for variables is simple – a variable expression is evaluated by copying the UHL variable x that represents the ML variable into x_0 which represents the result of the expression. Variable expressions never result in exception throws,

shown by the fact that there is no arrows going to the dotted section in the right-hand edge of the box.

The case for let expressions show the notation for the translation of expressions with subexpressions. For each subexpression it is shown how the exceptional exits for Ex_1 and Ex_κ connect to the superexpression's exceptional exists; it is to be understood that these two shown connections represent a multitude of actual connections. However, a connection does not need actually to be generated if there is nothing at its "from" end, so if e_1 happens to be, say, a variable, the only $\boxed{\text{unbind } x}$ that needs to appear in the generated UHL program is the one for normal termination.

The body of the let expression is translated in an environment that has been extended with a fresh UHL variable to represent the ML variable. Because the rule uses $\circledcirc$ rather than $\oplus$, ML variable declarations naturally shadow each other in the standard way, with different UHL variables for each instance of the ML variable. (This is the primary reason why we do not simply take $\boxed{z}$ to be a subset of $\boxed{x}$).

It is an invariant of the translation that no flowchart fragment will unbind other UHL variables than its own temporaries. Therefore, the let-expression translation implies that the UHL variable that represents the ML variable will stay bound for as long as the ML variable is lexically in scope. This is slightly unrealistic – if the let expression appears in a tail-call context and contains a tail call, an actual compiler will probably discard the stack frame containing the variable as part of the tail call. However (and this is the point), this kind of deviation from implementation reality is harmless. It will clearly not disturb the region soundness of the eventual agent that the mutator keeps it data around for *shorter* periods in time than specified by the UHL program,[2] and it happens that our region inference process will not care about the precise times when local variables are unbound either. This point is worth making because there are often engineering reasons to want to do region inference at a point in the compilation pipeline where the lifetimes of local variables have not yet been determined.

4.2.2 Arithmetic, I/O, and conditionals

Next, let us consider integer arithmetic operations, whose translation are shown in Figure 4.4. These contain some nontrivial uses of misc nodes:

$$\boxed{x := n} \sim \mathsf{misc}\left\{\sigma \xrightarrow{\ominus} \sigma \oplus \left[{x \atop I(n)}\right] \mid \forall\sigma\right\}; \mathsf{goto}\ldots$$

$$\boxed{\begin{array}{l} x_0 := x \mathbin{\mathrm{op}} x' \\ \text{unbind } x, x' \end{array}} \sim \mathsf{misc}\left\{\sigma \oplus \left[{x \; x' \atop I(n)\; I(n')}\right] \xrightarrow{\ominus} \sigma \oplus \left[{x_0 \atop I(m)}\right] \mid n \mathbin{\mathrm{op}} n' = m\right\}; \mathsf{goto}\ldots$$

$$\boxed{\text{input } x_0} \sim \mathsf{misc}\left\{\sigma \xrightarrow{\text{in},n} \sigma \oplus \left[{x_0 \atop I(n)}\right] \mid \forall\sigma, n\right\}; \mathsf{goto}\ldots$$

$$\boxed{\text{output } x_0} \sim \mathsf{misc}\left\{\sigma \oplus \left[{x_0 \atop I(n)}\right] \xrightarrow{\text{out},n} \sigma \oplus \left[{x_0 \atop I(n)}\right] \mid \forall\sigma, n\right\}; \mathsf{goto}\ldots$$

where I is some fixed injective **representation function** $\mathbb{Z} \to \mathbb{W} \setminus \mathbb{A}$. It is important that Img I is disjoint from $\mathbb{A}$; otherwise nodes like $\boxed{x := n}$ could never be pointer blind.

[2] Actually, neither would it if data was kept around *longer* than the UHL abstraction says, as long as they are not used to access the heap.

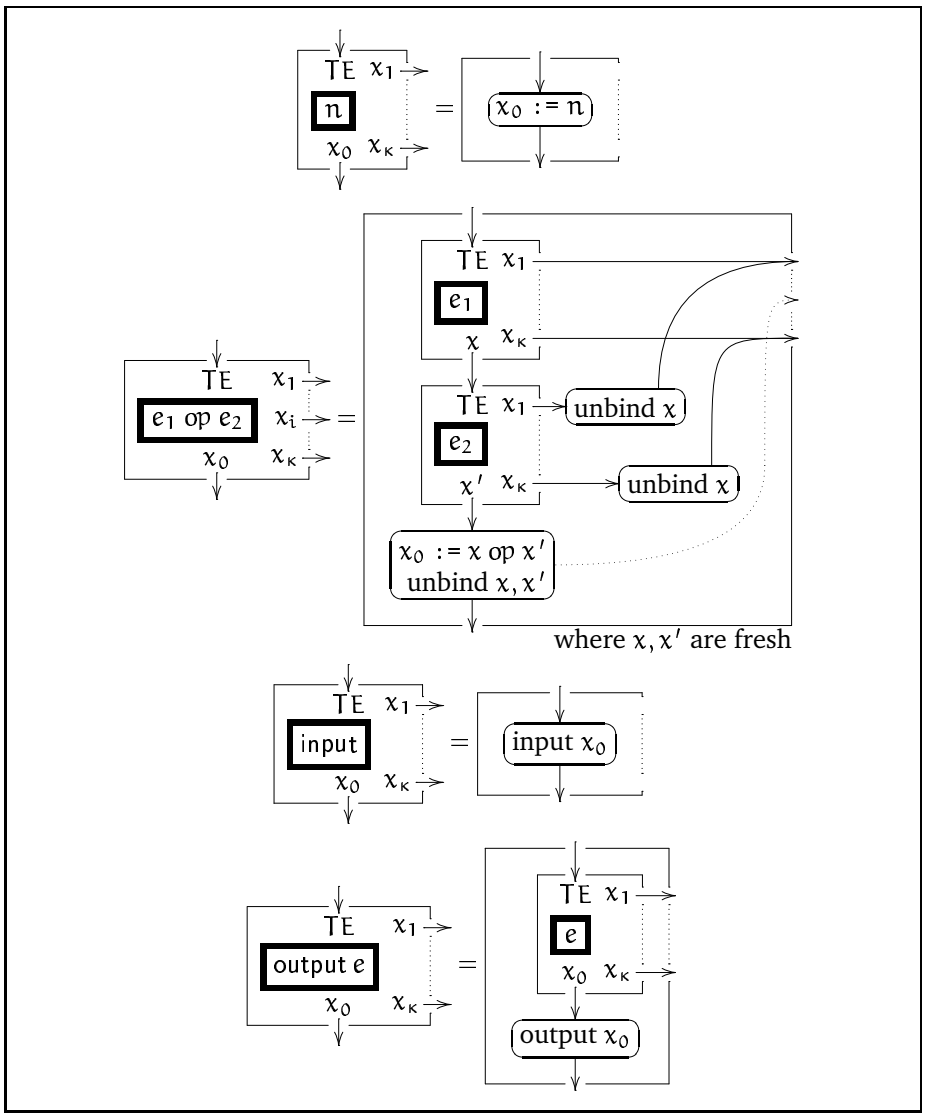


Figure 4.4: Translation for integer constants and arithmetic, and I/O.

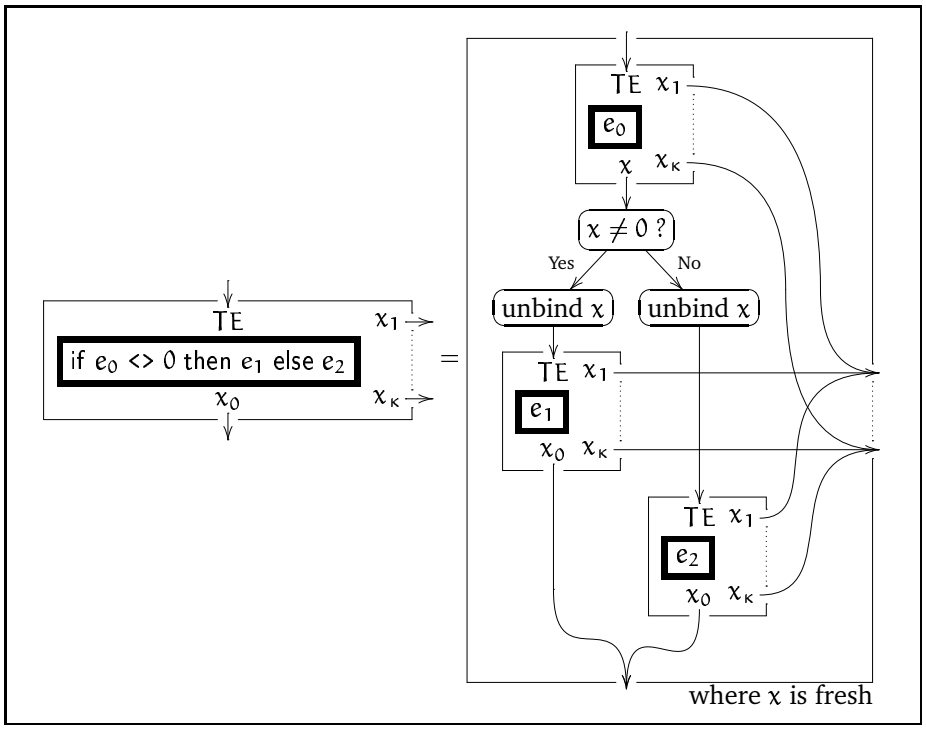


Figure 4.5: Translation for conditionals.

If one wants to model exceptions being thrown by errors such as division by zero, an explicit edge from the operation node to an appropriate exception Ex_i can be added straightforwardly, as suggested by the dotted line on the figure:

$$\boxed{\begin{array}{l} x_0 := x \operatorname{div} x' \\ \text{unbind } x, x' \end{array}} \sim \operatorname{misc} \left\{ \sigma \oplus \left[\begin{smallmatrix} x & x' \\ I(n) & I(n') \end{smallmatrix}\right] \xrightarrow{\ominus} \sigma \oplus \left[\begin{smallmatrix} x_0 \\ I(m) \end{smallmatrix}\right] \;\middle|\; m = \lfloor n/n' \rfloor \right\}; \operatorname{goto} s_0$$
$$\square \left\{ \sigma \oplus \left[\begin{smallmatrix} x & x' \\ I(n) & I(0) \end{smallmatrix}\right] \xrightarrow{\ominus} \sigma \oplus \left[\begin{smallmatrix} x_i \\ ? \end{smallmatrix}\right] \;\middle|\; \forall \sigma, n \right\}; \operatorname{goto} s_i$$

Ordinary conditionals, shown on Figure 4.5 are unsurprising; the test node $\boxed{x \neq 0\,?}$ stands for

$$\operatorname{misc} \left\{ \sigma \xrightarrow{\ominus} \sigma \;\middle|\; \sigma(x) = I(n) \text{ for some } n \neq 0 \right\}; \operatorname{goto} s_{\text{Yes}}$$
$$\square \left\{ \sigma \xrightarrow{\ominus} \sigma \;\middle|\; \sigma(x) = I(0) \right\}; \operatorname{goto} s_{\text{No}}$$

4.2.3 Lists

We are now ready to do something that involves the heap. The prototypical example is lists. Let us assume a representation where the empty list is represented by a special value nil, different from all pointers, and other lists are represented as pointers to blocks of two heap words containing the car and cdr part, respectively. List introduction constructs are now easy and shown on Figure 4.6.

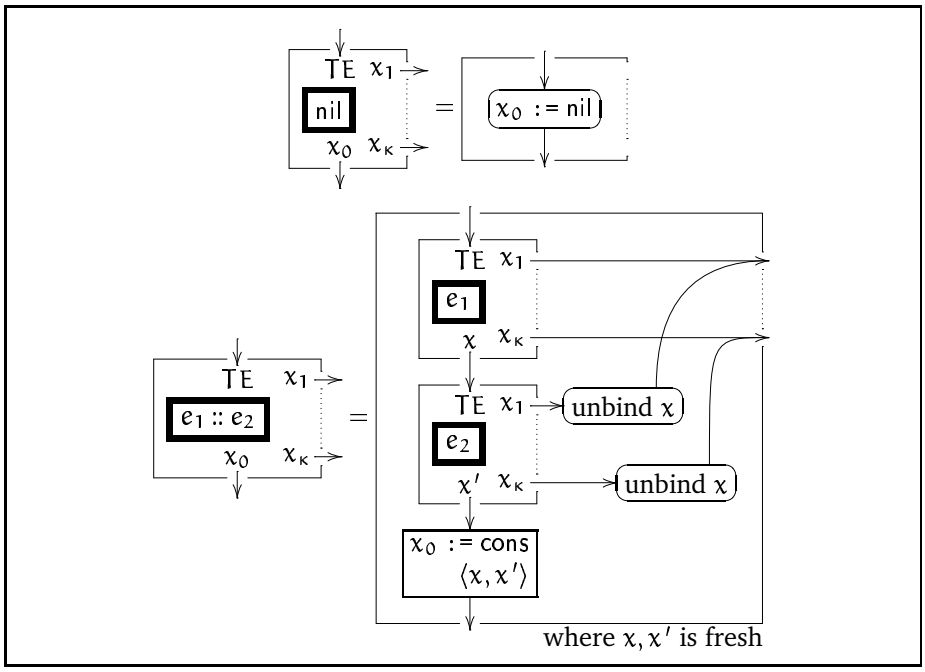


Figure 4.6: Translation for list construction

Elimination of lists, Figure 4.7, is slightly more complex. As expected, the translation looks like a cross between that for a conditional, and the branch for nonempty list contains read instructions that retrieve the list's components from the heap. The interesting detail, however, is the test node:

$$\boxed{x = \mathsf{nil}\,?} \sim \mathsf{misc}\,\{\sigma \xrightarrow{\ominus} \sigma \mid \sigma(x) = \mathsf{nil}\}; \mathsf{goto}\, s_{\mathrm{Yes}} \\ \Box\,\{\sigma \xrightarrow{\ominus} \sigma \mid \sigma(x) \neq \mathsf{nil}\}; \mathsf{goto}\, s_{\mathrm{No}}$$

Is that not a pointer comparison? Though nil is not itself a pointer, pointers are compared with it when the program runs. But no – this comparison treats all pointers equally, so it is allowed to use it in a pointer blind program.

4.2.4 Function abstractions and application

Function abstractions and application are translated according to the strategy presented in Section 4.1. The details are shown on Figure 4.8. We use the calling convention that the function-group entry s^{ℓ} receives a pointer to the closure in callee-variable C and the function's argument in callee-variable A. The ordinary end node delivers the return value in callee-variable V; exceptional ends use the same variable for the thrown exception's argument.

The translation of applications is simple; the interesting things all happen in the function definition. The code that goes into the flowchart for the function containing the definition just constructs a closure on the heap. The translation of the body itself connects to shared entry and exit code for the entire procedure group.

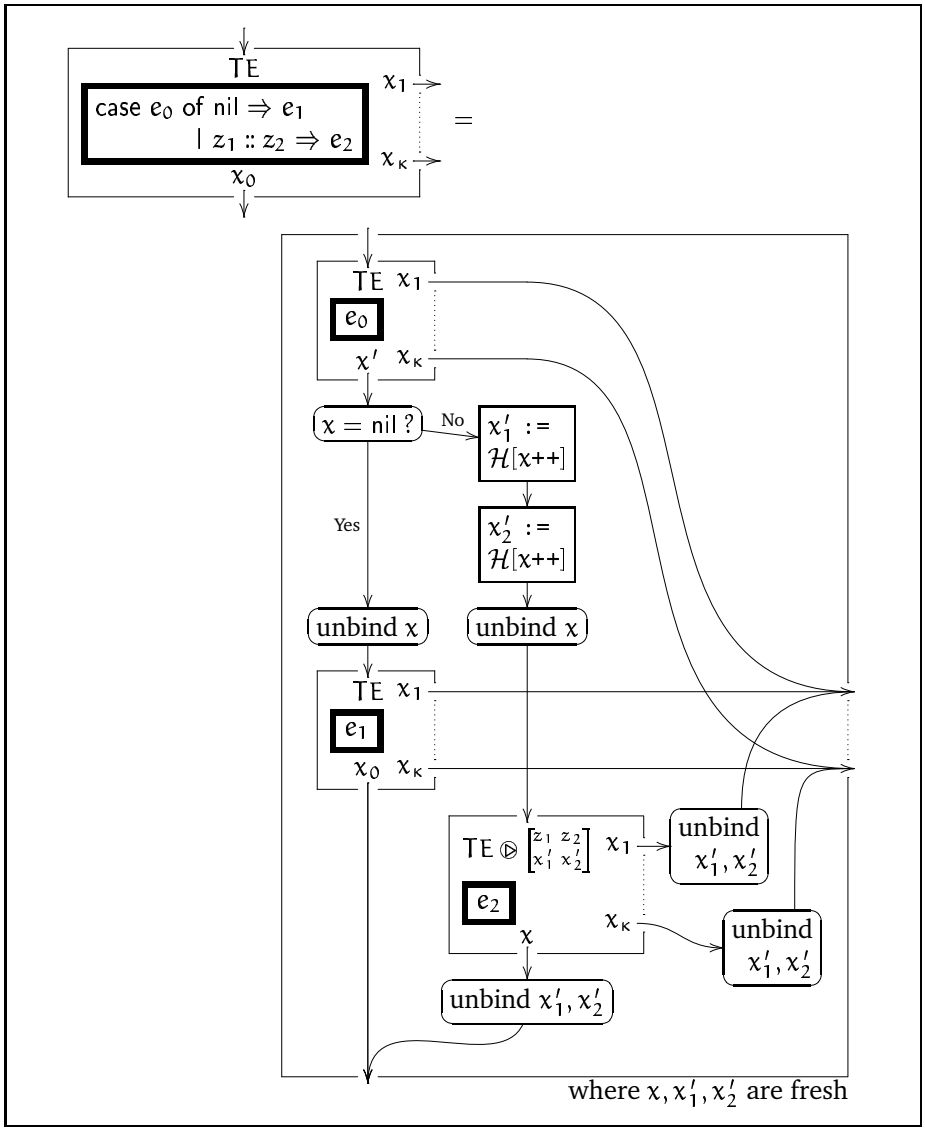


Figure 4.7: Translation for case analysis on lists.

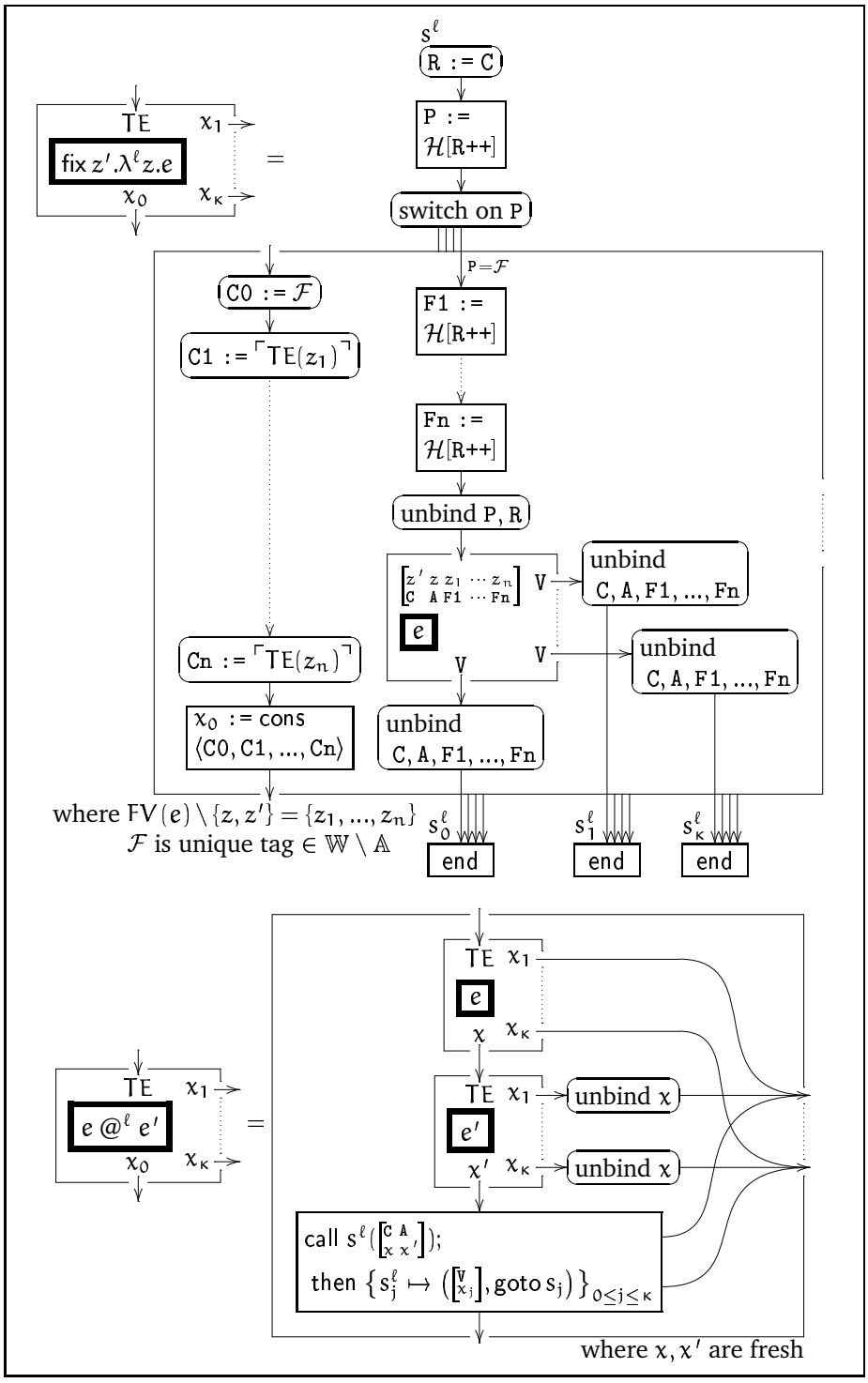


Figure 4.8: Translation of function abstraction and application. The nodes that are shown outside the translation box for abstraction are shared between all abstractions in the function group.

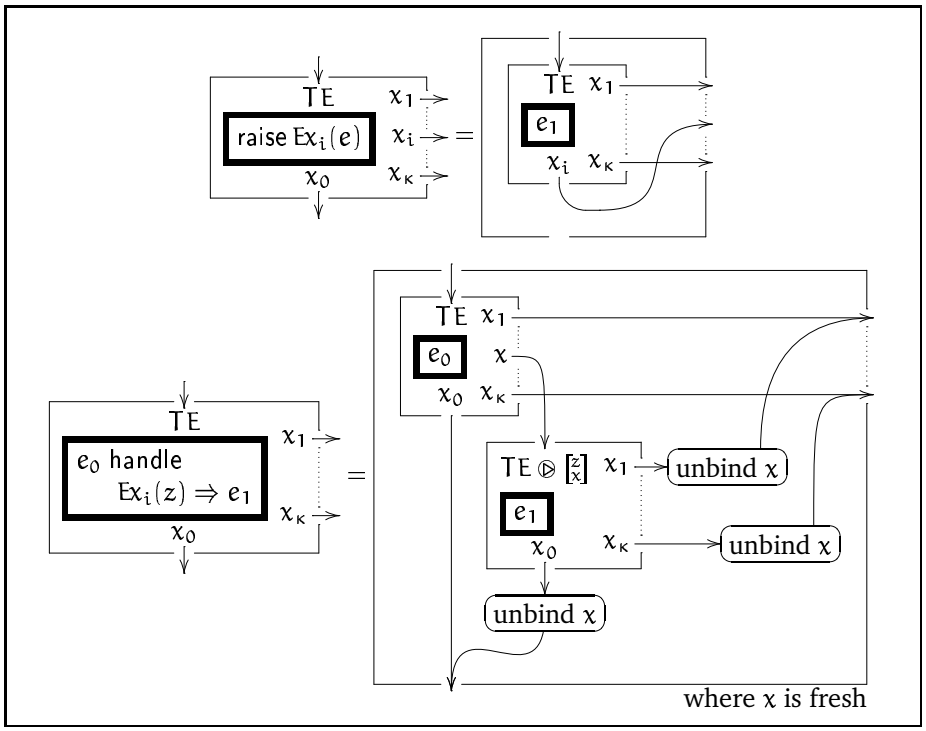


Figure 4.9: Translation for throwing and catching exceptions.

In the function body, UHL variables `F1` through `Fn` represent the free variables of the abstraction. They could just have specified as x_i's with a freshness side condition, but it is simpler to name them explicitly when there is no risk of collision. Similarly, the code that allocates the closure uses fixed temporary variables `C0` through `Cn` to store the contents of the closure before the allocation.

The variable `R` is used as a temporary pointer into the closure during the callee's initialization – the original pointer `C` is kept unchanged so that it can be used for recursive references to the function from within its own body.

If a global exception analysis is available, it can be used to restrict the number of exceptional exit nodes s_i^ℓ to the ones for exceptions that can actually be thrown by one of the functions in the procedure group. Such dead-code elimination would improve the efficiency of the region inference and would prevent it from generating region operations for exceptional exits that are never taken. It will probably not have any effect on the efficiency of the generated agent (but it may be possible to construct pathologic counterexamples to this conjecture).

4.2.5 Simpe exceptions

The unsurprising translation for the exception primitives are shown on Figure 4.9.

4.3 Region annotations for ML

The only task left in a full specification of a translation from our ML fragment to UHL is to decide which edges in the flowchart to consider "annotatable" in the sense of Section 3.4. This question is tightly connected with the problem of expressing region annotation as actual annotations on the ML source syntax, so let us consider them together.

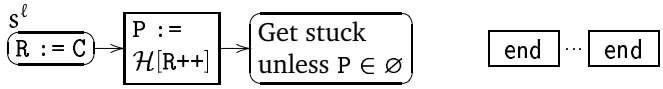
The easy task is expressing the $\mathfrak{at}$ annotations on cons nodes. We can simply attach each of them to the ML expression that is directly responsible for the cons node:

$$\begin{array}{lll} \text{At annotations:} & \mathfrak{at} ::= & \mathsf{at}\,\rho \\ & & \mid \mathsf{nowhere} \\ \text{Expressions:} & e ::= & \cdots \\ & & \mid (\mathsf{fix}\, z_0.\lambda z_1.e)\ \mathfrak{at} \\ & & \mid (e_1 :: e_2)\ \mathfrak{at} \end{array}$$

The annotations for procedure calls are only sligthly more challenging. We need to decorare each application expression with enough information to recover a $\mathfrak{Ma}$ and a $\mathfrak{Ua}$ for the call itself and one $\mathfrak{Ma}$ for each return from the procedure group. The problem is mostly one of notation. In the HMN system it was solved by choosing, arbitrarily, an ording of the callee-sides of each of the maps, and listing the caller-variables in a "call annotation"

$$\text{Call annotations: } \xi ::= \textbf{[c:}\ \rho_1, ..., \rho_n;\ \textbf{i:}\ \rho'_1, ..., \rho'_{n'};\ \textbf{o:}\ \rho''_1, ..., \rho''_{n''}\textbf{]}$$

All function definition in the group then carries identical corresponding ξ's with the callee-variables in the canonical order. (If the group contains no function definitions at all, then the names of the callee-variables are not important, because the entire UHL code for the procedure group will be



which contains no opportunity of even mentioning any region variables in the callee, because the two edges are non-annotatable).

We might use the same strategy for the translation we have defined here, except that we would need several "**o:**" sections in each ξ: One for the normal return and one for each exception.

Things get more interesting when we turn to region operations and the selection of annotatable edges. A simple inspection of each case in the translation shows us that all the requirements of Section 3.4 will be fulfilled if we decide that the annotatable edges are excatly the ones that cross an "expression boundary" – the boxes that symbolize the translation of a single subexpression. As long as we do not consider exceptions, this strategy gives rise to the HMN syntax of region operations:

$$\begin{array}{lll} \text{Expressions:} & e ::= & \ldots \\ & & \mid e\ [\![\mathfrak{Rop}]\!] \\ & & \mid [\![\mathfrak{Rop}]\!]\ e \end{array}$$

where the region operations for each annotatable edge can be represented as a cascade of zero or more postfix or prefix region operations, according to whether the edge crosses the expression boundary on its way in or on its way out. However, this correspondence is not perfect, because many edges cross more than one expression boundary. We can resolve most of this ambiguity by deciding that the region operations for an annotatable edge always attach to the *first* expression boundary crossed by it.

There is, however, one problem left with this naive strategy. Consider, for example, the translation for handle expressions in Figure 4.9. There are two different edges leaving the expression through the normal exit. Where are we to notate annotations on the one coming from the (unbind z) that follows e_1? It happens that every expression boundary that this edge crossed is also crossed by the edge(s) that come out of e_0 (*i.e.*, when an expression is not thrown). No matter where we notate the annotations, it will look as if they apply to the other edge, too.

One way of solving this would be to enforce the invariant that only one edge leaves each expression box. This would require a number of no-op join nodes to be added to the translation. Another option would be to eliminate the (unbind z) node by restructuring the translation such that discarding variables becomes the task of the last expression that uses them. This would actually improve the performance of the region inference a tiny bit, but it would be notationally inconvenient.

But there is much easier solution: Simply use the pruning algorithm from Section 3.4.3. It will never leave an edge annotatable if it goes from a non-split misc node whose incoming edges are all annotatable. Therefore, the problem dissolves completely, and so does a similar problem for the case translation.

The pruning algorithm is quite well suited to being integrated with the translation to UHL itself. In particular, the computation of which kinds of nodes can reach a given node "through edges that were non-annotatable initially" can be statically read off from each individual case in the translation. The only problem is identifying join-nodes while the flowchart is being built. It will be most convenient to be slightly imprecise and always consider the ordinary exit target s_0 for a conditional, case, or handle expression to he a join-node (even though that may not actually be the case if one of the branches is a raise expression). Similarly, the exceptional exit targets s_i are always considered to be join-nodes.

Purging the annotatable nodes also means that edges that leave an expression box by one of the exceptional exits will not usually be annotatable. That is a good thing, because it means that we don't have to invent a ML-level syntax for annotations on these edges. The exceptions to these rules are the exceptional return edges from a call node, and the exceptional edge from arithmetic operations (divide-by-zero etc.).

The annotations for exceptional returns for function calls can be lumped together with the $\mathfrak{Ra}$ return specification in the call annotation. Call annotations may end up being fairly hard to read, but there seems to be no good alternative to this, except for trying to minimize the number of exceptional return edges by a global exception analysis. For divide-by-zero one would put the region operations as an annotation on the operator itself, *e.g.*,

$$\text{Expressions: } e ::= \ldots \mid e_1 \,\mathtt{div}^{\mathfrak{Rop}_1 \ldots \mathfrak{Rop}_n}\, e_2 \qquad (n \geq 0)$$

This completes the description of how to map region expressions for the intermediate language back to the ML source for display purposes. It is sometimes desirable to be able to go the other way: to take an ML source with region annotations in it and translate create the intermediate-language representation directly, skipping the region inference step. This could be necessary, for example, in an experimental system, or of the region inference has been implemented as a separate program for engineering reasons. In that case, we need to let the programmer put prefix and postfix region annotations on *any* expression; it is not reasonable to expect human users to predict which edges will still be annotatable after the pruning phase. Instead the edges for which the user has specified region operations must be artificially prevented from having their annotatability pruned away. If the annotated ML program specifies region operations for an expression boundary that is crossed by multiple edges, the operations can either be duplicated across all the edges, or (better) a no-op misc node can be inserted to join the edges before the region operations.

4.4 Extending to full Standard ML

The techniques we have presented so far ought to scale to most constructions commonly found in ML-like languages without major trouble. We will leave it to the reader to imagine most of the details and only comment on the features that are not straightforward.

Among the straightforward features are tuples and records, which can be implemented directly using UHL's heap primitives. An implementation may decide to *unbox* tuples in certain situations; the outcome of such a transformation can also be represented naturally in UHL. For example, because UHL can represent functions that return multiple values simultaneously, a tuple-shaped return value can be unboxed.

4.4.1 Datatype and pattern matching

Datatypes can also be represented easily. In the most general representation, a value of a datatype is a pointer to a two-word structure containing a tag and the argument, or a magic non-pointer value in the case of a nullary constructor. There are a number of commonly used optimizations, such as unboxing a constructor argument of tuple type or omitting the tag if there is only one non-nullary constructor, as examplified by the special-case list representation on Figures 4.6 and 4.7. These tricks all have natural UHL representations.

Pattern matching on one level of datatypes at a time follows the pattern of Figure 4.7. Standard ML also has complex patterns and matches that cannot be expressed directly as a decision tree of primitive matches, such as the one shown in Figure 4.10. Because there are no *a priori* restrictions on the shape of a UHL flowchart, we have no problem representing such matching operations.

It might be noted, however, that the translation sketched on the figure contains a join-node – $\boxed{\mathtt{K} := \mathcal{H}[\mathtt{Y}{+}{+}]}$ – which is reached from split-nodes by paths that stay completely within the pattern-match construction. If we stay with the principle that only edges that cross expression boundaries can be annotatable, we cannot satisfy the rule from Section 3.4.2 that there should be an annotat-

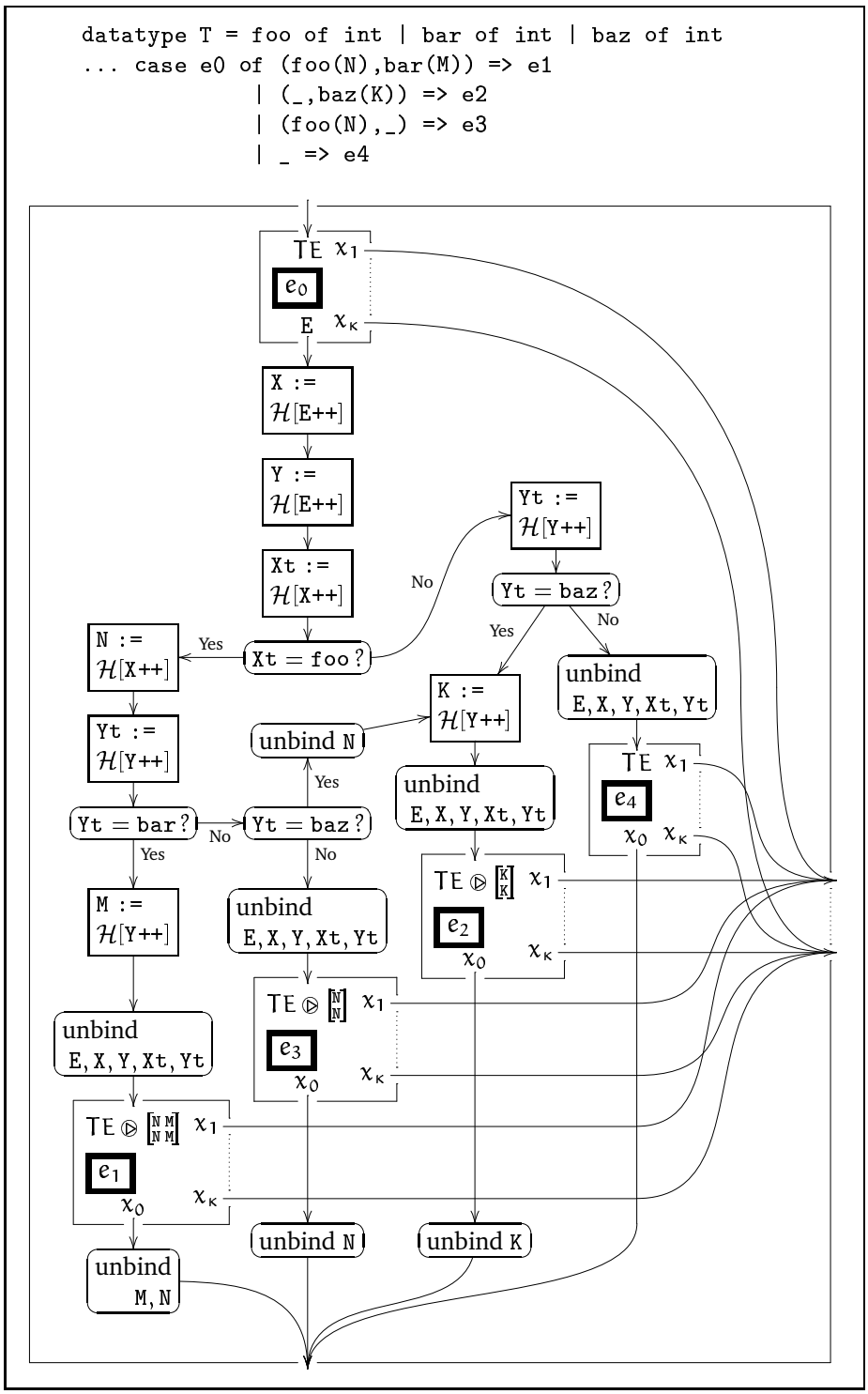


Figure 4.10: Example of a complex pattern-match operation

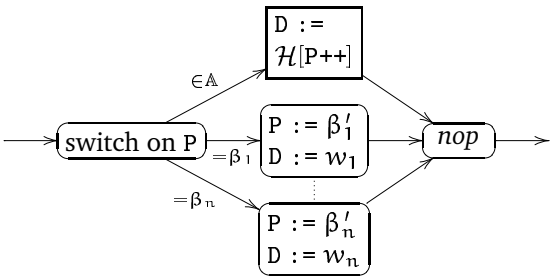
able edge on such a path. Luckily this rule is not a hard requirement but just a general principle for where it would be useful to have annotatable edges. In this particular case, our region inference algorithms would not get any benefit from having an annotatable edge in the middle on the pattern match (because the types are the same throughout the matching, basic region inference would need to create the same region operations for all paths to a particular e_i), so there is no reason to worry about the rule here.

4.4.2 References

References with destructive update is straightforward too; we show the translation on Figure 4.11 just because it is our only chance to show a translation involving write operations. The "⟨arbitrary⟩" at the end of the := translation is the representation of the unit value of the update expression.

4.4.3 Constant expressions

One optimization that is frequently used to a larger or smaller degree by implementations is to translate expressions that construct constant immutable data structures to static read-only data rather than allocate a copy of the structure on the heap each time the expression is evaluated. Expressing this directly in UHL is somewhat tricky, because the read and write primitives in UHL are supposed to be used only with pointers to memory allocated on the heap by the cons operation. Therefore, in an accurate UHL model for the program, we need to represent pointers into static read-only data as special values $\beta \in \mathbb{W} \setminus \mathbb{A}$. We must then instrument each read in the program that might referece the static constants, like



However, it is not really necessary to present this structure to the region inference algorithms. For an agent to be region safe with respect to the instrumented read, we just require that *if* the value of P is a pointer *then* what it points to is still live. But that is also the criterion for being region safe for the original uninstrumented read, so the region inference will do just as good a job if we show it the simpler, but not fully correct, code for the read.

4.4.4 Standard ML exceptions

It is not trivial either to support the full exception concept of Standard ML well. Of course, one straightforward strategy would be to say that Standard ML has exactly one kind of exception whose argument type happens to be the extensible datatype exn. This would be true to how Milner et al. [1997] actually specify

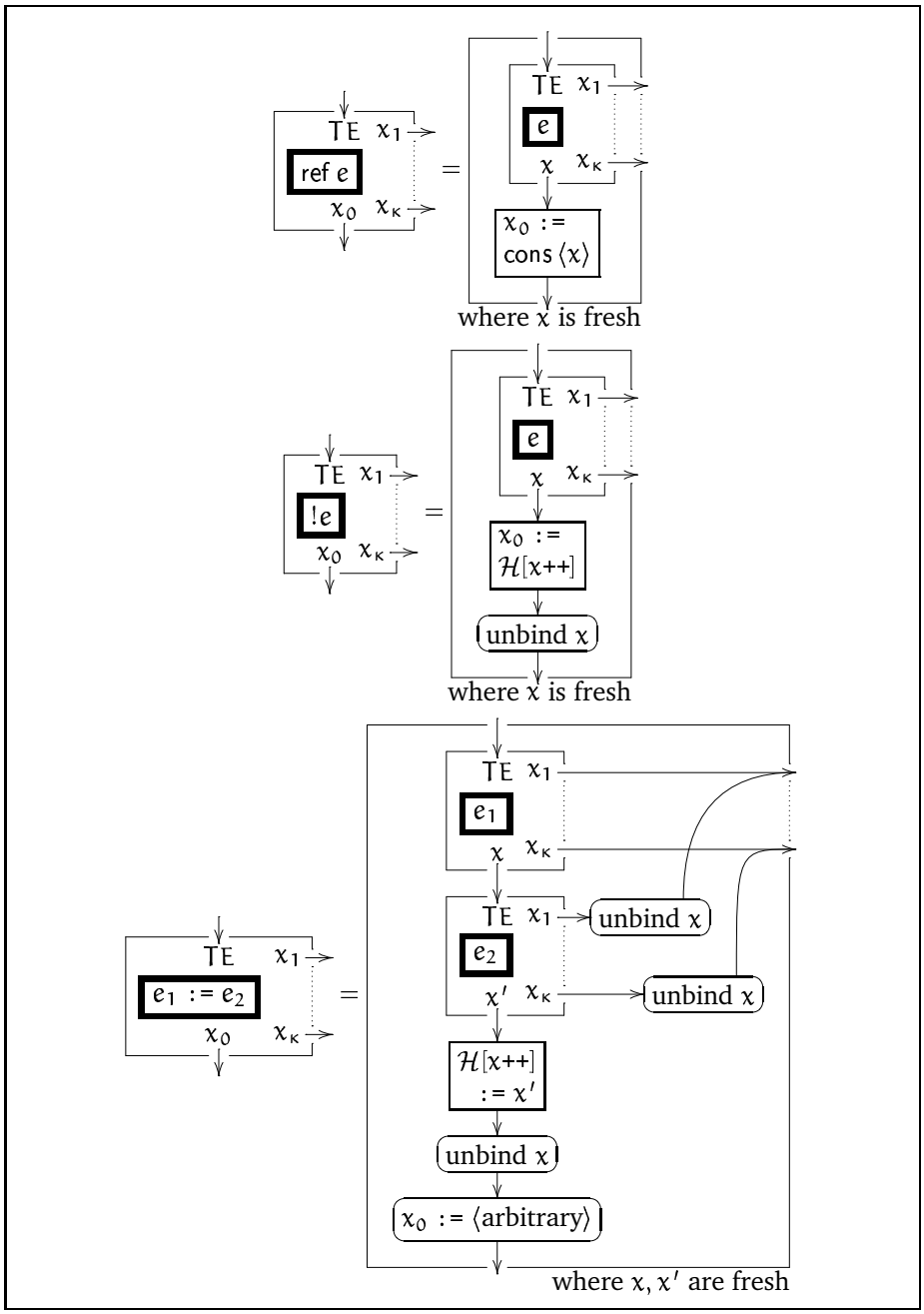


Figure 4.11: Translation for references.

exceptions to work, but in practise it is common to raise and handle specific exceptions rather than just arbitrary values of type `exn`. Lumping everything together in one uhl-level exception would mean losing the knowledge that *this* raise expression may be caught by *that* handler, but not by the one over there.

Whether or not this is actually a problem depends on the programming style in the original program. If it uses exceptions only as a bail-out mechanism for catastropic failures, it is not very important how they are handled by the region inference. But for programs that are written to raise and handle exceptions in the course of normal program execution, it will be inefficent to use a system that means that one raise expression must initialize region variables that are only used by a handler that will never apply to the exception thrown.

Therefore (and since we are only concerned with whole-program analysis in this work anyway) I propose to create one UHL-level exception (*i.e.*, one dedicated `end` state for each procedure group) for each `exception` declaration in the program. Then `raise` exceptions will have do to a case analysis of the `exn` value to find out were to jump to, except in the common case that the form of the expression inside `raise` explicitly names the exception.

A catch-all `handle` will either have its handler expression duplicated for each of the concrete exceptions that can be caught, or have a series of subhandlers that constructs a fresh `exn` value and jumps to a single copy of the programmer-written handler. The former choice will be good for handlers that rethrow the same expression after some clean-up; the latter is suitable for handlers that ignore the exception value or explicitly treat it as first-class data (by storing it in other data structures, etc). I have presently no proposals for how to distinguish automatically between those two cases.

Local exception declarations pose a special problem. In Standard ML, a function such as

```
fun freshexp() =
    let exception E
        fun guard f x = SOME(f x) handle E => NONE
    in (E,guard) end
```

produces a pair of an `exn` and a function that knows how to catch exactly that exception. If `freshexp` is called twice in succession, the `guard` part of one return value will not catch the `E` part of the other. Implementations typically achieve this effect by letting the two exception constructors really be as different as if they were declared separately, but that strategy will be no good for static analysis. Satisfactory handling of this situation in the UHL translation will probably have to involve

- An escape and reentrancy analysis to identify local exception declarations that can safely be lifted to the top level
- The UHL representation of a handler for an exception that is still local after this must start with a test that it was the right instance of the exception that was caught.
- Such a test can be expressed by giving the exception a sequence number (for the instance of the exception declaration) as an artificial argument.
- The master counter for the sequence number can be threaded through the entire UHL program, in and out of every function call. However, because

it will just be an integer, it will not affect the region inference, so the threading does not actually need to be done, just imagined.

4.4.5 Mutual recursion

Mutually recursive functions (`fun` ... `and` ...) may or may not be simple to represent, depending on how the implementation handles them at runtime. If if just creates a set of closures that contain pointers to each other, the knot can be tied by appropriate write operations after the closures have been allocated. Other implementations may allocate a single heap block containing partially overlapping closures. Then functions with higher closure addresses need to index backwards from their own closure in order to find their brethren. This requires adding UHL constructions that decrement pointers in a controlled way, as described at the end of Section 3.3.2, such that we can still reason from region safety to region soundness.

4.4.6 The module language

Finally, I propose that the ML module language can be handled by unfolding the entire program to a core program before region inference. After all, this thesis explicity does not handle separate compilation. (A number of existing ML compilers, including the ML Kit, use this strategy – the ML Kit documentation calls it “static interpretation” – but the ML Kit does the unfolding *after* region inference).

Chapter 5

A region type system for UHL

In Section 1.3 I defined

> *A **region type system** is a structured (and algorithmically feasible, in some sense) characterization of a class of mutator–agent pairs, such that all pairs in the class are region sound.*

If we design our region inference algorithm such that its output will always be accepted by the region type system, we will then know that the agents it producess will always be sound. Or, if we don't trust the producer of agents (we may be afraid of bugs in the implementation of our region inference, or an unknown entity across the net which supplies an agent with applet code that it wants us to run), we could ask him to provide a certificate according to an agreed-upon region-type-system such that we could trust the particular agent he sends us.

A secondary function of the region type system is to guide the design of an *algorithm* for basic region inference. The algorithm can be understood directly as an attempt to construct the most fine-grained typing for the uniform mutator that will be accepted by the region type system.

In this chapter we will develop a region type system that could be applied to UHL programs produced by the translation from ML in Chapter 4 – *as long as the original programs do not use type polymorphism*. It is still unknown what the best way to add ML-style polymorphism to the system is.

As in most extant region type system, we will not argue directly that the agents accepted by the system is region *sound*. Instead will prove (in Section 5.3) that all accepted agents are region *safe*, which, by Theorem 3.46 that an agent which is accepted by the region type system must be region sound. (Remember that an soundness is the *extrinsic* property that the run-time behavior of the agent-mutator combination is the same as the mutator's ideal behavior; safety is the *intrinsic* property that no deallocated heap cell will be referenced).

It is worth emphasizing at this point that the words "type system" in the phrase "region type system" refers to the internal structure of the system, and not to its role in the larger theory. This means that one should not expect the region type system to have the same properties that are in general expected of type system.

For example, we put relatively little weight on the feasibility of *type checking* an agent in the region type system. In some applications it is, of course, important to be able to create and check certificates that an agent satisfies the region type system, but in such an application one simply has to include enough of the actual typing in the certificate that it can be reconstructed with sufficient ease.

Likewise, we shall be concerned not at all with whether the region type system is *decidable* in the complete absence of hints from a certificate. I suppose that it is not.

5.1 A stepwise introduction

Though each ideas that go into the region type system is by itself fairly simple, the final construction ends up being so complex that it would only lead to confusion to start by presenting it in full. The complexity comes partly from generality (we want to present a system that can easily be generalized to many of the languages that UHL works for), and partly from the fact that is has to work with the fairly low-level UHL representation – which itself stems from the generality of UHL. As sources of complexity go, this is a rather benign one. It means that it will usually be possible to specialize the type system for a particular way of producing UHL code, and have much of the complexity go away in the process. Therefore, most concrete implementations of region inference that builds on our principles will not normally be as complex as the raw region type system might seem to imply.

To tackle the complexity in installments, we will present the features of the region system one at a time. This will also give us the opportunity to derive the region type system from first principles, rather than simply asserting that this is how it needs to be. Our starting point will be the bare desire to reason statically about the safety of annotated uniform mutators that are not too different in structure from the ones we constructed in Chapter 4.

The exposition will remain informal until we reach the full system. Readers who are eager for formality may skip directly to the formal definitions to Section 5.2; they do not depend on any definitions in the preceding discussion.

5.1.1 Basics: Simple pointers and procedure calls

Let us, initially, restrict our attention to mutators where all cons operations are unary (*i.e.*, allocate one heap cell only), and also assume that values of heap cells never contain pointers.

These are, of course, madly unreasonable premises – if observed in reality they would prevent the mutator from making any nontrivial use of the heap. However, they will allow us to investigate the essential *region-oriented* features of the region type system without caring about what it is that the pointers actually point to. These essential features are fairly simple. Once we have a good grasp of those and begin enlarging our horizon towards more realistic mutators, we will need to add a lot of auxiliary constructs, but they will not have much significance for regions as such; essentially they will just be a scaffolding on which to fasten well-known kinds of region information.

In this primitive setting we can begin to construct the simplest imaginable region type system:

$$\begin{array}{lll} \text{Places:} & \mathfrak{p} ::= \rho \\ & \quad \mid \top \\ \text{Environments:} & \Gamma \in \boxed{\boxed{\mathfrak{x}}} \xrightarrow{\text{fin}} \boxed{\boxed{\mathfrak{p}}} \\ \text{Typings:} & \mathcal{T} \in \boxed{\boxed{\mathfrak{s}}} \xrightarrow{\text{fin}} \boxed{\boxed{\Gamma}} \end{array}$$

A Γ describes the relationship between mutator variables $\mathfrak{x}$ and region variables ρ at some point in the program:

$\Gamma(\mathfrak{x}) = \rho$ means that *if* the value of $\mathfrak{x}$ is a pointer, *then* the cell it points to is being kept alive by the region currently bound to ρ.

$\Gamma(\mathfrak{x}) = \top$ means that the value of $\mathfrak{x}$ may be a pointer to an unallocated address in the heap, so it must not be used for heap access. (The reason for using the symbol "$\top$" will become apparent in a few pages).

The typing for an entire program will be given by a $\mathcal{T}$ which assigns an Γ for each control state.

We can then have rules like

$$\frac{}{\{\Gamma\}\ \mathsf{new}\ \rho^c\ \{\Gamma\}} \qquad \frac{}{\{\Gamma[\rho^c \mapsto \rho]\}\ \mathsf{alias}\ \rho\ \mathsf{to}\ \rho^c\ \{\Gamma\}}$$

$$\frac{\rho^c \notin \mathrm{Img}\,\Gamma}{\{\Gamma\}\ \mathsf{release}\ \rho^c\ \{\Gamma\}} \qquad \frac{}{\{\Gamma\}\ \mathsf{rename}\ \rho_1^c\ \mathsf{to}\ \rho_2^c\ \{\Gamma[\rho_1^c \mapsto \rho_2^c]\}}$$

$$\frac{\mathcal{T}(s) = \Gamma}{\mathcal{T} \vdash \{\Gamma\}\ \mathsf{goto}\ s} \qquad \frac{\{\Gamma_1\}\ \mathfrak{Rop}\ \{\Gamma_2\} \qquad \mathcal{T} \vdash \{\Gamma_2\}\ \mathfrak{Jmp}}{\mathcal{T} \vdash \{\Gamma_1\}\ \mathfrak{Rop}; \mathfrak{Jmp}}$$

$$\frac{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\mathfrak{x}\\ \rho\end{smallmatrix}\right]\}\ \mathfrak{Jmp}}{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\mathfrak{x}'\\ \mathfrak{p}\end{smallmatrix}\right]\}\ \mathfrak{x} := \mathsf{cons}\,\langle \mathfrak{x}' \rangle\ \mathsf{at}\ \rho;\ \mathfrak{Jmp}} \qquad \frac{\mathfrak{p} \neq \top \qquad \mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\mathfrak{x}' & \mathfrak{x}\\ \top & \top\end{smallmatrix}\right]\}\ \mathfrak{Jmp}}{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\mathfrak{x}\\ \mathfrak{p}\end{smallmatrix}\right]\}\ \mathfrak{x}' := \mathcal{H}[\mathfrak{x}{+}{+}];\ \mathfrak{Jmp}}$$

and so forth.

The rule for release has a side condition that prevents a region from being deallocated as long as Γ says that there are pointers that point into the region. Therefore, as long as $\Gamma(\mathfrak{x})$ is *some* ρ, no matter which, the cell it points to will still be allocated, and therefore the rule for read operations actually does guarantee that the program will never go Wrong. After the read operation, neither the pointer (which has been incremented past the allocated cell) nor the value read (which our primitive system does not keep track of) can be assumed to point to anything.

The rule for rename operations simply say that after a region renaming, all values that were previously being kept alive by the old name for the region are now kept alive by the new name. The rule for alias is more interesting. It says that something that was kept alive by ρ before the alias operation may now be considered to be kept alive by *either* ρ *or* ρ^c. The formal structure of this rule is seen to coincide with the axiom for assignment statements in Hoare logic

$$\frac{}{\{Q[\mathfrak{x}_2 \mapsto \mathfrak{x}_1]\}\ \mathfrak{x}_2 := \mathfrak{x}_1\ \{Q\}}$$

which is the ultimate reason why we use Hoare-like braces in the notation for our judgment forms. Niss [2002] has investigated the relation between Hoare logic and the HMN region type system in detail, but we will not pursue this connection further here.

The rule for procedure calls would be something like

$$\frac{\Gamma' = (\mathfrak{Ra} \oplus \mathfrak{Ua} \oplus \left[\begin{smallmatrix}\top\\\top\end{smallmatrix}\right]) \circ \mathcal{T}(s) \qquad \mathrm{Dom}\,\Gamma \cap \mathrm{Img}\,\mathfrak{Ra} = \varnothing \qquad \forall s' \in \mathcal{E}(s) : \begin{cases} \mathcal{T} \vdash \{\Gamma''\}\ \mathfrak{Jmp} \\ \quad \text{where } \Gamma'' = \Gamma \oplus \left((\mathfrak{Ra}' \oplus \mathfrak{Ua} \oplus \left[\begin{smallmatrix}\top\\\top\end{smallmatrix}\right]) \circ \mathcal{T}(s')\right) \\ \qquad \left(\mathfrak{Ra}', \mathfrak{Ma}', \mathfrak{Jmp}\right) = \mathfrak{Rtn}(s') \end{cases}}{\mathcal{T} \vdash \{\Gamma \oplus \Gamma'\}\ \mathsf{call}\ s(\mathfrak{Ra}, \mathfrak{Ua}, \mathfrak{Ma});\ \mathsf{then}\ \mathfrak{Rtn}}$$

There are two interesting points in this rule. One is the side condition that says that $\mathrm{Img}\,\mathfrak{Ra}$ must be disjoint from $\mathrm{Dom}\,\Gamma$. $\mathrm{Img}\,\mathfrak{Ra}$ are the actual region parameters that are consumed by the call. The side condition prevents them from being used in the region description for the caller's local variables after the call. This is parallel to the side condition for release mentioned above.

Perhaps more surprising is the construction of Γ''. Here $\mathfrak{Ua}$ suddenly appears in the part of the rule that is concerned with returning *from* the procedure, whereas in the entire operational development for UHL in Chapter 3, $\mathfrak{Ua}$ has only been used in for calling *into* the procedure! This means that if, at the end node, $\mathcal{T}(s')(x)$ is an uncounted region variable ρ^u, we can map it back to the region variable in the caller that was used to initialize ρ^u at the time of the call, and use that variable as the new description of $\mathfrak{Ma}(x)$. This is a safe reasoning principle because no region operations are allowed to change the value of $R(\rho^u)$.[1] Without this ability to reason through $\mathfrak{Ua}$, the caller-variables that receive the function's return value could never have a region description that was identical to that of a value that the caller computed before the call and did not pass as a parameter (except if both descriptions were $\top$ which is not helpful). Having identical region descriptions will be important once we add recursive types to the system in Section 5.1.7.

5.1.2 Subplacing: HMN without complex data structures

The initial system sketched so far (we have not shown rules for read or misc nodes, but they would contain no surprises except for some unpleasant notation) is like the first-order fragment of the in that it will keep a region alive for as long as a pointer into it is in scope. The UHL-based model for region allocations allows the scope to be tracked tighter than the Tofte and Talpin's `letregion` construct, but it is still the case that if the variable `X` may contain a pointer into some region, that region cannot be deallocated until `X` has been explicitly unbound or consumed in some other way.

It should be intuitively clear that this is suboptimal. It is not a good reason not to deallocate a memory block that a pointer to it happens to *exist*; only if the mutator may need to *access* the block do we have to keep it alive. For a garbage collector the mere existence of a pointer (in scope) to the block is sufficient reason not to collect it, but that is because the garbage collector does not know what the mutator will do. Region inference happens statically, so it ought to be able to do better.

[1] The rule would not have been safe if we had allowed uncounted region variables to be renamed – which is the actual reason why uncounted regions must not be renamed. Renaming of uncounted variables would not have hurt Proposition 3.24, for example, as long as the new name for an uncounted variable was still uncounted.

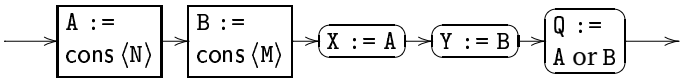
Unfortunately, the simplifying assumptions we are still working under makes it hard to construct a really convincing concrete example of where we can do better. In most situations it would be possible simply to unbind the variable as soon as it is not going to used anymore, but consider this scenario: An algorithm in the beginning of the program produces in `X` either nil or a pointer to an integer. Much later, some decision will depend on whether the result was nil or not, but not on the value of the pointed-to integer, if any. Because a simple comparison between `X` and nil does not depend on the memory still being allocated, it will be safe to deallocate the region where the integer lived as soon as the initial algorithm finished. But the reasoning system we have so far will not allow this, because $\Gamma(\mathtt{X}) = \rho$ prevents the deallocation.

An immediate idea for solving this would be to replace the rule for release by

$$\overline{\{\Gamma\}\ \mathsf{release}\ \rho^c\ \{\Gamma[\rho^c \mapsto \top]\}}$$

which allows deallocation of a region at any time, but rewrites Γ such that no later heap access may use a variable that might have been a pointer into the deallocated region. This rule turns out to be sound, and its generalization to settings with complex data structures will be a qualitative improvement over TT-style systems.

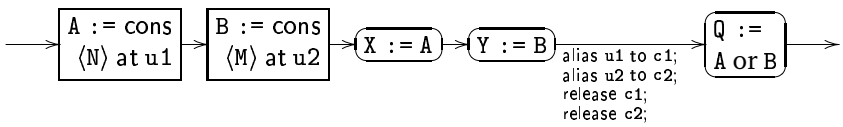
It turns out that the ability to change $\Gamma(\mathtt{X}) = \rho$ to $\Gamma(\mathtt{X}) = \top$ may be beneficial even in situations where we have no immediate reason for releasing ρ. Take, for example, the code fragment



where `Q := A or B` stands for a misc node that nondeterministically chooses between assigning `Q` the value or `A` or that of `B`. We have not given a general rule for misc nodes, but it is intuitively clear that we need $\Gamma(\mathtt{Q})$ to equal $\Gamma(\mathtt{A})$ and $\Gamma(\mathtt{B})$ simultaneously. By transitivity, we need $\Gamma(\mathtt{A}) = \Gamma(\mathtt{B})$. (Once we introduce recursive types, we will see that less contrived reasons for needing $\Gamma(\mathtt{A}) = \Gamma(\mathtt{B})$ exist).

Further assume that the code following the example will access the heap through `X` and `Y` – but not necessarily equally often – but never through `A` or `B`.

At first sight it would seem that we must allocate `A` and `B` in the same region – we cannot have $\Gamma(\mathtt{A}) = \Gamma(\mathtt{B}) = \top$, because that would prevent the later accesses through `X` and `Y`. However, consider the following trick:

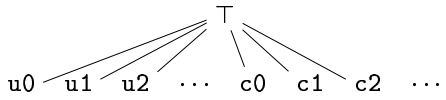


With these annotations we can have $\Gamma = \left[\begin{smallmatrix}\mathtt{A} & \mathtt{B} & \mathtt{X} & \mathtt{Y}\\ \mathtt{c1} & \mathtt{c2} & \mathtt{u1} & \mathtt{u2}\end{smallmatrix}\right]$ before the two release operations and $\Gamma = \left[\begin{smallmatrix}\mathtt{A} & \mathtt{B} & \mathtt{X} & \mathtt{Y}\\ \top & \top & \mathtt{u1} & \mathtt{u2}\end{smallmatrix}\right]$ after them. Thus, by inserting a sequence of region operations with no net operational effect at all, we can certify the safety of a program that does allocate `A` and `B` in different regions. Clearly it is not desirable to need to do such spurious reference-count manipulation, so we extend the system with a general **subplacing** rule:

$$\frac{\mathcal{T} \vdash \{\Gamma \oplus [{}^{x}_{\top}]\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \{\Gamma \oplus [{}^{x}_{\rho}]\}\ \mathfrak{Imp}}$$

Now we can derive the new release rule from the old one and some number of applications of the subplacing rule. Therefore we go back to the original release rule with the $\rho^c \notin \mathrm{Dom}\,\Gamma$ side condition – it seems cleaner to restrict rewriting of Γ's to as few rules as possible.

In the classic type-theoretic concept of subtyping, "$\top$" is a type that describes all values and therefore is a supertype of any other type. This matches well with the role of $\top$ in the subplacing order on $\boxed{p}$:

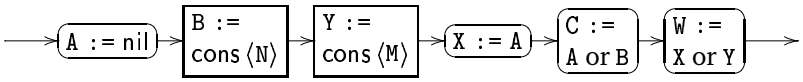


This diagram, and the connection to subtyping in general, suggests that we consider a dual to $\top$, written $\bot$. Its rule is

$$\frac{\mathcal{T} \vdash \{\Gamma \oplus [{}^{x}_{\rho}]\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \{\Gamma \oplus [{}^{x}_{\bot}]\}\ \mathfrak{Imp}}$$

and the intuitive meaning of $\Gamma(x) = \bot$ is that the value of x is not a pointer at all, that is, it belongs to $\mathbb{W} \setminus \mathbb{A}$. In that case, if we try to access the heap using x, the read/write operation would get stuck immediately instead of going Wrong, which means that such an access *can* actually be allowed in a region-safe agent. Therefore it is also safe to jump from code that assumes $\Gamma(x) = \bot$ to code that assumes $\Gamma(x) = \rho$ for any ρ.

An example where $\bot$ is useful is



where the following code accesses the heap through `C` and `W`. The use of $\bot$ will allow us to allocate `B` and `Y` in different regions, if we let $\Gamma = [{}^{A\ B\ X\ Y}_{\bot\ c1\ \bot\ c2}]$ just after $\boxed{\texttt{X := A}}$, and then lift it to $[{}^{A\ B\ X\ Y}_{c1\ c1\ c2\ c2}]$ using two subplacing steps.

In the full system with complex data, $\bot$ will have the more subtle task of describing values that are known not to be pointers because they do not exist at all! This possibility may arise with lists – for example, the constant "nil" can have the type "list of pointers to something at $\bot$". Closures or tagged sums is another possibility; the thought experiment that originally led me to invent $\bot$ involved considering an ML program like

```
fun f x = x+5
fun work xs = (if ... then f
                      else let fun g x = hd xs in g end)(42)
```

where `f` and `g` are in the same procedure group. The region-annotated type for `f` will need to contain a p that describe the `xs` pointer in the closure for `g`. Without $\bot$ this p would need to be some region variable, which would force all lists ever used as an argument to `work` to be allocated in that particular region.

The features of the region type system we have seen yet directly match features the HMN system. From here things begin to diverge.

First, HMN has a rich language of *types*, such that each $\Gamma(x)$ can contain several different p's, corresponding to pointers that can be reached *indirectly* from the value of x. We will add such a type language presently, but it will be more low-level than the HMN one.

Second, HMN's region type system incorporates a scoping discipline for regions that closely matches the one we studied in Section 3.2.3. We will not add such a discipline to our typing rules themselves, for the pragmatic reason that they would make rules like *RTCALL* (on page 134) even harder than it already is. Instead we will stipulate separately that the agent is supposed to be well-formed according to Section 3.2.3

This has the minor consequence that we cannot directly express the condition that whenever a subplacing step changes a $\bot$ to a region variable ρ, ρ will actually be bound to something. We will solve this by arranging the rules such that "unbound ρ" will only appear where $\bot$ *could* just as well have appeared. (However, the region inference algorithms do not depend on this "feature"; it will appear only in the region safety proof in Section 5.3).

5.1.3 Pointers to pointers

Now that we know how places p work, we can now begin to erect the "type scaffolding" where places are kept in the region type system. To a certain extent, these mechanisms depend on the native type system (if any) of the host language, because one wants to be able to map native types to region types. Here we just develop the minimum of features that are needed to handle UHL generated from monomorphic ML programs in reasonable generality.

First, let us throw away the assumption that the values in heap cells are not pointers (we still assume that conses are unary, however). Then, the rule for a read operation such as $\boxed{\texttt{A} := \mathcal{H}[\texttt{P++}]}$ will need to

a. Check that the cell that P points to is still allocated. This is just as before.

b. Somehow construct a new value for $\Gamma(\texttt{A})$. The only reasonable way to do this is to let $\Gamma(\texttt{P})$ *contain* the correct new value for $\Gamma(\texttt{A})$.

This calls for the insertion of a (syntactically) recursive **type** layer between the Γ and the p:

$$\begin{array}{ll} \text{Places:} & p ::= \bot \mid \rho \mid \top \\ \text{Types:} & \tau ::= \langle \tau \rangle_p \\ & \quad \mid \bullet \\ \text{Environments:} & \Gamma \in \boxed{x} \xrightarrow{\text{fin}} \boxed{\tau} \end{array}$$

where the production rule "$\tau ::= \bullet$" is just an artificial base case, to allow finite types. Later we will replace it by other devices.

We can then extend the rules for reads and cons:

$$\frac{\mathcal{T} \vdash \left\{\Gamma \oplus \left[\begin{smallmatrix} x \\ \langle \tau \rangle_\rho \end{smallmatrix}\right]\right\} \mathfrak{Imp}}{\mathcal{T} \vdash \left\{\Gamma \oplus \left[\begin{smallmatrix} x' \\ \tau \end{smallmatrix}\right]\right\} x := \mathsf{cons}\,\langle x' \rangle \text{ at } \rho;\ \mathfrak{Imp}}$$

$$\frac{p \neq \top \qquad \mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix} x' & x \\ \tau & \bullet \end{smallmatrix}\right]\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix} x \\ \langle\tau\rangle_p \end{smallmatrix}\right]\}\ x' := \mathcal{H}[x{+}{+}];\ \mathfrak{Imp}}$$

In the new rule for reads, the final value for $\Gamma(x)$ is now $\bullet$ rather than $\top$. This is just a temporary placeholder that we will replace by something else shortly.

In addition to the two new rules, we also ought to reformulate some of the other rules, because the mathematical expression we have given them depended on a Γ mapping variable names directly to places. Therefore we could write side conditions such as "$\Gamma = (\mathfrak{Ra} \oplus \mathfrak{Ua} \oplus \left[\begin{smallmatrix} \top \\ \top \end{smallmatrix}\right]) \circ \mathcal{T}(s)$" which do not make sense anymore.

However, we shall not bother to define a concrete syntax for such renaming-of-regions-in-environment operations just now, because we would need to modify it each time we add a new feature to the type language. Eventually we will define suitable notation for the final type system in Definitions 5.3ff.

5.1.4 Tuples

Now we're ready to extend the system to handle conses that allocate more than one cell at one time. With the work we have done so far, this is not difficult. We redefine the syntax of types:

$$\text{Types: } \tau ::= \langle \tau_1, \tau_2, ..., \tau_k \rangle_p \qquad (k \geq 0)$$

The $k = 0$ case "$\langle\rangle_p$" intuitively stands for a pointer that has been incremented past the last of the cells it used to point to. It can also be used as a leaf in the type tree, so we can abandon the artificial "$\bullet$" type from the previous subsection.

The updated rules for these types are not surprising, either:

$$\frac{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix} x \\ \langle \tau_1, ..., \tau_k \rangle_\rho \end{smallmatrix}\right]\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix} x_1 & \cdots & x_k \\ \tau_1 & \cdots & \tau_k \end{smallmatrix}\right]\}\ x := \mathsf{cons}\ \langle x_1, ..., x_k \rangle\ \mathsf{at}\ \rho;\ \mathfrak{Imp}}$$

$$\frac{p \neq \top \qquad \mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix} x' & x \\ \tau_1 & \langle \tau_2, ..., \tau_k \rangle_p \end{smallmatrix}\right]\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix} x \\ \langle \tau_1, ..., \tau_k \rangle_p \end{smallmatrix}\right]\}\ x' := \mathcal{H}[x{+}{+}];\ \mathfrak{Imp}}$$

5.1.5 Tagged sums

The support for multi-call blocks in the previous section is not quite enough to express a tagged sum type such as "a heap block that consists of two non-pointer values if the first cell contains the constant `Foo` and four non-pointer values if the first cell contains the constant `Bar`". For this we need another rewriting of the type syntax:

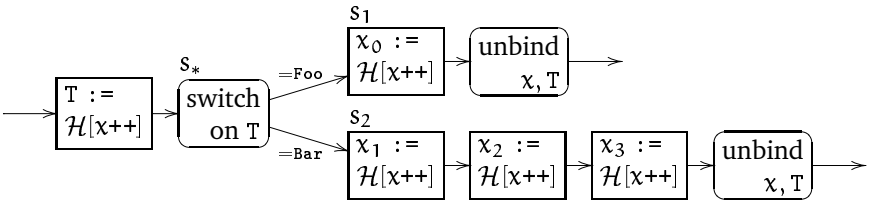
$$\begin{aligned} &\text{Types: } \tau ::= \{w\} && (w \in \mathbb{W} \setminus \mathbb{A}) \\ &\qquad\quad | \ \langle \tau_{11}, \tau_{12}, ..., \tau_{1k_1} \mid \cdots \mid \tau_{n1}, \tau_{n2}, ..., \tau_{nk_n} \rangle_p && (n \geq 1, k_i \geq 0) \end{aligned}$$

The singleton type $\{w\}$ describes the non-pointer value w and nothing else, so we can express "a pointer to a cell containing `Bar` followed by three other cells" as $\langle \{\texttt{Bar}\}, \tau_1, \tau_2, \tau_3 \rangle_p$. The entire tagged sum type is now expressed as

$$\langle \{\texttt{Foo}\}, \tau_0 \mid \{\texttt{Bar}\}, \tau_1, \tau_2, \tau_3 \rangle_p$$

where the bar separates alternatives: The pointer points to *either* a `Foo` followed by a τ_0 *or* a `Bar` followed by τ_1 through τ_3.

How should the existing rules be changed to support such types? Reading the tagged sum (and branching on the tag) turns out to be the most intricate case, so consider the UHL idiom for this:



Now, the problem is to find a type invariant that holds at state s_*. Here the type of x is either $\langle\tau_0\rangle_p$ or $\langle\tau_1,\tau_2,\tau_3\rangle_p$ – but one cannot see from the pointed-to values which is the correct one. Which should $\mathcal{T}(s_*)(x)$ be?

My answer is: both. We extend the definition of a $\mathcal{T}$ such that each state can have not just one environment but a *set* of environments. Then $\mathcal{T}(s_*)$ can be $\left\{\left[\begin{smallmatrix}T\\ \{\texttt{Foo}\}\end{smallmatrix}\begin{smallmatrix}x\\ \langle\tau_0\rangle_p\end{smallmatrix}\right], \left[\begin{smallmatrix}T\\ \{\texttt{Bar}\}\end{smallmatrix}\begin{smallmatrix}x\\ \langle\tau_1,\tau_2,\tau_3\rangle_p\end{smallmatrix}\right]\right\}$, meaning that at least one of the two Γs will describe the actual σ whenever control reaches s_* The rule for $\boxed{\text{switch on T}}$ (which is yet to be defined) will be designed to notice that the two cases match the branches of the switch, such that $\mathcal{T}(s_1)$ and $\mathcal{T}(s_2)$ need only contain one Γ each.

$$\text{Typings: } \mathcal{T} \in \boxed{S} \xrightarrow{\text{fin}} \mathcal{P}_{\text{fin}}(\boxed{\Gamma})$$

The rule for goto becomes

$$\frac{\Gamma \in \mathcal{T}(s)}{\mathcal{T} \vdash \{\Gamma\}\ \text{goto}\ s}$$

and for $\mathcal{A}$ to be well-typed by $\mathcal{T}$, it must hold that $\mathcal{T} \vdash \{\Gamma\}\ \mathcal{A}(s)$ for *each* $\Gamma \in \mathcal{T}(s)$, for all s.

The following rule allows going from one Γ in which x maps to $\langle\cdots|\cdots|\cdots\rangle_p$ to several Γ's that each maps x to a type without bars:

$$\frac{\forall i \le n : \mathcal{T} \vdash \left\{\Gamma \oplus \left[\begin{smallmatrix}x\\ \langle\tau_{i1},\ldots,\tau_{ik_i}\rangle_p\end{smallmatrix}\right]\right\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \left\{\Gamma \oplus \left[\begin{smallmatrix}x\\ \langle\tau_{11},\ldots,\tau_{1k_1}|\cdots|\tau_{n1},\ldots,\tau_{nk_n}\rangle_p\end{smallmatrix}\right]\right\}\ \mathfrak{Imp}}$$

If we insert it in the typing before the first read operations, the typing rules for heap reads will not need to know about types with bars in them. (An alternative would be to build the it into the rule for reads, but a separate rule is stronger, because it allows the typing of schemes where the tag is not the first cell in the block).

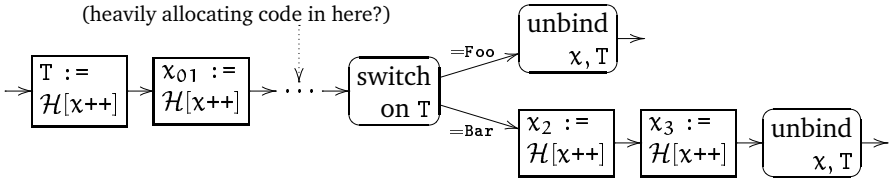
With the dual rule[2]

[2]This rule looks suspiciously like a subtyping rule. Might it not be combined with the subplacing concept? Answer: It might, but it would make the eventual definition of subplacing more complex and turns out not to be necessary for any of the languages I consider here. If, however, the host language has a native concept of structural subtyping, it would be necessary to integrate it with the subplacing concept, and then it would be natural to include such expansion rules in the same package.

$$\frac{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\chi\\ \langle \cdots | \tau_1, \ldots, \tau_k | \cdots \rangle_p\end{smallmatrix}\right]\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\chi\\ \langle \tau_1, \ldots, \tau_k \rangle_p\end{smallmatrix}\right]\}\ \mathfrak{Imp}}$$

we also do not have to extend the rule for cons.

Aside: This mechanism is strong enough to be able to handle low-level "prefetching" optimizations like



which might allow a region to be deallocated earlier if the reads of χ_2 and χ_3 can be optimized away (for, example, if they match a wildcard pattern in ML).

5.1.6 Destructive update

What should the rule for heap writes be? It is clear that the value written into a heap cell should have the type that the heap cell already has. Otherwise later reads from the cell using another pointer will get a value of another type than it expects, breaking the invariant of the type system. That is, unless we can locate and change the type descriptions for all existing pointers to the updated cell – which is a hard problem because some of the existing pointers may lie dormant on the call stack, which is not described by the Γ, and in any case it is not apparent from the type of a pointer whether it refers to the particular cell we're about to update, or just another one with the same type.

Therefore we stick to the "do not change types at destructive updates" rule. It comes from the ML Kit's support for ML's reference type, and though it is not ideal for programs that make heavy use of destructive update (see Section 6.1.1) it is the best anyone has been able to think of so far. The natural attempt to formalize the rule, given the system have developed already, would be

$$\frac{p \neq \top \qquad \mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\chi\\ \langle \tau_1, \ldots, \tau_k \rangle_p\end{smallmatrix}\right]\}\ \mathfrak{Imp}}{\mathcal{T} \vdash \{\Gamma \oplus \left[\begin{smallmatrix}\chi\\ \langle \tau_0, \tau_1, \ldots, \tau_k \rangle_p\end{smallmatrix}\ \begin{smallmatrix}\chi'\\ \tau_0\end{smallmatrix}\right]\}\ \mathcal{H}[\chi{+}{+}]\ := \chi';\ \mathfrak{Imp}}$$

Unfortunately this rule is not sound. The trouble is that the subplacing operations introduced in Section 5.1.2 could be used to subvert the correctness of the place annotations on the types; an example is shown in Figure 5.1. It is well known that such problems arise if one combines subtyping with destructive updates carelessly. The solution is also well known: The type of the contents of an updateable cell must be treated as **bivariant**, that is, it must not change in subplacing steps.

One way to achieve this would be to forbid subplacing beyond the first layer of pointers. But this is much too harsh for our purposes; it would prevent many useful and benign opportunities for early deallocation. Another and better strategy is to *mark* in the type the (few) cells that may be updated later:

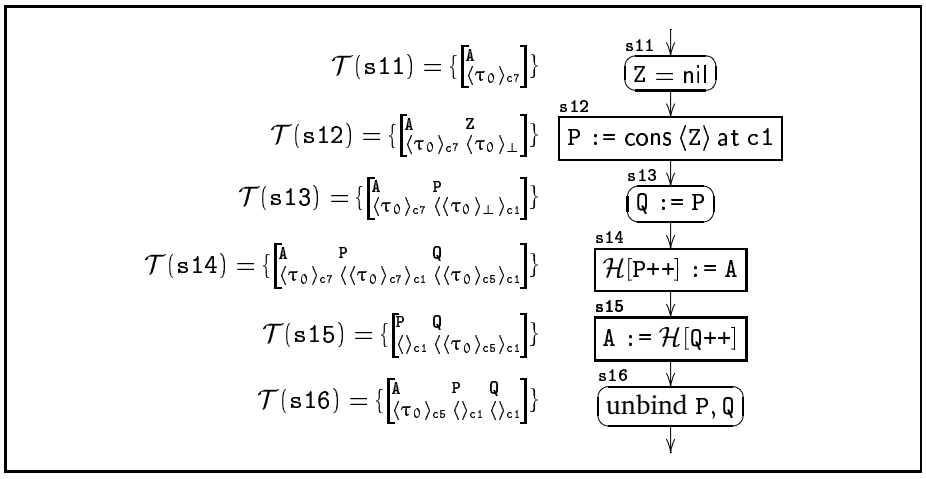


Figure 5.1: Example of how to use subplacing to cheat the naive rule for heap writes. The net effect of the sequence is to change the type of A *from* $\langle\tau_0\rangle_{c7}$ *to* $\langle\tau_0\rangle_{c5}$.

Permissions:	$\nu ::= \mathtt{rw} \mid \mathtt{ro}$	
Cell types:	$\theta ::= \nu{:}\tau$	
Types:	$\tau ::= \{w\}$	$(w \in \mathbb{W} \setminus \mathbb{A})$
	$\mid\ \langle\theta_{11}, ..., \theta_{1k_1} \mid \cdots \mid \theta_{n1}, ..., \theta_{nk_n}\rangle_p$	$(n \geq 1, k_i \geq 0)$

A cell type marked `ro` can be used only for reads, not for writes, but on the other hand the type *inside* the cell is subject to subplacing. A cell type marked `rw` may be written to (or read from), but is treated as bivariant.

We can incorporate this in the definition of subplacing:

$$\frac{}{\perp \leqslant p} \qquad \frac{}{\rho \leqslant \rho} \qquad \frac{}{p \leqslant \top} \qquad \frac{}{\mathtt{rw}{:}\tau \leqslant \mathtt{rw}{:}\tau} \qquad \frac{\tau \leqslant \tau'}{\nu{:}\tau \leqslant \mathtt{ro}{:}\tau'}$$

$$\frac{p \leqslant p' \qquad \forall i, j : \theta_{ij} \leqslant \theta'_{ij}}{\langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots\rangle_p \leqslant \langle \cdots \mid ..., \theta'_{ij}, ... \mid \cdots\rangle_{p'}} \qquad \frac{}{\{w\} \leqslant \{w\}}$$

$$\frac{\mathrm{Dom}\,\Gamma_1 = \mathrm{Dom}\,\Gamma_2 \qquad \forall x \in \mathrm{Dom}\,\tau_1 : \Gamma_1(x) \leqslant \Gamma_2(x)}{\Gamma \leqslant \Gamma'}$$

Note that the fifth of these rules imply $\langle\mathtt{rw}{:}\tau\rangle_p \leqslant \langle\mathtt{ro}{:}\tau\rangle_p$. That will be handy if one needs to use a few destructive writes to "tie the knot" in a circular structure – such as a cluster of mutually recursive function closures – and thereafter use it for reading only. It is safe to *read* from a cell using a subplaced types as long as each *write* to the cell uses the exact same type as was used to allocate the cell in the first place.

These restrictions on read-write cells only concern *subplacing*. The rewriting of region types that occurs in the rules for alias and rename operations, as well as during procedure calls and returns, do not really change the region description of data. On the contrary: They *maintain* the description of data such that it still means that same in a region environment R where the run-time regions are referred to by other region variables.

5.1.7 Recursive types

By now, the only feature we need to add for the region type system to be actually useful is recursion in types. Recursive types are necessary to be able to reason about arbitrarily long chains of pointers-to-pointers; such chains can arise either from explicit use of lists or other recursive data structures, or implicitly because the program builds a chain of closures that refer to each other.

Type recursion is usually viewed as an uninteresting feature that one can "easily" imagine how to add to a system.[3] It is, however, one of those features that are more tricky to handle formally that it intuition suggests. Let us start by stepping back a few paces and look at the syntax of the region type system so far:

Places:	$p ::= \bot \mid \rho \mid \top$	
Permissions:	$\nu ::= \texttt{rw} \mid \texttt{ro}$	
Cell types:	$\theta ::= \nu{:}\tau$	
Types:	$\tau ::= \{w\}$	$(w \in \mathbb{W} \setminus \mathbb{A})$
	$\mid \langle \theta_{11}, ..., \theta_{1k_1} \mid \cdots \mid \theta_{n1}, ..., \theta_{nk_n} \rangle_p$	$(n \geq 1, k_i \geq 0)$
Environments:	$\Gamma \in \boxed{X} \xrightarrow{\text{fin}} \boxed{\tau}$	
Typings:	$\mathcal{T} \in \boxed{S} \xrightarrow{\text{fin}} \mathcal{P}_{\text{fin}}(\boxed{\Gamma})$	

The classic way to add recursive types would be to introduce new syntax like

Type variables:	$\alpha ::= \texttt{'a} \mid \texttt{'b} \mid \texttt{'c} \mid \cdots$	
Types:	$\tau ::= \cdots$	
	$\mid \alpha$	
	$\mid \mu\alpha.\tau$	

and proceed to consider the types "$\mu\alpha.\tau$" and "τ with every α replaced by $\mu\alpha.\tau$" to be either identical types or isomorphic types, according to one's purpose and preferences. However, we run into problems if we try to do this with subplacing and bivariant (or contravariant) types around: What should the subplacing rule for $\mu\alpha.\tau$ be? And the one for α – if any at all?

The standard answers to these questions are given by Amadio and Cardelli [1993]. They consider recursive types with the $\mu\alpha.\tau$ syntax and define a subtyping relation by what is essentially a co-inductive definition[4]. However, they also present a concrete algorithm for deciding the subtyping relation, which consider the recursive types as *graphs* obtained by replacing each μ construction with back edges in the type's abstract syntax tree. In general, I find that viewing the recursive type as a graph is much closer to implementation intuition than the textual μ notation, so we shall build our formal support for recursive types upon that idea.

Thus, we introduce an indirection layer in the syntax, replacing types with indirect links to types via a type graph:

Type links:	$\pi ::= \texttt{t0} \mid \texttt{t1} \mid \texttt{t2} \mid \cdots$
Cell types:	$\theta ::= \nu{:}\pi$

[3] Certainly I myself am often guilty of that, and it seems that I am not alone: Many papers on type theory that purport to be applicable to real-world programming languages actually leave the addition of type recursion as an exercise for the reader, perhaps without saying it in so many words.

[4] See Brandt and Henglein [1998] for an explicitly co-inductive formulation of the Amadio–Cardelli rules.

$$
\begin{array}{lll}
\text{Types:} & \tau ::= \{w\} & (w \in \mathbb{W} \setminus \mathbb{A}) \\
& \quad | \ \langle \theta_{11}, ..., \theta_{1k_1} \mid \cdots \mid \theta_{n1}, ..., \theta_{nk_n} \rangle_p & (n \geq 1, k_i \geq 0) \\
\text{Environments:} & \Gamma \in \boxed{x} \xrightarrow{\text{fin}} \boxed{\pi} & \\
\text{Type graphs:} & \Pi \in \boxed{\pi} \xrightarrow{\text{fin}} \boxed{\tau} & \\
\text{Typings:} & \mathcal{T} \in \boxed{s} \xrightarrow{\text{fin}} \mathcal{P}_{\text{fin}}(\boxed{\Pi} \times \boxed{\Gamma}) &
\end{array}
$$

(It would perhaps have been more conventional to let each $\Gamma(x)$ be a pair of a graph and a distinguished link, but it will allow us to state typing rules more compactly if we let all links from one Γ refer to the same graph).

5.2 The full region type system

Definition 5.1. *The syntax of the region type system is as follows:*

$$
\begin{array}{lll}
\text{Places:} & p ::= \bot \mid \rho \mid \top & \\
\text{Type links:} & \pi ::= \texttt{t0} \mid \texttt{t1} \mid \texttt{t2} \mid \cdots & \\
\text{Permissions:} & \nu ::= \texttt{rw} \mid \texttt{ro} & \\
\text{Cell types:} & \theta ::= \nu{:}\pi & \\
\text{Types:} & \tau ::= \{w\} & (w \in \mathbb{W} \setminus \mathbb{A}) \\
& \quad | \ \langle \theta_{11}, ..., \theta_{1k_1} \mid \cdots \mid \theta_{n1}, ..., \theta_{nk_n} \rangle_p & (n \geq 1, k_i \geq 0) \\
\text{Environments:} & \Gamma \in \boxed{x} \xrightarrow{\text{fin}} \boxed{\pi} & \\
\text{Type graphs:} & \Pi \in \boxed{\pi} \xrightarrow{\text{fin}} \boxed{\tau} & \\
\text{Typings:} & \mathcal{T} \in \boxed{s} \xrightarrow{\text{fin}} \mathcal{P}_{\text{fin}}(\boxed{\Pi} \times \boxed{\Gamma}) &
\end{array}
$$

Definition 5.2. *The type graph Π is **closed** iff, for every $\langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots \rangle_p \in \operatorname{Img} \Pi$, each θ_{ij} is $\nu{:}\pi$ with $\pi \in \operatorname{Dom} \Pi$.*

We will assume implicitly that all the graphs we manipulate in the following are closed.

Definition 5.3. *Every (total or partial) map $f : \boxed{\rho} \to \boxed{p}$ extends naturally to a (total or partial) map $f^* : \boxed{\tau} \to \boxed{\tau}$:*

$$
\begin{array}{rcl}
f^*(\{w\}) & = & \{w\} \\
f^*(\langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots \rangle_\top) & = & \langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots \rangle_\top \\
f^*(\langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots \rangle_\rho) & = & \langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots \rangle_{f(\rho)} \\
f^*(\langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots \rangle_\bot) & = & \langle \cdots \mid ..., \theta_{ij}, ... \mid \cdots \rangle_\bot
\end{array}
$$

Notation 5.4. *The notation "$\Pi[\rho_1 \mapsto \rho_2]$" is an abbreviation for "$\left(Id \circledast \left[\begin{smallmatrix} \rho_1 \\ \rho_2 \end{smallmatrix}\right]\right)^* \circ \Pi$", where Id is the identity map on $\boxed{\rho}$.*

Notation 5.5. *The **free region variables** of a type graph Π are*

$$\operatorname{frv} \Pi = \left\{ \rho \;\middle|\; \langle ... \rangle_\rho \in \operatorname{Img} \Pi \right\}$$

The following series of definitions serve to define a coinductive subplacing relation between graphs:

Definition 5.6. *Let* $\mathbb{T}$ *be the set of pairs of closed type graphs and links into them:*

$$\mathbb{T} = \left\{ (\Pi, \pi) \;\middle|\; \Pi \text{ is closed}, \pi \in \operatorname{Dom}\Pi \right\}$$

Definition 5.7. *The relation* $\mathtt{R}$ *in* $\mathbb{T}$ *is a* ***type matching certificate*** *iff each time* $(\Pi, \pi)\ \mathtt{R}\ (\Pi', \pi')$*, one of the following hold:*

a. $\Pi(\pi) = \Pi'(\pi') = \{w\}$.

b. $\Pi(\pi) = \langle \cdots \mid ..., \nu_{ij}{:}\pi_{ij}, ... \mid \cdots \rangle_p$ *and* $\Pi'(\pi') = \langle \cdots \mid ..., \nu'_{ij}{:}\pi'_{ij}, ... \mid \cdots \rangle_p$ *with the same number of choices and cells in each case, such that* $\nu_{ij} = \nu'_{ij}$ *and* $(\Pi, \pi_{ij})\ \mathtt{R}\ (\Pi', \pi'_{ij})$ *for all* i, j.

The ***type matching relation*** $\equiv$ *is the union of all type matching certificates.*

One easily sees that $\equiv$ is indeed a type matching certificate itself, and that it is the greatest type matching certificate. $\equiv$ can be decided by trying to compute the *least* type matching relation that includes a given pair $((\Pi, \pi), (\Pi', \pi'))$. All pairs produced during this process will share the same Π and Π', and because those two graphs are finite, the process will terminate with either a finite certificate or a trace that shows that no certificate can include $((\Pi, \pi), (\Pi', \pi'))$.

Fact 5.8. $\equiv$ *is reflexive, symmetric, and transitive, i.e., an equivalence relation.*

Fact 5.9. *If* Π *is closed, then* $(\Pi \oplus \Pi', \pi) \equiv (\Pi, \pi)$ *whenever* $\pi \in \operatorname{Dom}\Pi$ *and* $\Pi \oplus \Pi'$ *is defined.*

Definition 5.10. *The relation* $\mathtt{R}$ *in* $\mathbb{T}$ *is a* ***subplacing certificate*** *iff each time* $(\Pi, \pi)\ \mathtt{R}\ (\Pi', \pi')$*, one of the following hold:*

a. $\Pi(\pi) = \Pi'(\pi') = \{w\}$.

b. $\Pi(\pi) = \langle \cdots \mid ..., \nu_{ij}{:}\pi_{ij}, ... \mid \cdots \rangle_p$ *and* $\Pi'(\pi') = \langle \cdots \mid ..., \nu'_{ij}{:}\pi'_{ij}, ... \mid \cdots \rangle_{p'}$ *with the same number of choices and cells in each case, such that*

1. *Either* $p = \bot$ *or* $p = p'$ *or* $p' = \top$.
2. *Whenever* $\nu'_{ij} = \mathtt{rw}$*, it holds that* $\nu_{ij} = \mathtt{rw}$ *and* $(\Pi, \pi_{ij}) \equiv (\Pi', \pi'_{ij})$.
3. *Whenever* $\nu'_{ij} = \mathtt{ro}$*, it holds that* $(\Pi, \pi_{ij})\ \mathtt{R}\ (\Pi', \pi'_{ij})$.

The ***subplacing relation*** $\leqslant$ *is the union of all subplacing certificates.*

Fact 5.11. *A type matching certificate is also a subplacing certificate. Therefore* $(\equiv) \subseteq (\leqslant)$.

Fact 5.12. $\leqslant$ *is reflexive and transitive, i.e., a preorder.*

Definition 5.13. *Define the relations* $\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Imp}$ *and* $\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Mop}$ *by the following rules:*

$$\frac{(\Pi, \Gamma) \in \mathcal{T}(s)}{\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathsf{goto}\, s}\ \textit{RTGOTO}$$

$$\frac{\operatorname{Dom}\Gamma = \operatorname{Dom}\Gamma' \qquad \begin{array}{c}\mathcal{T} \vdash \{\Pi', \Gamma'\}\ \mathfrak{Imp} \\ \forall x \in \operatorname{Dom}\Gamma : (\Pi, \Gamma(x)) \leqslant (\Pi', \Gamma'(x))\end{array}}{\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Imp}}\ \textit{RTSUB}$$

$$\frac{\begin{array}{l}\Pi(\pi) = \langle \theta_{11}, ..., \theta_{1k_1} \mid \cdots \mid \theta_{n1}, ..., \theta_{nk_n} \rangle_p \\ \forall i \le n : \begin{cases} \mathcal{T} \vdash \{\Pi, \Gamma \oplus [\frac{x}{\pi_i}]\}\ \mathfrak{Imp} \\ \quad \text{where } \Pi(\pi_i) = \langle \theta_{i1}, ..., \theta_{ik_i} \rangle_p \end{cases}\end{array}}{\mathcal{T} \vdash \{\Pi, \Gamma \oplus [\frac{x}{\pi}]\}\ \mathfrak{Imp}}\ \textit{RTEXPLODE}$$

$$\frac{\begin{array}{c}\Pi(\pi) = \langle \theta_1, ..., \theta_n \rangle_p \\ \mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x \\ \pi' \end{smallmatrix}\right]\right\} \mathfrak{Imp} \\ \Pi(\pi') = \langle \cdots \mid \theta_1, ..., \theta_n \mid \cdots \rangle_p\end{array}}{\mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x \\ \pi \end{smallmatrix}\right]\right\} \mathfrak{Imp}} \textit{RTEXTEND}$$

$$\frac{\rho^c \notin \operatorname{frv}\Pi \qquad \mathcal{T} \vdash \{\Pi, \Gamma\}\, \mathfrak{Imp}}{\mathcal{T} \vdash \{\Pi, \Gamma\}\, \mathsf{new}\ \rho^c; \mathfrak{Imp}} \textit{RTNEW}$$

$$\frac{\mathcal{T} \vdash \{\Pi, \Gamma\}\, \mathfrak{Imp}}{\mathcal{T} \vdash \{\Pi[\rho^c \mapsto \rho], \Gamma\}\, \mathsf{alias}\ \rho\ \mathsf{to}\ \rho^c; \mathfrak{Imp}} \textit{RTALIAS}$$

$$\frac{\rho^c \notin \operatorname{frv}\Pi \qquad \mathcal{T} \vdash \{\Pi, \Gamma\}\, \mathfrak{Imp}}{\mathcal{T} \vdash \{\Pi, \Gamma\}\, \mathsf{release}\ \rho^c; \mathfrak{Imp}} \textit{RTRELEASE}$$

$$\frac{\rho_2^c \notin \operatorname{frv}\Pi \qquad \mathcal{T} \vdash \{\Pi[\rho_1^c \mapsto \rho_2^c], \Gamma\}\, \mathfrak{Mop}}{\mathcal{T} \vdash \{\Pi, \Gamma\}\, \mathsf{rename}\ \rho_1^c\ \mathsf{to}\ \rho_2^c} \textit{RTRENAME}$$

$$\frac{\Pi(\pi) = \langle \nu_0{:}\pi_0, ..., \nu_k{:}\pi_k \rangle_\rho \qquad \mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x \\ \pi \end{smallmatrix}\right]\right\} \mathfrak{Imp}}{\mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x_0 & \cdots & x_k \\ \pi_0 & \cdots & \pi_k \end{smallmatrix}\right]\right\}\ x\ :=\ \mathsf{cons}\ \langle x_0, ..., x_k \rangle\ \mathsf{at}\ \rho; \mathfrak{Imp}} \textit{RTCONS}$$

$$\frac{\Pi(\pi) = \langle \nu_0{:}\pi_0, ..., \nu_k{:}\pi_k \rangle_\top \qquad \mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x \\ \pi \end{smallmatrix}\right]\right\} \mathfrak{Imp}}{\mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x_0 & \cdots & x_k \\ \pi_0 & \cdots & \pi_k \end{smallmatrix}\right]\right\}\ x\ :=\ \mathsf{cons}\ \langle x_0, ..., x_k \rangle\ \mathsf{nowhere}; \mathfrak{Imp}} \textit{RTNOWHERE}$$

$$\frac{\begin{array}{c}\Pi(\pi) = \langle \nu{:}\pi', \theta_1 ..., \theta_1 \rangle_p \qquad p \neq \top \\ \mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x' & x \\ \pi' & \pi'' \end{smallmatrix}\right]\right\} \mathfrak{Imp} \qquad \Pi(\pi'') = \langle \theta_1, ..., \theta_k \rangle_p\end{array}}{\mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x \\ \pi \end{smallmatrix}\right]\right\}\ x'\ :=\ \mathcal{H}[x{+}{+}]; \mathfrak{Imp}} \textit{RTREAD}$$

$$\frac{\begin{array}{c}\Pi(\pi) = \langle \mathtt{rw}{:}\pi', \theta_1, ..., \theta_k \rangle_p \qquad p \neq \top \\ \mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x \\ \pi'' \end{smallmatrix}\right]\right\} \mathfrak{Imp} \qquad \Pi(\pi'') = \langle \theta_1, ..., \theta_k \rangle_p\end{array}}{\mathcal{T} \vdash \left\{\Pi, \Gamma \oplus \left[\begin{smallmatrix} x' & x \\ \pi' & \pi \end{smallmatrix}\right]\right\}\ \mathcal{H}[x{+}{+}]\ :=\ x'; \mathfrak{Imp}} \textit{RTWRITE}$$

$$\frac{\begin{array}{rl} & \Pi_1 \text{ is closed} \qquad \operatorname{Img}\Gamma_1 \subseteq \operatorname{Dom}\Pi_1 \\ \Pi_1 = (\mathfrak{Ra} \oplus \mathfrak{Ua})^* \circ \Pi_2 & \Gamma_2 = \Gamma_1 \leftarrow \mathfrak{Ma} \qquad (\Pi_2, \Gamma_2) \in \mathcal{T}(s_2) \\ \operatorname{frv}\Pi \cap \operatorname{Img}\mathfrak{Ra} = \varnothing & \Pi \text{ is closed} \qquad \operatorname{Img}\Gamma \subseteq \operatorname{Dom}\Pi \\ \forall s_3 \in \operatorname{Dom}\mathfrak{Rtn}, (\Pi_3, \Gamma_3) \in \mathcal{T}(s_3): & \left\{\begin{array}{l} \operatorname{frv}\Pi \cap \operatorname{Img}\mathfrak{Ra}' = \varnothing \\ \mathcal{T} \vdash \{\Pi \oplus \Pi_4, \Gamma \oplus \Gamma_4\}\, \mathfrak{Imp} \\ \quad \text{where } \Pi_4 = (\mathfrak{Ra}' \oplus \mathfrak{Ua})^* \circ \Pi_3 \\ \qquad\quad \Gamma_3 = \Gamma_4 \leftarrow \mathfrak{Ma}' \\ \qquad\quad (\mathfrak{Ra}', \mathfrak{Ma}', \mathfrak{Imp}) = \mathfrak{Rtn}(s_3) \end{array}\right.\end{array}}{\mathcal{T} \vdash \{\Pi \oplus \Pi_1, \Gamma \oplus \Gamma_1\}\, \mathsf{call}\ s_2(\mathfrak{Ra}, \mathfrak{Ua}, \mathfrak{Ma});\ \mathsf{then}\ \mathfrak{Rtn}} \textit{RTCALL}$$

$$\frac{}{\mathcal{T} \vdash \{\Pi, \Gamma\}\, \mathsf{end}} \textit{RTEND}$$

The "*RT*" prefix stands for "Region Typing".

Note that, in *RTCALL*, the equation $\Pi_1 = (\mathfrak{Ra} \oplus \mathfrak{Ua})^* \circ \Pi_2$ implies that the free region variables of Π_2 all are in $\operatorname{Dom}\mathfrak{Ra}$ or $\operatorname{Dom}\mathfrak{Ua}$.

There is no general *RTMISC* rule, because the full generality of the misc con-

struction is not meant to be used all at once by the UHL generator. Instead, we expect that suitable rules are specified along with the translation from the host implementation's intermediate language to UHL. Of course, these must be compatible with the rest of the type system:

Definition 5.14. *The rule*

$$\frac{\begin{array}{ccc}\mathcal{T} \vdash \{\Pi, \Gamma_{11}\}\ \mathfrak{Imp}_1 & \cdots & \mathcal{T} \vdash \{\Pi, \Gamma_{1n_1}\}\ \mathfrak{Imp}_1 \\ \vdots & & \vdots \\ \mathcal{T} \vdash \{\Pi, \Gamma_{k1}\}\ \mathfrak{Imp}_k & \cdots & \mathcal{T} \vdash \{\Pi, \Gamma_{kn_k}\}\ \mathfrak{Imp}_k\end{array}}{\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathsf{misc}\ \mathfrak{Gs}_1; \mathfrak{Imp}_1 \,\square\, \cdots \,\square\, \mathfrak{Gs}_k; \mathfrak{Imp}_k}$$

is ***valid*** *if for each* i *and for each* $\sigma \xrightarrow{\wp} \sigma' \in \mathfrak{Gs}_i$*, there is a* j *such that for each* $x' \in \mathrm{Dom}\,\sigma'$*, one of the following holds:*

1. *There is an* $x \in \mathrm{Dom}\,\sigma$ *such that* $\Gamma(x) = \Gamma_{ij}(x')$ *and* $\sigma(x) = \sigma'(x')$*.*
2. $\Pi(\Gamma_{ij}(x'))$ *is not* $\{w\}$ *and* $\sigma'(x') \notin \mathbb{A}$*.*
3. $\Pi(\Gamma_{ij}(x')) = \{w\}$ *and* $\sigma'(x') = w$*.*
4. *There is an* $x \in \mathrm{Dom}\,\sigma$ *such that* $\Pi(\Gamma(x)) = \{w\}$ *and* $w \neq \sigma(x)$*.*

Definition 5.15. *The annotated mutator* $\mathcal{A}$ *is* ***well-typed*** *by* $\mathcal{T}$ *iff* $(\Pi_0, []) \in \mathcal{T}(s0)$ *for some* Π_0*, and each time* $(\Pi, \Gamma) \in \mathcal{T}(s)$ *it holds that* $\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathcal{A}(s)$ *can be derived by the rules of Definition 5.13 plus perhaps a valid rule according to Definition 5.14.*

5.3 Safety proof for the region type system

In this section we sketch a proof of

Theorem 5.16. *Let* $\mathcal{A}$ *be an annotated uniform mutator. If* $\mathcal{A}$ *is well-typed, then* $\mathcal{A}$ *is region safe.*

The details of the proof will not be used afterwards; readers who are willing to accept my word that the theorem is true can skip directly to the next section (or, indeed, to the next chapter).

Superficially, Theorem 5.16 is not much more than what was proved by Niss [2002, Chapter 4], but the proofwork we need here is more intricate, because we have to handle explicitly recursive types and circular values (which can be produced by destructive update). For this, we need to work with *store typings* (or a coinductive type interpretation, but especially in presence of destructive update, a store typing is easier to work with). A store typing may need to refer to regions for which no name is currently in scope (a reference to the region may be stored somewhere on the stack), so it will be convenient to work with region *names* r rather than region *variables* ρ.

Notation 5.17. *The variables* $\tilde{p}$*,* $\tilde{\tau}$*, and* $\tilde{\Pi}$ *range over places* p*, types* τ*, and type graphs* Π *in which all region variables* ρ *have been replaced by region names* r*. We call these "tilded" entities* ***runtime*** *places, types, etc.*

Definitions 5.3 through 5.10 carry over to runtime entities without change. Also, we can move between the static and runtime universes by using Definition 5.3 to extend a map $f : \boxed{\rho} \to \boxed{\tilde{p}}$ to $f^* : \boxed{\tau} \to \boxed{\tilde{\tau}}$.

Lemma 5.18. *Let* f *be any (total) map* $\boxed{\rho} \to \boxed{\tilde{p}}$*. If* $(\Pi, \pi) \leqslant (\Pi', \pi')$ *(where* $\leqslant$ *is the ordinary subplacing relation between static types from Definition 5.10), then* $(f^* \circ \Pi, \pi) \leqslant (f^* \circ \Pi', \pi')$ *(which is a subplacing relation between runtime graphs and links).*

Proof. It is easily seen that $(\Pi, \pi) \equiv (\Pi', \pi')$ implies $(f^* \circ \Pi, \pi) \equiv (f^* \circ \Pi', \pi')$ – the static type matching relation $\equiv$ becomes a runtime type matching certificate when f^* is concatenated onto the graph parts of all the relation pairs.

It is less obvious that the same technique works for $\leqslant$, because f may map some region variables to $\top$ or $\bot$. On further inspection, however, what we need to prove is just

$$p = \bot \vee p = p' \vee p' = \top \;\Rightarrow\; f'(p) = \bot \vee f'(p) = f'(p') \vee f'(p') = \top$$

where $f' = f \oplus \left[\begin{smallmatrix}\top & \bot\\ \top & \bot\end{smallmatrix}\right]$. But clearly each of the disjuncts on the left-hand side imply the corresponding disjunct on the right-hand side, so the entire implication is true. Therefore $\leqslant$ with f^* applied everywhere is indeed a subplacing certificate, and the lemma is true. □

Definition 5.19. *A* ***store typing*** Θ *is a finite map from* $\mathbb{A}$ *to* $\tilde{\mathbb{T}} = \boxed{\tilde{\Pi}} \times \boxed{\pi}$.

The role of a store typing is to remember which (runtime) type the value in each heap cell had when that cell was *allocated*. This can be used to define when some value w has the (runtime) type described by $(\tilde{\Pi}, \pi)$:

Definition 5.20. *The* ***value consistency*** *relation* $L, \Theta \models w : \tilde{\Pi}, \pi$ *is defined by*

$$\frac{\tilde{\Pi}(\pi) = \{w\}}{L, \Theta \models w : \tilde{\Pi}, \pi}\ \textit{TISINGLE} \qquad \frac{w \notin \mathbb{A} \qquad \tilde{\Pi}(\pi) = \langle \cdots \rangle_{\tilde{p}}}{L, \Theta \models w : \tilde{\Pi}, \pi}\ \textit{TIBASE} \qquad \frac{\tilde{\Pi}(\pi) = \langle \cdots \rangle_{\top}}{L, \Theta \models w : \tilde{\Pi}, \pi}\ \textit{TITOP}$$

$$\frac{\begin{array}{l}\tilde{\Pi}(\pi) = \langle \theta_{10}, ..., \theta_{1k_n} \mid \cdots \mid \theta_{n0}, ..., \theta_{nk_n} \rangle_r \qquad L(r) = (m, A)\\ \exists i : \forall j : 0 \le j \le k_i \Rightarrow \begin{cases} \{a + j\} \in A \\ \theta_{ij} = \mathtt{ro}{:}\pi' \Rightarrow \Theta(a + j) \leqslant (\tilde{\Pi}, \pi') \\ \theta_{ij} = \mathtt{rw}{:}\pi' \Rightarrow \Theta(a + j) \equiv (\tilde{\Pi}, \pi') \end{cases}\end{array}}{L, \Theta \models a : \tilde{\Pi}, \pi}\ \textit{TIRGN}$$

Rule *TIRGN* contains the runtime invariant that describes the difference between read-only and read-write cells: A read-write type description must match the cell's original type exactly. So, intuitively, we have the invariant that (abusing notation slightly) $L, \Theta \models a : \langle \mathtt{ro}{:}\tau_1 \rangle_r$ and $L, \Theta \models a : \langle \mathtt{rw}{:}\tau_2 \rangle_r$ together imply $\tau_2 \leqslant \tau_1$, so it is safe to destructively write a τ_2 to the cell and later expect to read a τ_1.

We consider the rule

$$\frac{\tilde{\Pi}(\pi) = \langle \rangle_r \qquad L(r) = (m, A)}{L, \Theta \models a : \tilde{\Pi}, \pi}$$

to be a special case of *TIRGN* with $n = 1$ and $k_1 = -1$ – then "$\forall j$" collapses to a tautology.

Fact 5.21. *If* $L, \Theta \models w : \tilde{\Pi}, \pi$ *and* $(\tilde{\Pi}, \pi) \leqslant (\tilde{\Pi}', \pi')$*, then* $L, \Theta \models w : \tilde{\Pi}', \pi'$.

We can extend this typing relation to entire data states and region environments, closing the link between static and runtime types:

Definition 5.22. *The **data state consistency** relation* $L, \Theta \models R, \sigma : \Pi, \Gamma$ *is defined by*

$$\frac{\mathrm{Dom}\,\sigma = \mathrm{Dom}\,\Gamma \qquad \forall x \in \mathrm{Dom}\,\sigma : L, \Theta \models \sigma(x) : (\mathit{Bot} \circledcirc R)^* \circ \Pi, \Gamma(x)}{L, \Theta \models R, \sigma : \Pi, \Gamma}\ \textit{TIENV}$$

where Bot is the map that maps every region name to $\bot$*.*

Proposition 5.23. *If* $L, \Theta \models R, \sigma : \Pi, \Gamma$ *and* $(\Pi, \Gamma(x)) \leqslant (\Pi', \Gamma'(x))$ *for all* $x \in \mathrm{Dom}\,\Gamma = \mathrm{Dom}\,\Gamma'$*, then* $L, \Theta \models R, \sigma : \Pi', \Gamma'$*.*

Proof. By Lemma 5.18 we have $((\mathit{Bot} \circledcirc R)^* \circ \Pi, \Gamma(x)) \leqslant ((\mathit{Bot} \circledcirc R)^* \circ \Pi', \Gamma'(x))$ for each x. The proposition now follows from applying Fact 5.21 for each x. □

Of course, we also need to be able to express that the actual heap agrees with that the store typing says:

Definition 5.24. *The **heap consistency** relation* $L \models H : \Theta$ *is defined by*

$$\frac{\forall a \in \mathrm{Dom}\,\Theta : L, \Theta \models H(a) : \Theta(a)}{L \models H : \Theta}$$

Proposition 5.25. *Assume* $L \models H : \Theta$*. If* $L = L' \oplus \{\not\mapsto (n, A)\}$ *, then there is a* Θ' *such that* $L' \models H|_{L'} : \Theta'$ *and whenever* $L, \Theta \models R, \sigma : \Pi, \Gamma$ *with* $r \notin \mathrm{Img}\,R$*, then* $L', \Theta' \models R, \sigma : \Pi, \Gamma$*.*

This proposition states that it is possible to maintain the store typing when a region is destroyed, provided no region variable is bound to it. The proof uses an auxiliary definition and a series of lemmas:

Definition 5.26. *Let* Θ *be a store typing and* r *be a region name. The store typing that corresponds to* Θ ***without*** r *is written* $\Theta \setminus r$ *and defined by*

$$\Theta(\pi) = (\tilde{\Pi}, \pi') \Longrightarrow (\Theta \setminus r)(\pi) = (\tilde{\Pi}[\not\mapsto \top], \pi')$$

Lemma 5.27. *If* $(\tilde{\Pi}, \pi) \equiv (\tilde{\Pi}', \pi')$ *then* $(\tilde{\Pi}[\not\mapsto \top], \pi) \equiv (\tilde{\Pi}'[\not\mapsto \top], \pi')$ *.*
If $(\tilde{\Pi}, \pi) \leqslant (\tilde{\Pi}', \pi')$ *then* $(\tilde{\Pi}[\not\mapsto \top], \pi) \leqslant (\tilde{\Pi}'[\not\mapsto \top], \pi')$ *.*

Proof. A type matching (or subplacing) certificate is still a type matching (or subplacing) certificate when the substitution $[\not\mapsto \top]$ is performed on the graph parts of all its elements. □

Note that the lemma could not be proved indirectly by high-level properties (such as transitivity) of the subplacing relation: it allows one to replace a region name by $\top$ *below* a `rw` mark, where it would not be allowed by subplacing. The reason why the lemma still holds is that the substitution must be performed on both sides of the relation.

Lemma 5.28. $L, \Theta \models w : \tilde{\Pi}, \pi$ *implies* $L, \Theta \setminus r \models w : \tilde{\Pi}[\not\mapsto \top], \pi$ *.*

Proof. By inspection of each of the rules in Definition 5.20. For *TIRGN*, use Lemma 5.27 for the subplacing premises. □

Corollary 5.29. $L \models H : \Theta$ *implies* $L \models H : \Theta \setminus r$*.*

Lemma 5.30. *If* $L_1 \oplus L_2, \Theta \models w : \tilde{\Pi}, \pi$ *and* $\mathrm{frv}\,\tilde{\Pi} \subseteq \mathrm{Dom}\,L_1$*, then* $L_1 \models w : \tilde{\Pi}, \pi$.

Proof. The only rule for $L, \Theta \models w : \tilde{\Pi}, \pi$ that depends on L is *TIRGN*. It only depends on the value of L at r's in $\mathrm{frv}\,\tilde{\Pi}$, which proves the lemma. □

Corollary 5.31. *Assume* $L_1 \oplus L_2 \models H : \Theta$ *and* $\mathrm{frv}\,\Theta \subseteq \mathrm{Dom}\,L_1$*. Then* $L_1 \models H : \Theta$.

Lemma 5.32. $L, \Theta \models w : \tilde{\Pi}, \pi$ *implies* $L, \Theta|_L \models w : \tilde{\Pi}, \pi$*, where* $\Theta|_L$ *is* Θ *restricted to* L*'s footprint, analogous to Definition 3.18.*

Proof. The only rule for $L, \Theta \models w : \tilde{\Pi}, \pi$ that depends on Θ is *TIRGN*. It only depends on the value of Θ at addresses that are explicitly checked to be in L's footprint. □

Corollary 5.33. *Assume* $L \models H : \Theta$*. Then* $L \models H|_L : \Theta|_L$.

Proof of Proposition 5.25. Set $\Theta' = (\Theta \setminus r)|_{L'}$. By Corollary 5.29 we have $L \models H : \Theta\setminus r$. By definition $r \notin \mathrm{frv}(\Theta\setminus r)$, so Corollary 5.31 implies $L' \models H : \Theta\setminus r$. Now Corollary 5.33 gives us $L' \models H|_{L'} : \Theta'$. This proves the first part of the proposition.

Now assume $L, \Theta \models R, \sigma : \Pi, \Gamma$ with $r \notin \mathrm{Img}\,R$. We must prove that $L', \Theta' \models R, \sigma : \Pi, \Gamma$, that is, for each $x \in \mathrm{Dom}\,\Gamma$ that $L', \Theta' \models \sigma(x) : \tilde{\Pi}, \Gamma(x)$, where $\tilde{\Pi} = (\mathit{Bot} \circledcirc R)^* \circ \Pi$. It is clear that $\mathrm{frv}\,\tilde{\Pi} \subseteq \mathrm{Img}\,R$, so $r \notin \mathrm{frv}\,\tilde{\Pi}$. Therefore, $\tilde{\Pi}[\rho \mapsto \top] = \tilde{\Pi}$ and by Lemmas 5.28 and 5.30 we get $L, \Theta \setminus r \models \sigma(x) : \tilde{\Pi}, \Gamma(x)$ and $L', \Theta \setminus r \models \sigma(x) : \tilde{\Pi}, \Gamma(x)$. Finally, Lemma 5.32 gives $L', \Theta' \models \sigma(x) : \tilde{\Pi}, \Gamma(x)$, which completes the proof. □

Propositions 5.23 and 5.25 contain the main insights behind the proof of Theorem 5.16. The rest the proof is just about putting the pieces together in the right order.

In the proof we generally assume the $\mathcal{A}$ is well-typed by $\mathcal{T}$.

Lemma 5.34. *Assume* $\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Jmp}$ *and* $L, \Theta \models R, \sigma : \Pi, \Gamma$*. Then there are* Π' *and* Γ' *such that* $L, \Theta \models R, \sigma : \Pi', \Gamma'$ *and such that* $\mathcal{T} \vdash \{\Pi', \Gamma'\}\ \mathfrak{Jmp}$ *can be derived without using* *RTSUB*, *RTEXPLODE*, *or* *RTEXTEND* *as the last step.*

Proof. By induction on the derivation of $\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Jmp}$. Obviously, the only cases we need to consider are those for *RTSUB*, *RTEXPLODE*, and *RTEXTEND*. Proposition 5.23 supplies the induction step for *RTSUB*. The cases for *RTEXPLODE* and *RTEXTEND* are immediate, given the existential quantifier in *TIRGN*. □

Definition 5.35. *Define the relations* $\mathcal{T} \vdash \mathcal{C}$ *and* $\mathcal{T}, L, \Theta \vdash \omega \rhd R$ *by*

$$\frac{}{\mathcal{T} \vdash \mathsf{Stop}}\ \textit{RTCSTOP} \qquad \frac{L \models H : \Theta \qquad L, \Theta \models R, \sigma : \Pi, \Gamma \qquad \mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Jmp} \qquad \mathcal{T}, L, \Theta \vdash \omega \rhd R}{\mathcal{T} \vdash (H, L, R, \sigma, \mathfrak{Jmp}, \omega)}\ \textit{RTCSTD}$$

$$\frac{}{\mathcal{T}, L, \Theta \vdash \bullet \rhd R}\ \textit{RTSEMPTY}$$

$$\frac{\begin{array}{c}\mathsf{R} \circ \mathfrak{Ua} \subseteq \mathsf{R}' \qquad \mathcal{T}, \mathsf{L}, \Theta \vdash \omega \rhd \mathsf{R} \\ \mathsf{L}, \Theta \models \mathsf{R}, \sigma : \Pi, \Gamma \qquad \operatorname{frv}\Pi \subseteq \operatorname{Dom}\mathsf{R} \\ \forall s_3 \in \operatorname{Dom}\mathfrak{Rtn} : \\ \forall (\Pi_3, \Gamma_3) \in \mathcal{T}(s_3) : \end{array} \begin{cases} \mathcal{T} \vdash \{\Pi \oplus \Pi_4, \Gamma \oplus \Gamma_4\}\ \mathfrak{Jmp} \\ \quad \text{where } \Pi_4 = (\mathfrak{Ra} \oplus \mathfrak{Ua})^* \circ \Pi_3 \\ \qquad\quad \Gamma_3 = \Gamma_4 \circ \mathfrak{Ma} \\ \qquad\quad (\mathfrak{Ra}, \mathfrak{Ma}, \mathfrak{Jmp}) = \mathfrak{Rtn}(s_3) \end{cases}}{\mathcal{T}, \mathsf{L}, \Theta \vdash (\mathfrak{Rtn}, \mathsf{R}, \sigma) :: \omega \rhd \mathsf{R}'}\ \textit{RTSFRAME}$$

Proposition 5.36 (Subject reduction, level 2). *Assume $\mathcal{T} \vdash \mathcal{C}$ and $\mathcal{A} \overset{\Psi}{\twoheadrightarrow} \mathcal{C}$. If $\mathcal{C} \xrightarrow[\mathcal{A}]{\varphi} \mathcal{C}'$ or $\mathcal{C} \xrightarrow[\mathcal{A}]{} \mathcal{C}'$, then $\mathcal{D}, \mathcal{S}, \mathcal{A} \vdash \mathcal{C}'$.*

Proof. Because Stop has no successors, the derivation of $\mathcal{T} \vdash \mathcal{C}$ must be by *RTC-STD*, so $\mathcal{C}$ has the form $(\mathsf{H}, \mathsf{L}, \mathsf{R}, \sigma, \texttt{goto}\, s, \omega)$ and there exists Θ, Π, Γ such that $\mathsf{L} \models \mathsf{H} : \Theta$ and $\mathsf{L}, \Theta \models \mathsf{R}, \sigma : \Pi, \Gamma$.

We also have $\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Jmp}$, and without loss of generality (Lemma 5.34) we can assume that the last rule for $\mathcal{T} \vdash \{\Pi, \Gamma\}\ \mathfrak{Jmp}$ is the *RTXXX* rule that corresponds to the syntax of $\mathfrak{Jmp}$. Now proceed by case analysis on the derivation of $\mathcal{C} \xrightarrow[\mathcal{A}]{(\varphi)} \mathcal{C}'$.

MXROP by *MXNEW*. The addition of a new region to ρ will never make any of the consistency relations become less true.

For the addition of a new region ρ^c to R, observe that *RTNEW* gives us $\rho^c \notin \operatorname{frv}\Pi$. Therefore $(\mathit{Bot} \circledcirc \mathsf{R})^* \circ \Pi = (\mathit{Bot} \circledcirc \mathsf{R} \circledcirc \left[\begin{smallmatrix}\rho^c \\ \mathsf{r}\end{smallmatrix}\right])^* \circ \Pi$, so $\mathsf{L}, \Theta \models \mathsf{R}, \sigma : \Pi, \Gamma$ keeps holding.

MXROP restricts H to the footprint of the new L, but because we assume $\mathcal{A} \overset{\Psi}{\twoheadrightarrow} \mathcal{C}$, Proposition 3.23(a) says that this is actually a no-op.

MXROP by *MXALIAS*. The increment of the region's reference count will not in itself make any of the consistency relations become less true (as *TIRGN* ignores the reference count).

RTALIAS says that we have $\Pi = \Pi'[\rho^c \mapsto \rho]$ where Π' is the new type graph. It is evident that

$$\left(\mathit{Bot} \circledcirc (\mathsf{R} \oplus \left[\begin{smallmatrix}\rho \\ \mathsf{r}\end{smallmatrix}\right])\right)^* \circ \Pi'[\rho^c \mapsto \rho] = \left(\mathit{Bot} \circledcirc (\mathsf{R} \oplus \left[\begin{smallmatrix}\rho & \rho^c \\ \mathsf{r} & \mathsf{r}\end{smallmatrix}\right])\right)^* \circ \Pi',$$

so the data state consistency relation keeps holding.

Like for *MXNEW* we can ignore the restriction of H to $\mathsf{H}|_{\mathsf{L}}$.

MXROP by *MXRELEASEN*. Like for *MXALIAS*, the reference count manipulation in L does not concern the consistency relations. *RTRELEASE* gives us $\rho^c \notin \operatorname{frv}\Pi$, so as for *MXNEW* the addition or (in this case) removal of a binding for it in R does not make the data state inconsistent.

Because the region variable that is released is syntactically restricted, the values of R for *uncounted* region variables cannot change. This is important if the stack ω is not empty, because $\mathtt{rtsFrame}$ records the original original bindings of uncounted variables and requires them to keep holding.

Like for *MXNEW* we can ignore the restriction of H to $\mathsf{H}|_{\mathsf{L}}$.

MXROP by *MXRELEASE1*. Because $\rho^c \notin \operatorname{frv}\Pi$, this binding can be removed from R without harming the consistency of the data state. Lemma 3.25 now tells

us that this binding was the *only* binding of a counted variable to r either in R itself or within the stack ω. By Lemma 3.26 this implies that there can be no uncounted variables that refer to r either. Therefore we can use Proposition 5.25 to find a new Θ that will be valid in all the necessary consistency relations after the region has been deallocated from L and H.

MXROP by *MXRENAME*. Much like for *MXALIAS*. The side condition $\rho_2^c \notin \mathrm{frv}\,\Pi$ in *RTRENAME* guarantees that no name capture will happen.

MXMISC. $\mathfrak{Jmp}$ must be $\mathtt{goto}\,s$, so from *RTGOTO* we know that $(\Pi,\Gamma) \in \mathcal{T}(s)$. Therefore, and because $\mathcal{A}$ is assumed to be well-typed, there is a valid rule according to Definition 5.14 that applies to Π and Γ, giving a new Γ' while relating σ to the new σ' according to conditions (1) through (4) of Definition 5.14.

It is impossible that condition (4) is actually used in the chosen step; this would contradict that $L,\Theta \models R,\sigma : \Pi,\Gamma$. Therefore, for each $x' \in \mathrm{Dom}\,\sigma'$ one of conditions (1) through (3). In case (2) or (3) a derivation of $L,\Theta \models \sigma'(x') : (\mathit{Bot} \circledcirc R)^* \circ \Pi, \Gamma'(x')$ can be constructed from scratch using *TIBASE* or *TISINGLE*. In case (1) a derivation is already present in that of $L,\Theta \models R,\sigma : \Pi,\Gamma$. Altogether, we can prove $L,\Theta \models R,\sigma' : \Pi,\Gamma'$, which is what all need to complete an application of *RTCSTD*.

MXCONS. $\mathfrak{Jmp}$ must be $\mathtt{goto}\,s$, so from *RTGOTO* we know that $(\Pi,\Gamma) \in \mathcal{T}(s)$. Therefore, and because $\mathcal{A}$ is assumed to be well-typed, we know that $\mathcal{T} \vdash \{\Pi,\Gamma\}\,\mathcal{A}(s)$ by *RTCONS*.

None of the consistency relations are affected by the fact that some new addresses are added to the appropriate part of L. When adding the cells to the heap, we must simultaneously extend Θ with $\{a \mapsto (\tilde{\Pi}, \pi_0), \ldots, a+k \mapsto (\tilde{\Pi}, \pi_k)\}$, where $\tilde{\Pi} = (\mathit{Bot} \circledcirc R)^* \circ \Pi$. Then $L' \models H' : \Theta'$ will still hold; the cases for the new cells come directly from the derivation of $L,\Theta \models R,\sigma : \Pi,\Gamma$.

This new Θ' allows a derivation of $L',\Theta' \models a : \pi$, so the new state is still consistent.

MXNOWHERE. As for *MXCONS*, we can deduce that *RTNOWHERE* must apply.

Because $\Pi(\pi) = \langle\cdots\rangle_\top$, rule *TITOP* allows us to derive $L,\Theta \models w : (\mathit{Bot} \circledcirc R)^* \circ \Pi, \pi$ which is all that is necessary for consistency of the new state.

MXREAD. As for *MXCONS*, we can deduce that *RTREAD* must apply.

Because $p \neq \top$, the consistency of the initial value of the pointer must be derived by either *TIBASE* or *TIRGN*. The former requires that $\sigma(x) \notin \mathbb{A}$ and thus contradicts *MXREAD* itself. So it must be *TIRGN*. From this, combined with Definition 5.24 and Fact 5.21 we get that the value read from the first heap cell has a typing consistent with Π and π'. We also get enough values of Θ to prove that the incremented pointer has the type predicted by *RTREAD*.

MXREADWRONG. As for *MXREAD* we deduce that consistency of $\sigma(x) = a$ must be derived by *TIRGN* with $n = 1$. Rule *RTREAD* says that the type is not $\langle\rangle_r$, so the bracket in *TIRGN* must hold for $j = 0$. Therefore there is $L(r) = (m, A)$ with $a + 0 \in A$. That is, a is in the footprint of L; by Proposition 3.23(a) a is also in $\mathrm{Dom}\,H$, which contradicts *MXREADWRONG*. So this case is impossible

MXWRITE. The mechanics of this case is much like that for *MXREAD*. The difference is that the value is written into the heap instead of read from it. *RTWRITE* guarantees that its description in the pointer's type is marked `rw`. Then get $\Theta(\mathfrak{a}) \equiv (\tilde{\Pi}, \pi')$ from *TIRGN*, use Fact 5.11 to derive $(\tilde{\Pi}, \pi') \leqslant \Theta(\mathfrak{a})$, and then Fact 5.21 to show that the value of the heap cell can be replaced without harming the consistency of the heap.

MXWRITEWRONG. Impossible, with the same argument as for *MXREADWRONG*.

MXCALL. There is a lot of uninteresting plumbing to take care of here. The general plan is that *MXCALL* and *RTCALL* together contains enough restrictions to construct instances of *RTSFRAME* and *RTCSTD* for the new state.

One slightly nontrivial point here is that each side of the split between parameter and caller-local data states, region environments, type graphs and type environments must still be consistent in itself. The important fact is that each of the two type graphs are closed. Then Facts 5.9 and 5.21 can be combined to see that every value in one of the two data states is still consistent with its own part of the type graph.

MXRETURN. Again, there is a large amount of uninteresting matching of symbols to be done here. The crucial point is showing that

$$L, \Theta \models R'', \sigma'' : \Pi \oplus \Pi_4, \Gamma \oplus \Gamma_4$$

where R'' and σ'' come from *MXRETURN* and Π, Π_4, Γ, Γ_4 come from *RTSFRAME*. Set $\Pi'' = \Pi \oplus \Pi_4$ and

$$\tilde{\Pi} = (\mathit{Bot} \circledcirc R'')^* \circ \Pi'' = \big((\mathit{Bot} \circledcirc R'')^* \circ \Pi\big) \oplus \big((\mathit{Bot} \circledcirc R'')^* \circ \Pi_4\big)$$

Because $\operatorname{frv}\Pi \subseteq \operatorname{Dom} R$ we have

$$(\mathit{Bot} \circledcirc R'')^* \circ \Pi = \big(\mathit{Bot} \circledcirc (R \oplus (R'_c \mathbin{\Leftarrow} \mathfrak{Ra}^{-1}))\big)^* \circ \Pi = (\mathit{Bot} \circledcirc R)^* \circ \Pi$$

so $L, \Theta \models R, \sigma : \Pi, \Gamma$ (which we know from *RTSFRAME*) implies $L, \Theta \models R'', \sigma : \Pi'', \Gamma$.

Further, we have

$$\begin{aligned}(\mathit{Bot} \circledcirc R'')^* \circ \Pi_4 &= (\mathit{Bot} \circledcirc R'')^* \circ (\mathfrak{Ra} \oplus \mathfrak{La})^* \circ \Pi_3 \\ &= \big(\mathit{Bot} \circledcirc (R'' \circ (\mathfrak{Ra} \oplus \mathfrak{La}))\big)^* \circ \Pi_3 \\ &= \big(\mathit{Bot} \circledcirc ((R'' \circ \mathfrak{Ra}) \oplus (R'' \circ \mathfrak{La}))\big)^* \circ \Pi_3\end{aligned}$$

But

$$R'' \circ \mathfrak{Ra} = \big(R \oplus (R'_c \mathbin{\Leftarrow} \mathfrak{Ra}^{-1})\big) \circ \mathfrak{Ra} = R'_c \subseteq R'$$

and

$$R'' \circ \mathfrak{La} = \big(R \oplus (R'_c \mathbin{\Leftarrow} \mathfrak{Ra}^{-1})\big) \circ \mathfrak{La} = R \circ \mathfrak{La} \subseteq R'$$

(where the last inequality comes from *RTSFRAME*). Because Π_4 is defined, we have $\operatorname{frv}\Pi_3 \subseteq \operatorname{Dom}\mathfrak{Ra} \cup \operatorname{Dom}\mathfrak{La}$, so

$$(\mathit{Bot} \circledcirc R'')^* \circ \Pi_4 = (\mathit{Bot} \circledcirc R')^* \circ \Pi_4$$

Therefore $L, \Theta \models R', \sigma' : \Pi_4, \Gamma_4$ (which we know from *RTCSTD*) implies $L, \Theta \models R'', \sigma' : \Pi_4, \Gamma_4$, and further $L, \Theta \models R'', \sigma' \mathbin{\Leftarrow} \mathfrak{Ma}^{-1} : \Pi'', \Gamma_4$ (because $\Gamma_3 = \Gamma_4 \mathbin{\Leftarrow} \mathfrak{Ma}^{-1}$).

All in all we get $L, \Theta \models R'', \sigma'' : \Pi'', \Gamma \oplus \Gamma_4$ as wanted.

MXSTOP. Trivial, by *RTCSTOP*. □

Proof of Theorem 5.16. From Proposition 5.36, an easy induction on $\twoheadrightarrow$ proves that $\mathcal{A} \xrightarrow{\Psi}\!\!\!\!\twoheadrightarrow \mathcal{C}$ implies $\mathcal{T} \vdash \mathcal{C}$. Because it is not true that $\mathcal{T} \vdash \mathsf{Wrong}$, the agent must be region safe. □

Chapter 6

Other host-language features

In this chapter we will briefly investigate how some common language features that are not found in the ML subset of Chapter 4 can be represented in the UHL model. We will also consider how some of them need special extensions to the region type system.

Due to time constraints (see the Preface), the examples are less fleshed out than I had hoped to be able to, but the walkthrough nevertheless ought to give a feel for the range of possible UHL applications.

6.1 Generic imperative language features

The first set of features we will look at are the ones found in traditional structured imperative languages such as Pascal or C. From our prespective the most important property of these languages is actually the *absence* of function closures or polymorphic typing, which means that a lot of the shortcomings our techinques have for ML are not relevant in this context. But there are also a number of imperative language features that present challenges of their own:

6.1.1 Destructive update of heap cells

The UHL model includes a primitive write operation that destructively updates a heap cell. We have already shown how to use it for ML references (Section 4.4.2) and how it can be handled in a region type system (Section 5.1.6). Still, there are a few things to be said about destructive update in the context of imperative languages.

The first factor is that the *possibility* for destructive update is usually ubiquitous in an imperative language. Most imperative languages' default concept of a heap block is one in which the content can be changed at any time. If the programmer wishes to generate a block of which the language will prohibit updates, he has to ask for it specifically (for example, with a `const` keyword in C or C++). Compare this with ML's convention where heap allocation results in an immutable heap block unless you ask specifically for a reference. Therefore programmers in C-like languages often use a mutable heap block even if they have no intention of actually updating after it has been initialized – partly just to save the keystrokes of typing `const`; partly because C (and C++) makes it

difficult to work efficiently and flexibly with records that have the same layout except for differences in which fields it is allowed to update.

This means that if one tries to construct a region typing naively from the original C or C++ types, it is likely to end up with a lot of `rw` permissions and very few `ro`'s, if any. This will severely limit the possibility of subplacing, thus lowering the attainable agent quality. This effect can be counteracted if the region inference itself tries to change premissions from `rw` to `ro` whenever possible. An algorithm for this will be presented in Section 7.3.7.

The other problem with destructive update is that the *use* of it is ubiquitous in imperative programs. The naive rule for destructive update – that the value written into a heap cell must have the exact same region type as the previous value – is acceptable in a language such as ML that discourages widespread use of destructive update. But in an imperative language, where programmers often prefer updating all data fields in a stucture to building a new one, this tying-together of successive data value's lifetimes is likely to have disastrous consequences for many programs.

I believe that this is the largest single obstacle to widespread use of region-based memory management for imperative languages. Unfortunately I have no good idea of what a solution might look like. I have some hope that work like Reynolds' "separation logic" [Reynolds 2002] may eventually inspire a better solution, but so far nothing concrete has come of this.

6.1.2 Loops and gotos

Procedure bodies in imperative languages tend to have more complex control structures than ones in functional languages. They often contain loops, and in languages with `goto` statements they may not even have any hierarchical structure.

The UHL model supports such unstructured control flow by design. Moving away from the nice acyclic flowcharts that functional languages exhibit will need some extra algorithmic elbow grease in the region inference; this will be discussed – and solved – in (*i.a.*) Section 7.3.8.

Local loops make sense because imperative languages allow the values of local variables (such as a loop counter) to change while the procedure executes. This is also supported natively by the UHL model, even though our formal specification of the model requires the variable to be explicitly unbound before a new value is assigned to it. This will make the region type system see the assignment not as a new value in an old variable, but as a new variable that just happens to have a name that has been used before. The new variable can have its own region-annotated type that is unrelated to the previous type, so local assignmens do not tie the lifetimes of the old and new values together, unlike destructive updates on the heap.

6.1.3 Global variables

Imperative languages usually offer some concept of (statically allocated) global variables, if only in the form of "`static`" variables that are only visible in a small part of the program but exist for the duration of the program.

Ordinary global variables can be handled almost directly in the UHL model, by representing each of them as a UHL variable and formally thread its value in and out of procedure calls, utilizing the ability of UHL procedures to take any number of parameters and return any number of result values. This gives global variables the same ability as local ones to change region-annotated types when their values change.

In a large programs with many global variables, the threading of all the globals may be a liability for the performance of region inference. Therefore one probably ought to combine it with a simple analysis of which procedures that may side-effect which global variables, either by themselves or by provedures that they call. If a procedure does not have any effect on a certain global variable, the threading of that variable can be short-circuited around calls to it. Furthermore, global variables of atomic (non-pointer) type do not affect region inference at all, and may be omitted in practise.

The same solution is good for modeling a nested procedure's access to local variables of enclosing blocks in Pascal, as long as the nested procedure is called directly. If one needs to model references to nested procedures to be passed as parameters (as Standard Pascal allows) or stored in "function pointers" (as I am told that newer members of Borland's family of Pascal dialects support by some kind of implicit thunking), the implementation's actual access mechanism may need to modeled more exactly.

6.1.4 By-reference parameters

Passing arguments by reference is often used in imperative programs to "return" more than one atomic value from a subroutine. It would be easy to dismiss this use as obsolete – in a modern language with automatic memory management, the argument would say, it should be easy to construct a tuple of return values and return that. However, in fact many widely used language still lack convenient syntactic support for constructing such a tuple and (in particular!) picking it apart in the caller. And even in languages that have such syntax, the natural intertia of programmer habits means that the idiom of passing by reference is often common nevertheless. One reason for this is, of course, that unless one consistently adopts a value-oriented programming style, there is little practical benefit of returning a tuple instead of using call-by-reference – and in traditional (or garbage-collected) implementations call-by-reference is often the more efficient option.

Therefore, a region-based implementation of an imperative language ought to be able to handle the call-by-reference idiom with reasonable grace. Once we develop a way to represent pointers to global and local variables (which we will do in the following subsections), the reference-passing mechanism itself could be expressed "directly" in the UHL model. But this is not really a good solution for region inference.

The major problem is that when we pass a pointer *into* a procedure instead of getting a value *out* of it, the region type system must use the type of the input pointer to close the link between where the procedure produces the output value and where the caller expects to find it. This means that the regions containing the produced value must be decided before the call, thereby preventing the callee from selecting the regions itself and pass them as output regions.

This will harm the precision of region inference – for example, we saw in Section 2.4.1 that this ability were essential for handling the Game of Life example well without artificual rewriting.

Another problem is that by treating the target variable as something that can be pointed to, we force the region type system to apply to it rule from Section 5.1.6 that updates cannot change its region-annotated type. Thus we lose the usual ability of local variables to get new region-annoatated types when they get new values.

For these reasons, it is desirable to try to convert the call-by-reference to copy-restore in the UHL representation of the program; the UHL model natively allows a procedure to return more than one value. It takes some program analysis to determine when this transformation is semantically sound; but when it is, the two approaches give the same results, so we are free to use the efficient call-by-reference mechanism in the object code the compiler eventually produces, while pretending to the *region inference* that we are actually doing copy-restore. Thus we can get the best of both worlds.

One minor problem with converting call-by-reference to copy-restore is that the extra return values may sometimes prevent a tail call from looking like a tail call. That might harm the heuristics the the region inference ordinarily uses to prevent agent code from causing loss of tail contexts (one such heuristic will be presented in Section 7.3.8). Fortunately, there are not many lanaguages where both of call-by-reference and tail recursion are common idioms.

6.1.5 Pointers to global variables

If the language has an address-of operator that can create pointers to global variables, things get more complex, as the UHL model does not have any built-in concept of a mutator value $w \in \mathbb{W}$ referring to an UHL variable rather than a heap cell.

I believe the most flexible solution is to consider a pointer to a global variable to be a "non-pointer" at the UHL-level, like we did for statically allocated constant data structures in Section 4.4.3. The UHL representation of Reads and writes through pointers that may point to global variables must be instrumented with a switch on the pointer value that goes to either an ordinary heap operation, or a simulated access to the threaded UHL variable that represents the pointed-to global.

However, unlike in Section 4.4.3 the instrumentation needs to be visible to the region inference, so it will pay (in terms of precision of the region inference and possibly also the efficiency of the inference process) to do a pointer analysis in advance to determine which pointers have any risk of pointing to which globals at all. There is a large literature on suitable pointer analyses for imperative languages – see Hind [2001] for a recent survey with lots of references.

6.1.6 Pointers to local (stack-allocated) variables

Real challenges arise then the address-of operator is used to create a pointer to a *local* variable and it cannot be handled as an instance of call-by-reference that can be transformed to copy-restore. The general solution for global variables cannot be used, because a local variable in a recursive procedure may

exist in unboundedly many incarnations simultaneously, and they cannot all be threaded into other procedures at once.

The peculiar property of pointers to local variables is that the host language's type system usually cannot prevent the pointer from escaping the lifetime of the pointed-to variable, so a well-typed mutator can be memory unsafe even without allocating a single heap cell. We have two different viewpoints to choose from in this situation.

One is to view the dangling pointer to a former local variable as part of the problem that region inference must solve. After all, the programmer's specification that the variable is to be local (*i.e.*, allocated on the call stack) is a form of manual memory management, and automatic memory management in general exists to prevent erroneous manual memory management from causing program crashes. According to this viewpoint, the declaration of a seemingly-local-but-pointed-to variables should be represented as a *heap allocation* in the entry to the variables's scope, and then it should be left to region inference (or garbage collection) to deallocate it at an appropriate, safe time. A pointer to the freshly allocated memory block would be stored in a "shadow variable" which is used to simulate the address-operator, in addition to direct accesses.

Later in the process, it should be the agent's right to allocate space for some regions on the call stack (which cannot be expressed in in the raw UHL model as presented here, but might be possible as a general extension of the model; see Section 3.5.3). Then the original local variable may or may not end up actually being allocated on the stack, as may any allocation that on the surface refers to the heap.

A variant of this principle is to take the programmer's choice to use a local variable as a hint that the block should be stack allocated if possible, even if it means extending its lifetime until the end of the block it is local to. (The programmer may wish this because stack allocation is faster and more efficient than allocation in a heap region). Taking the hint would involve inserting a dummy read of the variable at the very end of its scope, which would force the region inference to extend the lifetime appropriately. If, however, a yet longer lifetime were necessary such a dummy read would not change anything.

The opposite viewpoint says to Trust The Programmer. He presumably knew what he was doing when he specified a specific lifetime for the variable and would have used `new` or `malloc` if he wanted automatic memory management.[1] This positions may be a little unusual, but it will be instructive to investigate its consequences.

One first observation to make about it is that it is compatible with our decision (Section 3.3) to take region soundness rather than region safety to be the main correctness criterion for agents. A region-sound agent is allowed to let the program crash with a pointer error, as long as the program also crashes with a pointer error when no heap memory is reused. And precisely that is, intuitively,

[1] The "ideology" of automatic memory management commonly asserts the non-existence of trustworthy programmers as a fundamental truth. However, some real-world programmers *can* be trusted, and some of these are not above using automatic memory management to reduce the boring parts of programming, as long as they can decide for themselves when to take it and when to leave it.

the case when the crash is caused by a stray pointer into the *stack* rather than the heap.

Then comes the question of how to express the trust-the-programmer viewpoint faithfully in the UHL model. A first attempt would be to extend the UHL with primitives for explicit allocation and deallocation. If these are not region annotated they could be used to specify the allocation and deallocation of stack-allocated objects explicitly in the uniform mutator.

A problem with this approach is that if the mutator attempts to access memory through an obsolete stack pointer, the ideal semantics will end up in the Wrong state, which is not formally distinguishable from the kind of errors that the agent *is* responsible for avoiding. If the uniform mutator itself is not memory safe, there can be no region-safe agents (see footnote 5 on page 92), so we cannot use Theorem 3.46 to establish that an agent is sound.

This can be avoided by distinguising lexically between stack pointers and heap pointers, such that a read or write operation only goes Wrong if the pointer is an unknown *heap* pointer but blocks immediately for an unknown *stack* pointer, just as for a non-pointer. Then it would also be natural to split the H part of the semantics configuration into a map for the heap pointers and one for the stack pointers.

But it turns out that this can all be modeled in the ordinary UHL language if we thread the entire stack around in the program as if it were a global variable (which, in a sense, it is). The stack can grow arbitrarilty large, but the UHL semantics does not constrain the size of what can be stored in an UHL variable. Specialized misc operations can be used to allocate and deallocate stack frames, and to instrument read and write operations to jump around the heap-access operation and simulate it using the stack representation if appropriates.

The attraction of this solution is that the results about region soundness in Section 3.3 would all be directly applicable to the uniform mutators that are built with it. One might expect major changes to the region type system in Section 5.2, but only little explicit support would be needed. The region type system would enforce a special status for the UHL variable that represents the stack, and special rules would be necessary for the misc operations that access the stack. However, if the canonical type of a stack pointer is $\langle \cdots \rangle_\perp$, the misc variants used for instrumenting a read or write would add no constraints on the typing beyond those of the read or write itself – so the region inference will not need to to see these instrumenting miscs at all!

The safety proof for the region type system in Section 5.3 would of course need a major upheaval, adding store typings for the stack in parallel to the one already maintained for the heap.

6.1.7 Separate initialization of new heap blocks

Some languages have a model for heap allocation where freshly allocated blocks of heap memory is initialized to either random bit patterns or some kind of default values. After the allocation, the caller must explicitly fill in meaningful values.

This is usually not a problem for the UHL model. The fresh block can be considered to have the right types from the beginning if only default contents is considered to be non-pointers. Even if the default contents are random bit

patterns that might coincide with a pointer, we can imagine the UHL semantics running on a machine with "invisible" origin tags on all data, such that a heap access using a pointer that appeared by chance in a freshly allocated block does not go Wrong but just gets stuck silenty.

In the region type system, one can let the permissions of the fields in the memory block be `rw` until they have been filled, and then – if applicable – change them to `ro` afterwards such that subplacing will be available.

There is, however, a small potential for problems here: The region type system will demand that the region annotations for the values that are filled into the block are decided on no later than the block is allocated. This means that programs which allocate the block first and then go on to compute the values that should be in it will not have the full power of the HMN-like agent programming language available for the compuatation. This is a problem in particular for Prolog, where such early allocations is a common idiom (see Section 6.3.3).

6.1.8 Pointer tricks

As described in Box 3.1 (page 55), the operational model of UHL supports all kinds of dirty pointer tricks with no trouble. However, the region type system does not work with them at all, and I know of no way to generate agents for mutators with pointer tricks automatically.

6.2 Object-orientation

We now turn to the techniques needed to represent object-oriented programs in the UHL model. We will imagine a host language that is "somewhat like C++ or Java", although these two full languages are not currently within the scope of automatic region inference.

Of course, much of the discussion in Section 6.1 will also be relevant to object-oriented host languages.

Because I have explicitly chosen not to consider modular region inference in the thesis, we will work under the **closed-world assumption**, that is, that we know the entire class hierarchy of the program. This assumption was also made by Christiansen and Velschow [1998], who were the first to try to define region-based memory management for a Java subset; it seems to be essential for designing region type systems for object calculi at the current state of the art.

6.2.1 Class hierarchies and subtyping

Assume a "standard" representation of objects at run time: The object consists of a number of consecutive heap cells, of which the first contains a class identifier and the following ones contain the values of the object's fields. Fields that are inherited from a superclass are placed first, such that a pointer to the object can also count as a pointer to an object of the superclass, except for the value of the class identifier. The class identifier is commonly a pointer to a statically allocated class description, but we will treat it as an opaque magic number here.

Small implementation-dependent variations on this theme are, of course, common, for example in connection with multiple inheritance.

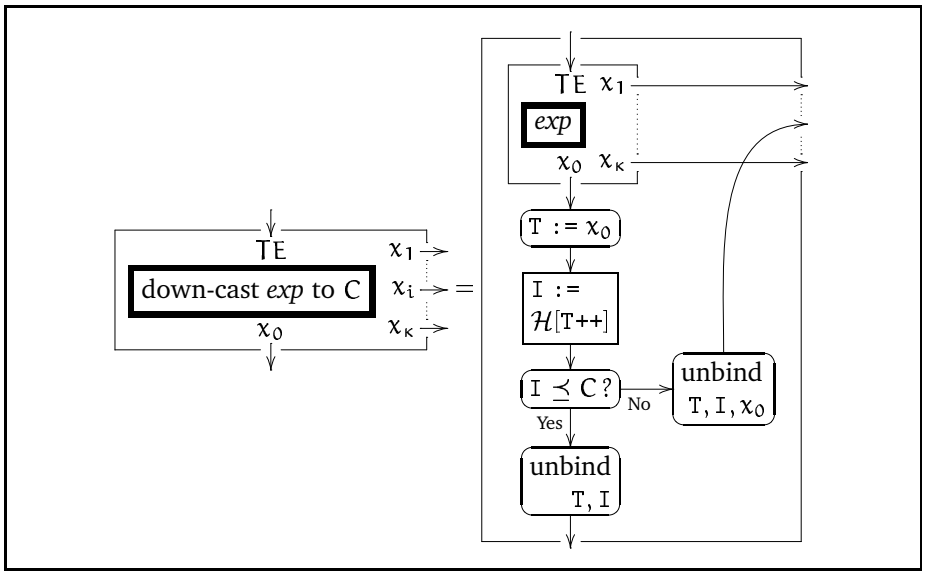
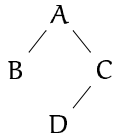


Figure 6.1: Translation of a down-cast expression

For a single class, creation of objects and access to fields (reading values or writing a new value) fits well with UHL's heap primitives. Field accesses generally ignore the class identifier.

Up-casts and **down-casts** are operations associated with the class hierarchy. An up-cast is usually a no-op operationally, and is also invisible at the UHL level. A down-cast consists of reading the the class identifier and throwing an exception if the object's real class is not either the target class of the cast or one of its subclasses. A possible UHL representation is shown in Figure 6.1.

Now consider how to represent this in the region type system. In the host language's object-oriented type system, a value of type "pointer to C" may actually be a pointer to C or any of its subclasses. Therefore, when we translate the host-level types to region-annotated types, we need to know the entire class hierarchy, say,



and the type vertex for a "pointer to C" becomes

$$\langle \{A\}, \theta_{A1} \mid \{B\}, \theta_{A1}, \theta_{B1}, \theta_{B2}, \theta_{B3} \mid \{C\}, \theta_{A1}, \theta_{C1} \mid \{D\}, \theta_{A1}, \theta_{C1}, \theta_{D1}, \theta_{D2} \rangle_p$$

The down-cast in Figure 6.1 can be typed by inserting a *RTEXPLODE* before $\boxed{\texttt{T} := x_0}$ and typing the inspection operation separately in different environments for each of the "pointer to precisely this class" types like $\langle \{D\}, \theta_{A1}, \theta_{B1} \rangle_p$. After the "yes" branch, only "pointer to precisely C" and "pointer to precisely D" are left, and each of these can be converted to the ordinary "pointer to C" type

$$\langle \{C\}, \theta_{A1}, \theta_{C1} \mid \{D\}, \theta_{A1}, \theta_{C1}, \theta_{D1}, \theta_{D2} \rangle_p$$

by the *RTEXPAND* rule.

Upcasts can be typed by the same general principle: First use *RTEXPLODE* split the environment into cases for each class in the subhierarchy below the "old" class; then use *RTEXPAND* to convert each of these to the standard type for the "new" class. (In practise, of course, one is free to imagine a joint rule that does both of these at the same time).

For field access, use *RTEXPLODE* on the temporary copy of the pointer just before the first read instruction. When the temporary pointer is unbound after the actual access, the difference between the different exploded environments disappear (the field that is access has the *same* type in all of the relevant classes), and no special rules are necessary to make them converge again.

6.2.2 Dynamic method dispatch

Many object-oriented languages provide static methods and perhaps also free-standing funcitons that are called directly. These are, of course, easy to implement in UHL. Dynamic methods, where a method call goes to different code depending on the actual class of the object, require a little ingenuity.

The solution is much like the one for ML in Section 4.1, but is actually simpler because we don't need any separate control-flow analysis to identify procedure groups. Instead, we can use the principle that

Each method name corresponds to one UHL *procedure.*

except that this short phrasing tacitly assumes that we use alpha renaming to get rid of identically-named methods in disjoint parts of the class hierarchy and C++-like overloading of methods.

Consider again the class hierarchy from before:

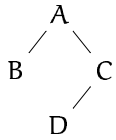


Figure 6.2 shows the overall structure of an UHL procedure representing a method $\mathtt{m}$ that is defined in class A and have overriding definitions for C and D. If we ignore, initially, the dotted nodes at the top right, this looks much like Figure 4.2 on page 102. The dispatching code at the top can be understood as a simultaneous down-cast to each of the classes that implement the method. The $\boxed{\mathit{nop}}$ entry node is there to make space for the *RTEXPLODE* rule that opens the down-cast idiom.

If the implementations in C and D contain explicit calls to the inherited implementation, these calls can be done directly through a secondary entry point at its beginning, thanks to the UHL model's support for multiple entries. In this way, a direct call from $\mathtt{C.m}$ to $\mathtt{A.m}$ might avoid having to set up the region properties of the arguments in ways that are required only by $\mathtt{D.m}$.

A more ambitious use for multiple entries is sketched by the alternative dispatch code s_C^m that is shown in dotted lines at the top right half of the figure. If we let the UHL program contain that as well as there ordinary s_C^m we can encode into the uniform mutator the fact that when we call the $\mathtt{m}$ method of an object

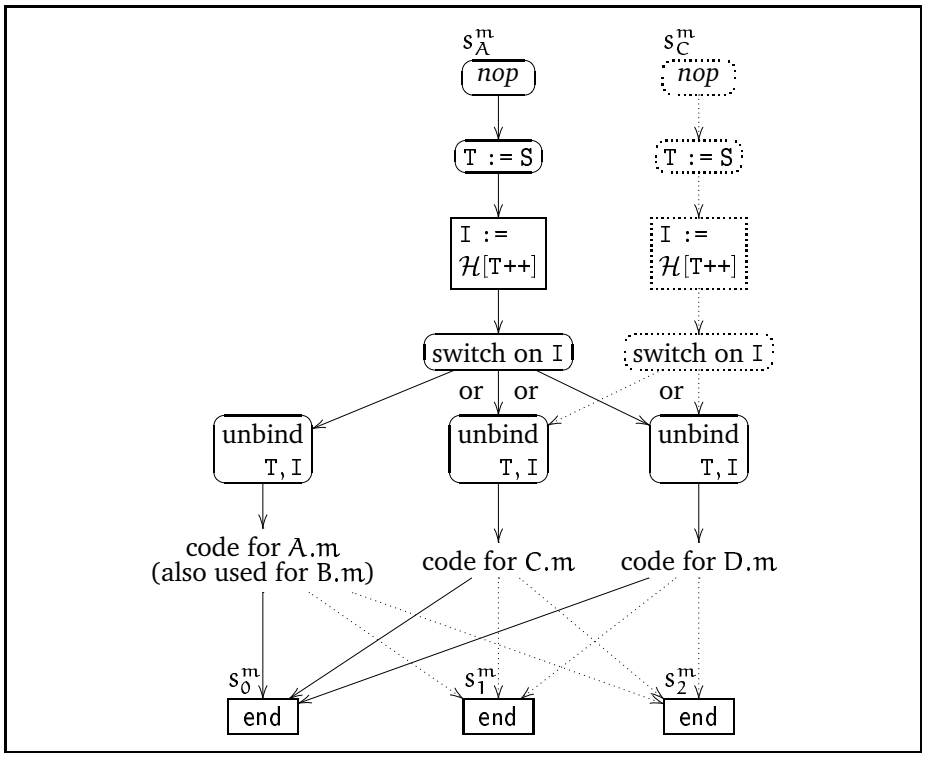


Figure 6.2: Sketch of a procedure group with consisting of three methods that implement the same interface m*. The calling convention is that* S *contains the self (or* `this`*) pointer for the receiving object; explicit arguments are passed in other* UHL *that are not shown in the figure.*

we know as a C, there is no risk that the call will end at A.m. This may be beneficial if A.m needs some special region properties, for example that two of the method's arguments must be in the same region.

However we can only put this construction to full use if the outgoing edges from the switch on I nodes are annotatable. Otherwise both entrypoints will be in the same flowchart chunk, and the same region parameters must be passed in both cases. However having an annotatable edge here means exactly that the method may need to start by doing different according to where the call comes from. This is not directly possible with the usual "vtable" implementation of dynamic dispatch, so some special support from the implementation is needed here. The simplest option would be to have several different entries for m in the vtables of C and D. One is used by calls that consider the target object an A, the other by calls that know it to be a C or D.

6.3 Logic programming

The least mainstream programming language to which region-based memory management has been adapted is probably Prolog. The first work on combining Prolog and regions was my M.Sc. thesis [Makholm 2000b], in which I

constructed an experimental region-based Prolog compiler from scratch (and very little knowledge about how Prolog is usually implemented). Later, as part of my Ph.D. work, I collaborated with Kostis Sagonas on adding region-based memory management to an existing, competitive, Prolog implementation. The joint paper that resulted from that work Makholm and Sagonas [2002] mainly describes operational issues that are not touched upon in this thesis.

6.3.1 Backtracking

The most conspicuous feature of Prolog is its control-flow constructs, which are based on backtracking and "cut"s. One would expect this to create problems with fitting it to the UHL model; backtracking can cause the flow of control to jump back into a deep recursion tree that had apparently all been unwound. Such a very non-local control flow seems to be incompatible with the execution model of UHL.

However, this problems have already found a (satisfactory, I think) solution with the *backtracking-aware region manager* that I created for the M.Sc. prototype and described in [Makholm 2000c]. The backtracking-aware region manager implements the same interface as the standard model described in Section 1.2 but in addition to this it executes backtracking and choice-point manipulation operations in unison with the mutator. When the mutator backtracks, so does the region manager, unwinding its entire (extensional) state to what it was when the choice point in question was created. Regions that were supposedly deallocated spring to life again, their contents intact, and regions created since the choice point disappear without trace. Even allocations made in existing regions after the choice point was created are undone.

I will not describe here how the backtracking-aware region manager works its magic; interested readers are referred to the M.Sc. thesis [Makholm 2000b].

However, the fact that the region manager handles backtracking transparently means that the region inference can proceed as if backtracking never happens. On the other hand it must be prepared that a predicate call may jump to another of the callee's clauses than the first matching one (when what actually happened was that the first clause was tries but that execution path eventually failed and caused backtracking).

The only question is whether we can still trust the results of Chapter 3 (such as Theorem 3.46, for example) without changing the UHL semantics and redoing all of the proofs. It turns out that we can, by the trick of modeling Prolog's sequential nondeterminism by the implicit nondeterminism of UHL's misc operation. We can record choice-point operations as special "I/O events", such that once the (ideal or managed) semantics of UHL has produced a set of behaviors, we can extract the actual extensional behavior of the Prolog program from that.

Example. Consider the Prolog program

```
sub(X) :- write(s1), fail.
sub(X) :- write(s2).
sub(X) :- write(s3), write(X).
main :- write(m1), fail.
main :- write(m2), sub(p1), write(m3), fail.
main :- sub(p2), write(m4).
main :- write(m5).
```

```
?- main.
```

where the `writes` are there to make the backtracking behavior visible from the outside. (Prolog's `write` primitive displays text to the user; it does not match the UHL convention that "write" refers to the heap and "output" to the external world). The program can be represented as a uniform mutator with the (abstracted) ideal behaviors

$$\begin{aligned}\mathcal{A}bs(()\,[\![\mathcal{M}]\!]) = \{\ & (\mathsf{Try}_1, \mathtt{m1}, \mathsf{Fail}),\\ & (\mathsf{Try}_2, \mathtt{m2}, \mathsf{Try}_1, \mathtt{s1}, \mathsf{Fail}),\\ & (\mathsf{Try}_2, \mathtt{m2}, \mathsf{Try}_2, \mathtt{s2}, \mathtt{m3}, \mathsf{Fail}),\\ & (\mathsf{Try}_2, \mathtt{m2}, \mathsf{Try}_3, \mathtt{s3}, \mathtt{p1}, \mathtt{m3}, \mathsf{Fail}),\\ & (\mathsf{Try}_3, \mathsf{Try}_1, \mathtt{s1}, \mathsf{Fail}),\\ & (\mathsf{Try}_3, \mathsf{Try}_2, \mathtt{s2}, \mathtt{m4}, \Downarrow),\\ & (\mathsf{Try}_3, \mathsf{Try}_3, \mathtt{s3}, \mathtt{p2}, \mathtt{m4}, \Downarrow),\\ & (\mathsf{Try}_4, \mathtt{m5}, \Downarrow) \qquad\qquad \}\end{aligned}$$

(as well as prefixes of them, of course). From this set one can mechanically recover the observable behavior of the Prolog program:

$$\mathtt{m1}, \mathtt{m2}, \mathtt{s1}, \mathtt{s2}, \mathtt{m3}, \mathtt{s3}, \mathtt{p1}, \mathtt{m3}, \mathtt{s1}, \mathtt{s2}, \mathtt{m4}, \mathsf{Yes}, \Downarrow$$

The special event "Try_n" means that the program has reached a choice point and (nondeterministically) chosen the nth way to proceed. "Fail" means of that the current execution path has failed. The uniform mutator will be written to kamikaze after having emitted Fail – say, by jumping to an empty misc node – whereas the actual Prolog program backtracks. When we *interpret* the set of ideal behaviors, we can recover the backtracking by looking for another (abstract) behavior that has a prefix in common with the failed one.

The point of this roundabout definition is that the (mathematical) function from $\mathcal{A}bs([\![\mathcal{M}]\!])$ to the observable behavior can be used on $\mathcal{A}bs([\![\mathcal{A}]\!])$ from the managed semantics without changes. Therefore, if we know that an agent is *region sound* in the UHL world, it will also be true in the Prolog world that the observed behavior of the program with the agent is the same as the observed behavior without memory management. And therefore it is safe to use region-inference algorithms that assume that the program *will* kamikaze instead of backtracking.

6.3.2 Types

Prolog is a typeless language, which means that the region inference needs to include its own type inference such that the region type system will have some types to place its region annoatations on. There is quite a bit of literature on type analyses for logic programming, but much of what I have reviewed is based on assumptions (such as giving a free variable an atomic type with no place to put structured region annotations) that make them difficult or impossible to use as a basis for a region type system.

My prototype implementations use ordinary techniques from type inference from ML-like languages to do type inference in a simple "soft typing" system of extensible tagged sums. If the type inference uses a graph unification algorithm to unify types without occurs-checks, it can invent enough recursive types to

type any source program, so the type system does not need to be visible to the programmer.

If the program uses Prologs `arg/3` and `functor/2` to build or analyze values dynamically, the task of the type system gets harder. In the experiment in [Makholm and Sagonas 2002] we added some rudimentary support for these primitives on an ad-hoc basis to the extent that our (legacy) benchmark programs required them. But I have no good general theory of this problem to offer.

6.3.3 Unification

The other nonstandard feature of Prolog, aside from backtracking, is that first-order term unification is a primitive and indeed the only way to build and inspect structured data. This mechanism has enough of an imperative flavor that many of the considerations in Section 6.1 are valid for Prolog, too.

Superficially, unification is just a limited form of destructive updates on the heap, but treating it just as that will degrade the strength of region inference severely. In fact, only a few cases of unification will usually need such treatment. The majority of unification operations in a Prolog program can be identified as either just building new terms, taking existing terms apart, – or returning data to the caller of a procedure.

Unification is the only mechanism for passing data out of a predicate – essentially all returning of values must be done using call-by-reference. Thus the recommendation in Section 6.1.4 of attempting to convert call-by-reference to copy-restore applies doubly to Prolog.

A special problem is that Prolog programmers very often use the call-by-reference returning of values to initialize fields of heap blocks that have been allocated with default contents (*i.e.*, unbound variables). For example, a common idiom for constructing a list is

```
make_list([X|L]) :- compute_element(X),
                    !,
                    make_list(L).
make_list([]).
```

and this programming style is often used for the creation of non-recursive data, too:

```
compute(In,Out) :- Out=data(X,Y,Z),
                   subroutine(In,1,X),
                   subroutine(In,2,Y),
                   subroutine(In,3,Z).
```

where the region type system from Chapter 5 will not allow `subroutine/3` to choose the regions for its output, because the region annotations for `Out` must be decided as soon as the `data` block is allocated. I have no finished ideas about how to repair this. The logic-programming tradition contains many examples of *mode analyses* that infer the kind of information we need here, but I have not yet investigated how to combine them with a region type system.

6.3.4 The WAM

Most Prolog implementations are built around a bytecode language that descends from **Warren's Abstract Machine** [Aït-Kaci 1991; Warren 1983]. According to the principle in Section 1.4.2, region inference should take place at the WAM bytecode level. In the region inference for [Makholm and Sagonas 2002] we tried to do region inference at the Prolog source level in a stand-alone process, but it turned out that the correctness of the generated agent depended on being able to predict within rather strict tolerances which exact sequence of WAM instructions the bytecode compiler would generate. The prototype Prolog region inference sometimes guesses wrong and generates agents that may not be safe. A non-toy implementation would almost certainly need to do region inference only *after* most of the phases in the bytecode compiler.

The WAM uses three kinds of memory for data

1. A bank of *scratchpad registers* named `X1`, `X2`, `X3`, ... These can be modeled directly by UHL variables.

   There are also various other registers, most of which have to do with the implementation backtracking and need not be modeled in the UHL representation.

2. The **local stack**, which is more or less a call stack with bells and whistles (for backtracking) added. WAM instructions can refer to "permanent variables" `Y1`, `Y2`, etc., which are locations in the stack frame of the current procedure.

   `Yi` variables can be pointed to by `Xi`'s or `Yi`'s in other stack frames. This is a little more benign by general pointers to local variables, and the bytecode compiler guarantees that pointers to `Yi`'s will not escape the lifetime of the stack frame they point into. Thus they can be safely handled by reifying the entire local stack as an UHL variable, as described at the end of Section 6.1.6.

3. The **heap**, which is the part that we usually want to manage with regions.

The WAM uses a data model where a heap cell (or local variable) in general can contain a pointer *and* some tag bits that describe how to interpret the pointed-to value. This need some extensions to the syntax of types in the region type system, but no large conceptual changes.

Chapter 7

Region inference algorithms

7.1 Overview

Region inference in general inherently depends on the host language. In our setting, this means: If it is to be done well, one needs to take into account how the uniform mutator was produced from a concrete program in a concrete host language. This is because the process involves developing an understanding – even if just in a mechanical sense – of what the mutator does and how it works. And too many hints about this get dropped on the floor on the way from an implementation-specific intermediate language to the raw UHL representation.

It is natural to object that the region type system of Chapter 5 (and the arguments in Chapter 6 that it, or something much like it, applies to a wide range of languages) show that the safety (and, by implication, soundness) of an agent can be understood independently of the host languages. But that is like saying that because we know the rules of first-order predicate logic and the axioms of set theory, mathematicians do not need intuition. It is one thing to recognize a proof (or a region typing) when we see one, but a completely different challenge to create one, much less to decide what to prove, and yet much less to do it automatically.

Is all lost, then? Must we write off the UHL model as useless for finding agents automatically? (How sad it would be to reach such a conclusion after digesting 150 pages of preliminaries!) But no, the UHL abstraction is still useful even if it needs a little outside support. But we must strive to isolate those parts of the region-inference process that depend on the host language, and phrase as much as possible in pure UHL terms such that the creative work required for adapting a new host language (or implementation) to the model is minimized.

Similar considerations apply to the region type system. The system presented in Chapter 5 is fairly broadly applicable, but it is by no means the final word on reasoning about the safety of agents. For example, it would be nice to have a system that supported ML-like polymorphism, or that handled destructive updates of heap cells in a more flexible way. Anticipating future development in the area of region type systems, we should treat the region type system, too, as a replaceable component of the region inference – or at least try to identify the parts of the region inference that would be unaffected by a switch to a radically different region type system.

These goals lead to dividing the region-inference process into three phases:

- First, decide on a **skeleton** typing for the original uniform mutator – that is, a typing (according to the region type system of Chapter 5) where the places have not yet been filled in. The points in the program where *RTSUB* is used will also be determined in this phase.

  This task is where knowledge of the host language is necessary. If the host language is typed, knowledge of the host typing will be very helpful in constructing the skeleton typing – in Section 7.2 we will detail how to do this for the running ML example.

  If the host language is not typed, the region inference will need to do its own ("soft") typing to construct an appropriate skeleton, but even in this case knowing about the host language is useful. For example, a region inference for Prolog needs to pay special attention to the mechanism the implementation uses for unification. That would not be necessary for, say, Scheme – on the other hand, when constructing a skeleton for Scheme it would be advisable to have special heuristics for dealing with the common Scheme idiom that the value of the first element of a list determines the type of the following ones.

- Second, **basic region inference** computes a first-approximation agent for a given mutator. The overall goal is for this agent to deallocate data as soon as possible, except that it must (obviously) be sound. Of course, because soundness (and even safety) are extensional and therefore undecidable properties, deallocating everything "as soon as possible" must be understood as an unattainable ideal rather than a hard requirement of optimality. Deviations from the ideal may result from impreciseness in the region type system that we use to prove soundness, or from approximations we make to improve the efficiency of the basic region inference itself. In some cases it will also be necessary to choose between two possible agents where some data will be deallocated earlier by one choice and some other data will be deallocated earlier by the other.

  I give a detailed account of my techniques for this phase in Section 7.3. They do not directly depend on the host language, but they do depend on the general structure of the region type system. Some sections of Chapter 6 described small alterations to the region type system, which it would be straightforward to adapt the techniques to – but if a region type system based on entirely different principles were to be used instead, most of this phase would have to be reinvented.

  The result of basic region inference is a well-typed agent. By Theorem 5.16 we will know that it is safe, and by the reasoning principles in Section 3.3.2 (in particular Theorem 3.46) we will know that it is sound.

- Third, do **region optimizations** on the output of the basic region inference. They can be viewed as as ordinary compiler optimizations applied to the agent (but of course tailored to optimize the kind of optimizations that basic region inference produces) and are therefore independent of the host language as well as the region type system, as long as the agent programming language stays unchanged.

  The reason for needing region optimizations is that in basic region inference, we do not worry about is the efficiency of the generated agent itself

– for example, the number of region operations it executes or the number of region parameters it passes to and from procedures. The task of the region optimizations is worry about such things, but work under the constraint of not changing the deallocation decisions the agent make. (This constraint corresponds to the usual expectation of compiler optimizations that they do not change the program's semantics).

Some essential region optimizations will be described (more briefly that I had hoped for) in Section 7.4. They are all *conservative* in the sense of Theorem 3.53 and therefore preserve *region soundness*. That is all we need if we trust the implementation of the optimization, but we will also briefly discuss how they preserve typeability in the region type system, such that optimized agents can be certified.

7.1.1 The prototype implementation

The region inference algorithms I present are the product of practical experience with the region inference for HMN that I wrote for the benchmarking results reported in Henglein et al. [2001]. At some points in the text I contrast what I did in the prototype implementation with what I propose to do more generally. Therefore, it is useful to know something about the constraints of the prototype implementation:

The prototype works for the HMN system's host language FUN, with a few practically motivated extensions: User-defined `datatypes` and Standard ML records (including n-tuples for $n \geq 2$ and a unit type).

One major difference between FUN and the subset we considered in Chapter 4 is that FUN allows functions to be defined only at the program's top level. Because functions cannot be curried either, this means that heap-allocated closures are not needed. Functions are still values which can be passed to and from other functions and stored in data structures, even though that is not quite as useful as in the usual simply typed lambda calculus.

In the FUN and REGFUN implementation, function values are represented by "non-pointer" constants that identify the function actually referred to. In the implementation the constant is actually a pointer to the compiled function body's entry point, but because the compiled code is not heap allocated it does not count as a pointer for region-inference purposes. Because indirect calls may be present, we still group functions into procedure groups as described in Section 4.1, but the (virtual, UHL only) entry code for a procedure group switches directly on the function value rather than trying to read the first word of a closure.

Another difference from Chapter 4 is that the prototype does not support exceptions. There is no deep reason for this, except that I invented the UHL approach to exceptions after the prototype was constructed.

Some further comments about the relation between the prototype and the UHL model in general. The prototype was written before I developed the UHL model that I have described in this thesis so far. It began as a series of transformation steps, each written as a conventional syntax-driven recursion on FUN and REGFUN abstract syntax trees. As more transformations were added,[1] I discov-

[1]The prototype has a somewhat larger repertoire of individual transformations and optimizations than described in this chapter, because for benchmarking purposes we added the capability of

ered that the source code for a typical transformation had a lot of "plumbing" to let intermediate results propagate along the syntax tree between rather small pieces of code that did the real work. This led to copy-and-paste programming, where a major source of errors was incomplete adaptation of old code to new situations.

Therefore, I restructured the code such that all transformations used a few general combinators (functors, actually) for traversal of abstract syntax tree. One goal of this was to eliminate code duplication; another was to isolate knowledge about the representation of abstract syntax trees to the traversal combinators, such that the FUN language could evolve without changing all of the transformations. This latter goal was met with success; shortly after the restructuring changed FUN from having a built-in list type to supporting datatypes. Thanks to the traversal combinators, this could be done with essentially no changes in the source files that defined the actual transformations.

This experience gave me the first idea that it might be possible to code region-inference techniques in a language-independent fashion. The idea was put to the test later, when I needed a region inference for Prolog in the experimental work I was doing with Kostis Sagonas [Makholm and Sagonas 2002]. Eventually it did turn out to be possible to share inference code between the FUN and Prolog implementations, by writing new implementations of the traversal combinators that transformed Prolog ASTs but otherwise had the same interface as the FUN traversors.

But the result was not pretty. It became clear that the signatures I had written for the FUN traversors were not quite as general as I had supposed – hidden assumptions about the structure of the language popped up here and there, and in some cases I needed to add some "hooks" to the combinators that had not been necessary in the FUN world (and then back-port the same hooks to the FUN implementation, such that the to implementations could keep sharing code letter-for-letter). This experiment taught me that a working theory of language-independence would need a more general and orthogonal interface between the language-specific and the transformation-specific parts of the implementation, than the rather ad-hoc traversor signatures I had developed for the FUN implementation. So I set out to develop such an interface.

The UHL model, as described in Chapter 3, is my result so far. I have moved from a higher-order interface ("a mutator is something that can be traversed in this-and-this way") to a first-order view ("a mutator is something that can be represented in UHL") because the latter makes it easier to specify the *semantics* of the common interface. Even though our uniform mutators are not meant to be programs that exist on the computer, it is nevertheless possible to *imagine* that they are programs, which is very helpful for understanding the individual transformations within the region inference. What a uniform mutator means and does can be explained using techniques from operational semantics that should be familiar to any research-oriented compiler hacker. On the other hand, it would be very difficult to explain how a higher-order traversal combinator is supposed to work, precisely enough to judge whether a proposed implementation is in fact correct, without appealing to some kind of UHL-like intuition.

"emulating" the TT, Kit and AFL models within the HMN framework, that is, constructing HMN agents with a strength that roughly parallels what a real TT region inference followed by the storage-mode or AFL analyses may reach.

As of this writing, my implementation of region inference has not yet been rewritten to be based on the UHL model rather than on the traversal-combinator model. Nevertheless, where I mention the prototype implementation in the explanations that follow, I will word the descriptions *as if* it were based on the UHL model but otherwise used the same algorithmic ideas as the real one does.

7.2 Constructing skeleton types

As mentioned in the overview, the first part of region inference is to find a way to fit the region type system of Chapter 5, or something sufficiently like it, to the uniform mutator – but without worrying about the "places" $\mathtt{p}$ within types yet.

In this section, I describe how this can be done for monomorphic programs in the the ML subset from Chapter 4.

The descriptions in Chapter 5 hopefully made it clear to the reader how most ML types can be expressed in the region type system. It might be thought that this is too obvious to warrant a section of discussion. However, as in Chapter 4, the important point here is not how to do it for ML in particular, but describing the "obvious" process in such a way that it is easier to see how it can be generalized to other host languages.

7.2.1 From ML-like types to type graphs

Imagine we have an ML-like type system (still without polymorphism) with an abstract type syntax such as

$$\begin{aligned}\text{Types: } \mathtt{t} ::= {} & \mathsf{int} \\ \mid {} & \mathsf{int_B} \\ \mid {} & \mathtt{t} \times \mathtt{t} \\ \mid {} & \mathtt{t}\ \mathsf{list} \\ \mid {} & \mathtt{t} \rightarrow \mathtt{t}\end{aligned}$$

Now, non-pointer types such as int and $\mathtt{t} \rightarrow \mathtt{t}$ (or booleans, characters, floats...) are all represented by a link to a type vertex marked $\langle\rangle_{\mathtt{p}}$. The place $\mathtt{p}$ will never be constrained by the typing rules (because non-pointers are used in neither cons, read, or write rules), so the region inference can proceed as if it were not there. One can imagine it being fixed to $\bot$, for example.

Boxed integers $\mathsf{int_B}$ can be represented by a link to a type vertex marked $\langle\pi_0\rangle_{\mathtt{p}}$ where π_0 represents the int type. Here $\mathtt{p}$ *is* used, of course.

Product types such as $\mathtt{t}_1 \times \mathtt{t}_2$ (where $\mathtt{t}$ ranges over ML types) are easily represented by a link π such that $\Pi(\pi) = \langle \mathtt{ro}{:}\pi_1, \mathtt{ro}{:}\pi_2\rangle_{\mathtt{p}}$, where π_1 and π_2 represent $\mathtt{t}_1$ and $\mathtt{t}_2$ respectively. Likewise, a list type $\mathtt{t}\ \mathtt{list}$ can be represented by a π such that $\Pi(\pi) = \langle \mathtt{ro}{:}\pi, \mathtt{ro}{:}\pi'\rangle_{\mathtt{p}}$, where π' represents $\mathtt{t}$, at least for the special-case run-time representation shown in Figures 4.6 and 4.7.

Generic user-defined datatypes are somewhat more complex. To the best of my knowledge, they are a field where all the existing literature leave it to the reader to imagine the "obvious" generalization. However, it still ought to be written down *somewhere* what this obvious generalization is, and perhaps it is

not so obvious after all? The general form of a datatype definition is[2]

```
datatype ('a,'b,'c) T1 = C11 of t11 | ... | C1n of t1n
     and ...
     and ('a,'b,'c) Tk = Ck1 of tk1 | ... | Ckn of tkn
```

where the argument types t_{ij} may contain the placeholders `'a`, `'b`, `'c`, as well as the type constructors T_1, ..., T_k themselves. It introduces new type syntax

$$\begin{aligned}\text{Types: } t ::= {} & \cdots \\ & \mid (t,t,t)\,T_1 \mid \cdots \mid (t,t,t)\,T_k\end{aligned}$$

Conceptually, what the prototype region inference does is to translate the datatype definition into a "template" type graph $\check{\Pi}$ defined by:

- Select distinguished links π_a, π_b, π_c and π_1, ..., π_k.
- For each new type constructor T_i, let

  $$\check{\Pi}(\pi_i) = \langle \mathtt{ro}{:}\{C_{i1}\}, \mathtt{ro}{:}\pi_{i1} \mid \cdots \mid \mathtt{ro}{:}\{C_{in}\}, \mathtt{ro}{:}\pi_{in}\rangle_{p_i}$$

  where π_{ij} is the representation of t_{ij} – with the special rule that whenever it is necessary to represent `'a`, the distinguished link π_a is used (and likewise for `'b`, `'c`), and whenever it is necessary to represent (`'a`, `'b`, `'c`) $T_{i'}$ for some i', the distinguished link $\pi_{i'}$ is used.
- $\check{\Pi}(\pi_a)$, $\check{\Pi}(\pi_b)$, $\check{\Pi}(\pi_c)$ are left undefined.

Whenever we later want to represent $(t_1, t_2, t_3)\,T_i$, we make a fresh copy of the *entire* $\check{\Pi}$ and instantiate the copies of π_a, π_b, π_c as representations of t_1, t_2, t_3 themselves.

This recipe does not say what to do if one of the type constructors being defined appears on the right-hand side of a definition with a *different* set of operands than the operand list on the left-hand-side. This can happen in two cases. The first, and benign, is if the programmer used the syntax for mutually recursive datatypes without actually using mutual recursion, as in

```
datatype 'a foo = Foo1 of int | Foo2 of 'a
     and 'b bar = Bar of 'b * int foo
```

The other case is if the programmer actually uses polymorphic recursion in types:

```
datatype 'a complete_tree = LEAVES of 'a
                          | BRANCHES of ('a * 'a) complete_tree
```

which for some reason Standard ML does allow, even though it is hard to make sensible use of such a datatype without polymorphically recursive *functions*. The prototype "solves" this problem by not supporting polymorphic type recursion – a type such as `complete_tree` cannot be expressed as a finite type graph at all. If one wants to really support such types, one either has to cut off the recursion artificially at some point (which is always possible as long as polymorphic recursion in functions is forbidden), or to replace the generic graph-based handling of type recursion with something more ad-hoc and syntactic.

[2]The reader is supposed to generalize this discussion to the situation with more or less type arguments than three, of course.

7.2.2 Region-annotated type expressions

The previous discussion have shown how to get from an ordinary tree-shaped ML type to a graph-shaped region type. Indirectly it also shows how to derive a tree-shaped notation for region types. Namely, the type graph will still be tree-shaped in its *global* structure: Each node in the type tree corresponds to a "cluster" of nodes in the type graph, and though the clusters may contain cycles internally, the links *between* clusters closely match the original type tree's shape.

Now, we can reach a tree-shaped syntax for region types by adding to each production of the type syntax exactly the ρ's that decorate the nodes in its "cluster" of the type tree. Normally there is exactly one such node, but we except the non-pointer node where we ignore the ρ – and for a datatype there can potentially be a large number of ρ's: one for each node in the template graph $\check{\Pi}$ for the datatype. If we select some arbitrary but canonical order for these many ρ's, we get

$$
\begin{aligned}
\text{Region-annotated types: } \bar{\tau} ::= {} & \mathsf{int} \\
| \;\; & (\mathsf{int}_B, \rho) \\
| \;\; & (\bar{\tau} \times \bar{\tau}, \rho) \\
| \;\; & (\bar{\tau}\ \mathsf{list}, \rho) \\
| \;\; & \tau \to \tau \\
| \;\; & \cdots \mid ((\bar{\tau}, \bar{\tau}, \bar{\tau})\ \mathsf{T}_i, \rho, ..., \rho) \mid \cdots
\end{aligned}
$$

which is pretty close to the syntax of HMN's "region types" on Figure 2.1 (on page 42). The only thing that is missing is the peculiar HMN syntax for function types.

In the translation *to* type graphs before, a function type translated to simply $\langle\rangle_\perp$, which is why it did not change in the grammar for $\bar{\tau}$ – there is no ρ to add to it. What the HMN function type really does, seen from the UHL perspective, is not to describe the bit pattern in the code pointer itself. It describes the typings for the entry and exit end of the *procedure group* that the actual function is part of. Remember from Section 4.1 that the HMN region type system doubles as a simple control-flow analysis to justify the division of the program into procedure groups. The HMN rules for function calls are written to get information about the callee from the type of the function value, but a better way to think of it may be: "look up the information in some global table, indexed by the procedure group label on the call". Only, in the HMN type system, the contents on the global table is replicated in the type syntax wherever a type that refers to the procedure group appears. That fits with the traditions of type-system design, but in the actual prototype region inference, function types are represented simply by an index into a procedure group table.

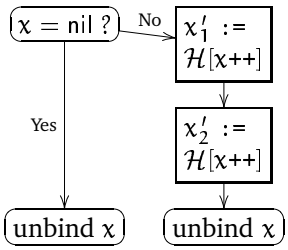
7.2.3 The skeleton typing in detail

Now that we know how to represent single type expressions by type graphs and vice versa, we can go on to derive an entire skeleton UHL typing from an ML type derivation. For this task we will follow two invariants:

1. A subplacing rule will be inserted each time an edge crosses an expression boundary (in the same sense as in Section 4.3), but nowhere else.

2. The UHL region type system's ability to have more than one type description for a particular control state will only be used *locally* while reading a memory block, until the switch of a tag value happens. In particular, such ambiguous typings will never happen across an expression boundary.

The first of these invariants mean that in any *local* network of mutator operations within the translation of an expression – for example



in the translation of case analysis from Figure 4.7 – the type graph will be the same throughout; only the environment can change. This common type graph must contain the representation of the ML types of all variables. Ideally, the type graphs for different UHL variable should be kept disjoint, but sometimes the UHL region type system's rules enforce sharing between them. For example, under the condition that the initial $\Gamma(x)$ in the fragment above must be the representation of a list type, the rules for the two read operations together imply that the initial $\Gamma(x)$ must equal the final $\Gamma(x_2')$, and that it must share its type graph with $\Gamma(x_1')$.

(Note, however, that such sharing of type vertices does not need to propagate across subplacing steps. The many-to-many nature of the subplacing relation means that the representation of two types can be distinct before a subplacing step and shared afterwards – *or vice versa!*)

The constraints on sharing inside the type graph work together to enforce exactly the identities between types and regions that are implied by the use of the same μ three times (and the same ρ two times) in the premises for the HMN typing rule for case expressions on page 47. The requirement that the twice-occurring ρ is a region variable rather than $\top$ corresponds to the side condition $p \neq \top$ in the UHL system's *RTREAD*. On the other hand, the UHL type system does not explain why the place could not be $\bot$ – and indeed the HMN type system would stay sound if it allowed $\bot$ in expressions where values were read rather than allocated.

In the same way, the rest of the rules of the HMN type system can be derived systematically from the UHL region type system and our decisions about how ML types are represented as type trees. The "anonymous entries" in HMN environments are seen to correspond to Section 4.2's "fresh (UHL) variables" when the latter were not bound to anything in the translation environment.

Sometimes there will be minor differences – for example, the HMN rule for "[]" wants to have a region as a place, whereas no restrictions can be derived from the UHL system and Figure 4.6 (the node x_0 := nil works with any place for x_0 at all, thanks to Definition 5.14(2).). These differences originate in differences assumptions about the target implementation: In Chapter 4, the empty list was represented by a special non-pointer value, whereas the HMN article assumed that a [] expression would actually need to allocated an "empty list" structure somewhere on the heap.

7.2.4 Adding closures

When we add general closures as in Section 4.1 we cannot represent a ML function type with $\langle\rangle_{\mathrm{p}}$ anymore.

If we know the types of a function definition's free variables, it is easy to to construct a region-annotated type for a closure referring to exactly that function definition. In the same way we can construct the type of a closure referring to any of the function definitions in a procedure group, using bars in the type.

The only problem is that a function definition can contain a free variable of the same type as the function being defined itself. Then the region type corresponding to the procedure group must be a recursive type. One can also construct examples with mutual recursion in closure types between two or more procedure groups.

In general, this can be solved by imagining a group of recursive datatype definitions where each procedure group corresponds to a (nullary) type name and each function definition corresponds to a data constructor. This implicit datatype definition can then be represented as a type graph using the general scheme for datatypes describes in Section 7.2.1

7.3 Basic region inference

The overall principle of the basic region inference is to try to distribute the mutator's data across as many regions as possible. This will further the general goal of deallocating data early, because a heap block can only be deallocated when everything else in the region it belongs to are clear to be deallocated. Having as small regions as possible will minimize the risk that heap blocks that are actually not used anymore will be kept alive by other inhabitants of their regions that *are* still needed.

Ideally the "optimal" implementation of this goal would be to have each heap allocation have its own region. However, this is easily seen to be impossible in general: Any well-formed agent has a static bound on how many regions it can have alive at one time for each recursion level, but most mutators will be able to allocated unboundedly many heap blocks, depending on the input, and expect them all to be still available. So some amount of region-sharing is necessary in general. This restriction is also implicit in the design of the region type system from Chapter 5: A type graph must be finite yet be able to describe, say, arbitrarily long lists, which means that some heap cells will be described by the *same* vertex in the type graph and thus need to be in the same region, namely one named in the vertex's place component.

We will try to minimize the region size by constructing a region typing in where as many different region variables as possible are used to describe each data state. Of course, because regions may be aliased, different region *variables* does not necessarily imply different *regions*, but this strategy will nevertheless help towards the original goal.

The basic region inference depends only on some fairly high-level features of the region type system. It ought to be a trivial taks to adapt the algorithm to another system that fits the following description, even if it is not exactly equal to the one from Chapter 5.

- A typing describes each control state with a **type graph** and a map from mutator variable names to vertices in the graph. (Or there may be several such graph–map pairs, for reasons explained in Section 5.1.5).
- Some[3] of the vertices in the type graph contain a **place** $\mathtt{p}$ which can be either $\bot$, $\top$, or a region variable ρ.
- Each *edge* in the type graph contains a **permission** ν which can be either `rw` or `ro`.
- The typing rules for the various mutator operations entail some restrictions on the shape of the type graph, as well as on the places and permissions within it. For example, *RTREAD* requires that the vertex referred to by π and the ones each of the π''s all contain the same place, and that this place is not $\top$.

  As another example, there are two typing rules for cons operations, which have much the same premises except for the place in the vertex that describes the newly allocated pointer. Together, these rules constrain the place to be either $\top$ or a region variable ρ; $\bot$ is not possible. The choice of rule must match the region annotation on the cons operation.
- The two different control states at either end of a flowchart edge may be described by different type graph by virtue of the **subplacing rule** *RTSUB*. The subplacing rule says that certain relations should hold between the two type graphs. These relations are certified by subplacing and type matching certificates, it is easy to compute the smallest certificates that contain all the required relation. Such a smallest certificate consists of a relation, generally many-to-many, between vertices (and edges) in one graph and vertices (and edges) in the other graph. The places (or permissions) that decorate related vertices (or edges) must match up according to Definition 5.10(b) – or, in some cases, be equal, by Definition 5.7(b).

7.3.1 General invariants and region-operation selection

Let us first consider how to create suitable region annotations once we have decided on place annotations on the skeleton typing such that the region typing rules are fulfilled within each chunk. (Recall the "chunk" concept from Section 3.4). One sort of region annotations we do not consider here are region parameter lists, which will be discussed in Section 7.3.4.

Some of the internal edges of each chunk may contain a *RTSUB* step. Without loss of generality, we can also assume that the annotatable edges that connect the chunks with each other also contain a *RTSUB* step each – if it is not already there it is easy to insert. Thus the subplacing edges divide each chunk into "**proto-chunks**", each with one common type graph.

We will maintain the fundamental invariant that

> *While the chunk is executed, the bound (counted) region variables are exactly the ones that appear at least once in one of its type graphs.*

[3] Actually, it would have been conceptually cleaner if *all* vertices contained a place and singleton types were a special case of π rather than a special case of τ. But this would have needed another meta-variable letter in the grammar for the type system, which is difficult enough to read already.

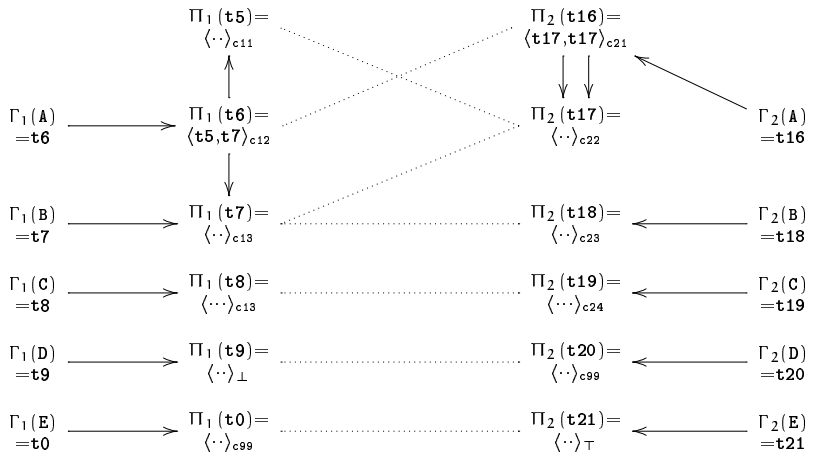
To convince ourselves that this is a good invariant, we must as two questions: Do we really need all of these region variables? And: Would anything be wrong with having more?

The first question is the trickiest one. Actually, our region type system does not strictly require that all region variables that occur in the type graphs are also bound. But this is just because it would have been too complex, technically, to express the invariant; we may nevertheless think of it as a "moral" invariant. This does not effectively restrict the *expressiveness* of the region type system: If we have a region scoping $\mathcal{D}$ and a region typing $\mathcal{T}$ that breaks the invariant, we can always derive one that does not, simply by replacing each region variable that is not bound with $\top$. This will be possible because the only rule that prevent a region variable in the type graph becoming $\top$ is *RTCONS*, but the corresponding scope rule *ASCONS* asserts directly that the region variable cannot be unbound, either. Another problem would be if a region was visible in the type graph at a time where it became bound or unbound, such that the doctored typing did not satisfy the subplacing requirements, but side conditions on the *RTXXX* rules for region operations prevent exactly that.

The second question was: Would anything good come of having more bound region variables than mentioned in the type graphs? At run-time such a region variable could have three kinds of binding. Either it is aliased to another live region variable, in which case it is superfluous. Or it is bound to an empty region, in which case we might as well move the creation of it to the end of the chunk. Or it is bound to a region with some thing in it, in which case the region type system allows us to deallocate it at entry to the chunk! Because deallocating things early is the overall goal of basic region inference, that is what we should do.

Now for the actual generation of region annotations. The easy part is the "at ρ" or "nowhere" annotations on cons nodes. We can read these directly off the type graph – a region variable as decoration on the type vertex that represents the allocated block gives rise to an at annotation; a $\top$ gives rise to a nowhere. The case $\bot$ for this particular place must be forbidden; it corresponds to no correct region typing.

The difficulty is all in selecting region operations for an annotatable edge that connects two chunks. In general, the type graphs on each end of the annotatable edge will have different shapes and annotations; we must insert suitable region annotations to make them match up according to the region type system. A representative example would be



where we want to invent region annotations to go from state (Π_1, Γ_1) to (Π_2, Γ_2).

The dotted lines between the two type graphs symbolize the minimal subplacing certificate that can certify $(\Pi_1, \Gamma_1(\chi)) \leqslant (\Pi_2, \Gamma_2(\chi))$ for $\chi =$ A, B,..., E. For our purposes here, the difference between subplacing and type matching is not important; both kinds of certificates say that *if* two related nodes are annotated with region variables, *then* the region variables must be the same. So the minimal certificate says that the region known as c11 before the edge must be the same as c22 after the edge, c12 before must become c21 after, and so on.

Elements of the subplacing relation that go to and from a vertex marked $\bot$ or $\top$ can simply be ignored. Because c99 on the LHS appears only in t0 which has no region-variable partner to the right, c99 can be treated as if its was not mentioned in Π_1 at all. Such region variables (which must be bound in the left chunk but do not take part in the region-variable matching across the annotatable edge) should be released in the region operations on the edge.

At the other end of the edge there is another c99 which has nothing to do with the one just considered. It, and other region variables that must be bound in the right chunk but have no LHS partners, must be bound to a fresh, empty region during the region operations.

We now need to construct region operations that implement this relation between LHS region variables and RHS region variables. The relation between c12 and c21 is easy – it can be achieved with the region operation

rename c12 to c21

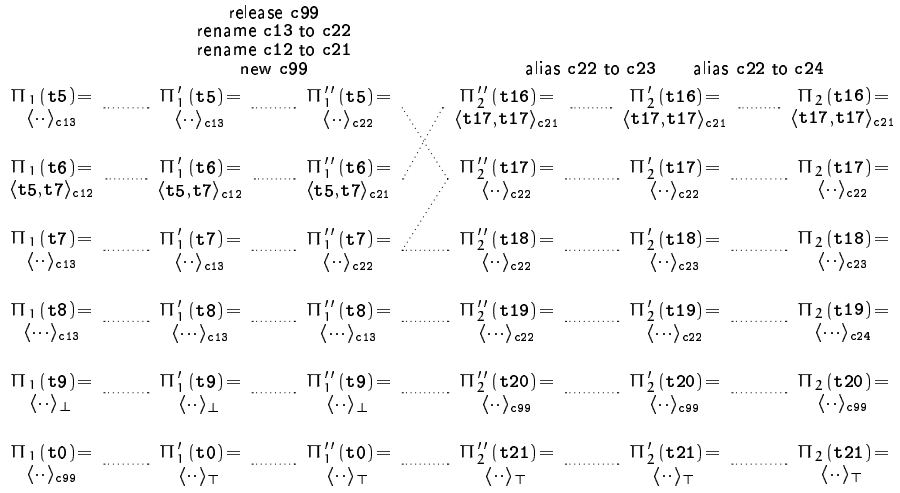
c13 is a little more involved. There are two vertices marked c13s on the left-hand side, with three partners on the right-hand side. So the region known as c13 before must become c22, c23, *and* c24 after the edge. We can achieve this by some alias operations:

rename c13 to c22; alias c22 to c23; alias c22 to c23

Wait! We have forgotten the relation between c11 and c22 – and there is trouble brewing here. The subplacing certificate says that c22 after the edge must be the same as c11 before *and* the same as c13 before. This is impossible – there is no legal sequence of region operations that will allow us to do a "reverse alias". Instead what we need to do is to change the region-variable names on the left-hand side such that c11 and c13 becomes identical:

If $(\Pi_1, \pi_1) \leqslant (\Pi_2, \pi_2)$ *and* $(\Pi_1, \pi_1') \leqslant (\Pi_2, \pi_2')$ *are both in the certificate, and* $\Pi_2(\pi_2)$ *and* $\Pi_2(\pi_2')$ *contain identical region variables, then the region variables in* $\Pi_1(\pi_1)$ *and* $\Pi_1(\pi_1')$ *must be identical, too.*

What we have just seen in the example is an instance of this rule with $\pi_2 = \pi_2' = \texttt{t17}$. If we do enforce the rule, say by writing `c13` instead of `c11`, we have a working sequence of region operations, however:



where the first column is an auxiliary subplacing step that changes `c99` to $\top$ in $\Pi_1(\texttt{t0})$ such that `c99` is not in $\mathrm{frv}\,\Pi_1'$ and can be released before the new `c99` is created (which, similarly, must happen before `c99` appears in its new place in the type graph.

This principle of this example works in general. On each annotatable edge we create the following sequence of region operations:

1. (An auxiliary subplacing step that removes region variables with no RHS partners from the LHS type graph such that they can be released).
2. release operations for region variables that are bound in the before-chunk but have no RHS partners.
3. rename operations that rename each of the RHS region variables to one of the LHS variables they correspond to.
4. new operations for each region variable that must be bound in the after-chunk but have no LHS partners.
5. (The main subplacing step goes here).
6. alias operations that construct the necessary aliases where several LHS variables share the same RHS partner.

7.3.2 No loops, no procedures, no subplacing

What is left now is to find a way to a way to chose places in the skeleton typing. As an easy beginning we will consider the restricted case where the uniform mutator contains no call nodes and is acyclic, at least after having been simplified to chunks. We will also, for the time being, ignore subplacing and assume that all places p must be region variables.

Consider a single chunk. It consists of a number of proto-chunks, separated by subplacing steps and each with its own type graph. Not all of the regions mentioned in the type graphs will be different. As already mentioned, some of the typing rules impose requirements that two or more places in a single type graphs must be identical to each other. The subplacing rules force equalities between places in *different* type graphs – these can be found by computing the minimal subplacing certificate that contains the relations needed by the *RTSUB* rule. All of these requirements combine to form an equivalence relation between the places in all of the chunk's type graphs. This equivalence relation can be computed efficiently by a union-find structure.

Once the equivalence relation has been computed, we can forget most of the information about how the chunk looks inside. The only thing we need to remember are the region annotations on conses and the type graphs for the proto-chunks where annotatable edges attach to the chunk. (Often it will be possible to use shortcuts to compute the equivalence relation even more efficiently by using precomputed summaries of the relation for idioms often produced by the translation from host language to UHL. Then one never needs to represent the internal structure of those idioms directly).

However, the equivalence relations for different chunk also influence each other through the subplacing condition on page 169: equivalences in later chunks may provoke further equivalences in earlier chunks (but not in the other direction, thanks to reference counting and alias operations). This suggests that we can do the region inference as a **weakest-precondition** computation, where a "weaker precondition" means a type-graph annotation that allows more different regions. Therefore, we should consider the chunks in reverse topological order, from the end to the beginning of the program. And that is what the prototype inference does:

Algorithm. Process the chunks in reverse topological order. For each chunk, do the following:

1. Compute the region-variable identities that are forced by the contents of the chunk.
2. Add in further identities between region variables, as specified by the subplacing structure of the outgoing annotatable edges.
3. Choose region-variable names for the resulting equivalence classes of type annotations. The exact names chosen are not important; there will be renaming phases later in the region inference process. However, for readability of the intermediate agent, the prototype inference does try (heuristically) to assign each equivalence class a name that matches one of its RHS parters in an outgoing edge. Equivalence classes without any RHS partners always get fresh names.
4. Construct region annotations on each of the outgoing edges, according to the principles in the previous section.
5. Proceed with the next chunk in the reverse topological order.

It should be intuitively clear that this algorithm produces an agent that causes every block of heap memory to be deallocated as soon as possible under the given restrictions (however, "the given restrictions" includes the skeleton typing and the premise that no subplacing is to be used).

(That is, if it results in a well-formed agent at all. That does not happen automatically; if the chunk that contains s0 has any nontrivial type graphs, the generated agent will expect to be started with a number of region variables already bound to (empty) regions. This is not a large practical problem, however. It will be solved when we add subplacing. Or one may add an artificial entry chunk in front of the old entry point, with an annotatable edge that will carry the requisite new operations).

7.3.3 Handling uncounted variables

Our next topic is to consider how to extend the inference algorithm from the previous section to work with subroutines. Subroutines add two sources of complication to the task. First, the interdependence between region annotations on call sites and on callee code means that a single topological traversal of the program is not sufficient to fix the region annotations. Second, the UHL calling mechanism means that uncounted region variables enter the picture, and they need special support even while handling other code than calls.

In this section, when I speak of a "procedure" I mean one of the maximal (weakly) connected components of the mutator's flowchart. There may be multiple actual entry points in the procedure, but as long as they are all connected we will consider the totality of the code that is reachable from them a single procedure.

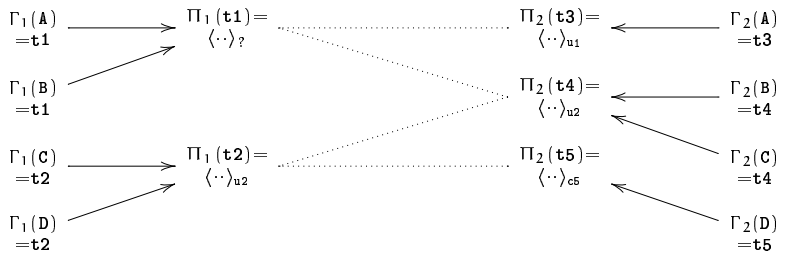
Let us start by discussing region inference for a "leaf" procedure: a (maximal) strongly connected component of the flowchart that does not contain any call nodes itself. We want to build upon the weakest-precondition idea from the previous section, so assume that we have already determined, somehow, which annotations to put on the type graphs on the end nodes in the procedure group.

The problem is now that, due to circumstances of the caller(s), it may be necessary for some of the type vertices at the end nodes to be annotated with uncounted region variables. However, these are the only uncounted region variables we need; there is never a reason to use any uncounted region variable that is not present at at least one end node.

How can we extend the region-operation selection from Section 7.3.1 to work with uncounted region variables? First, of course, we must specify that uncounted region variables are never released or initialized with new – an uncounted region variable is bound for the entire execution of the procedure, or not at all. Second, an uncounted variable cannot be the target of alias or rename operation. From this we derive the following rule:

> *If a type vertex on the right-hand side of an annotatable edge is decorated with an uncountable region variable and has a partner on the left-hand side, then the partner must be decorated with the same uncountable region variable.*

This rule has some perhaps unexpected consequences. Consider the following match-up of an annotatable edge:



With which region must $\Pi_1(\texttt{t1})$ be decorated? The rule says that it must equal u1 and also equal u2. The only way to make this true is to rewrite *all* region typings in the procedure we have constructed so far, and replace all occurrences of u1 by u2, or vice versa. (However, there is nothing wrong with having $\Pi_2(\texttt{t5})$ keep its c5 – that can be implemented by "alias u2 to c5", because uncounted regions *can* appear at the *source* side of an alias operation).

This can be implemented by working with two levels of union-find structures. The **local** union-find works within each chunk, just as described in Section 7.3.2. The **global** union-find is active for the whole traversal of the procedure and keeps track of identities among the uncounted region variables. Each equivalence class at the local level may or may not have attached a reference to the global level; if two local classes are unified and both have such a reference the unification propagates to the global level. On the other hand, it is not necessary to propagate unification from the global level to the local one; two local classes can just be considered "implicitly unioned" if both happen to reference the same global classes. When an inter-chunk subplacing relation is considered, references found in right partners are moved to left partners and unified with references already found there, if any.

After the entire procedure has been traversed, the final state of the global union-find structure can be used to backpatch cons annotations that refer to uncounted variables and alias operations with uncounted variables as the source.

7.3.4 Calling a leaf procedure

For the more difficult problem of what the inference should do for call chunks, we first consider an easy special case: Assume that the leaf procedure from before has exactly one call in the entire mutator. (If we were genuinely interested in producing a good agent, the intelligent plan would of course be to inline the entire procedure, but for the moment we are more concerned about understanding the *RTCALL* rule.)

RTCALL mentions three type graphs decorated with the caller's region variables:

- Π contains the types of the local (UHL) variables of the caller that are *not* passed to the procedure as arguments.
- Π_1 contains the types of the variables that get passed as arguments to the procedure. (Π_2 in the rule is the same graph, but decorated with the callee's names for the region variables).
- Π_4 – of which there is one for each exit from the procedure – contains the type of the return value. (Π_3 are the same graphs with the callee's names for region variables.)

Without loss of generality (if necessary, by adding some trivial subplacing steps in the callee's typing skeleton) we can assume that all these component type graphs are all disjoint.

(If the call chunk contains any "unbind" nodes in front of the call node, $\Pi \oplus \Pi_1$ will also need to contain types for the unbound variables. We can put them, as disjoint components, in Π, but we need not worry about them in particular: Even without a systematic treatment of subplacing it is easy to see the subplacing rules always allow these graphs to be decorated with all ro's and $\top$'s.)

Now, an algorithm for basic region inference for the call chunk could be:

1. We assume that the inference steps for the succeeding chunks (the targets of the return jumps in $\mathfrak{Jmp}$) have already been completed.
2. For each return jump, use the normal algorithm to derive identities among the region variables in the respective $\Pi \oplus \Pi_4$. Use a single local union-find structure for all of these.
3. For each return jump, inspect the Π_4 to create constraints on the corresponding Π_3. The rules are as follows:
   - Any vertex that is decorated by an *uncounted* region variable in Π_4 must be decorated by an *uncounted* (but not necessarily the same) region variable in Π_3.
   - Any vertex in Π_4 whose decoration has been unified with something in Π must be decorated with an *uncounted* region variable in Π_3.
   - If two vertices in Π_4 have been unified with each other but not with something in Π and do not refer to an uncounted variable, the must be decorated with *identical* (counted) region variables in Π_3.
4. When all the Π_3's have been suitably constrained, do the algorithm from Section 7.3.2 recursively for the callee. The callee has its own global union-find that works on the uncounted region variable positions in the Π_3's
5. After the recursive region inference, there may be more internal region identities in some of the Π_3's than in the corresponding Π_4. Redo these identities in the call chunk's local union find.
6. The recursive region inference may have done some unifications in the callee's global union-find structure. Replicate these in the call chunk's local union-find, even if they concern uncounted variables that started out in different Π_4's. (This may cause the unification to propagate to the called procedure's global union-find structure in the usual way). Now each uncounted variable in the callee is associated with an unambiguous class in the call chunk local union-find.
7. Inspect Π_2 (which was created in the recursive region inference), and convert any identities between region variables to identities in Π_1, to be recorded in the call chunk's local union-find structure. For each vertex in Π_2 that holds an uncounted region variable ρ^u, unify the corresponding places in Π_1 with ρ^u's associated class.
8. Assign names to the call chunk's counted region variables and fill in region annotations on its outgoing edges in the normal way.

9. Construct the input $\mathfrak{Ra}$ from the relation between Π_1 and Π_2. This is possible because step 7 is the only source of identities between the places in Π_1 that correspond to counted region variables in Π_2.

   Construct $\mathfrak{Ua}$ from the association between the callee's uncounted region variables and the call chunk's local union-find.

   Construct each per-return $\mathfrak{Ma}'$ from the relation between its associated Π_3 and Π_4.

7.3.5 Unrestricted calls

The above algorithm is all well and fine, as long as there is only one call to each procedure. As soon as some procedure can be called from several places, it becomes difficult (and quickly impossible) to arrange for the computations such that the traversal of a caller is blocked while the region inference for the callee is going on. Recursive calls, for example, mane the algorithm breaks down instantly.

Therefore, in the general case we will have to replace step 4 of Section 7.3.4 with *guessing* some usable annotations on Π_2 and the Π_3's. Later we can try to analyze the body of the callee and see if our guess was right – if it wasn't, the analysis produces a new "more right" Π_2 (and Π_3's), which we can use as a guess in a renewed region inference for the procedure that contains the call. By repeating the analysis of each procedure often enough, we may hope to reach a fixpoint that correspond to a consistent, well-typed agent for the mutator.

To be sure that the iteration eventually terminates, we decide on the principle that

> *Each new guess will have all the identities between counted region variables that the old guess had, except for positions where a counted variables has been replaced by an uncounted one. The set of positions that have uncounted variables on them will be nondecreasing, and all identities between uncounted variables in the old guess will still hold in the new guess.*

Thus, each time we need to update a guess either the number of positions with counted variables will decrease, or it will stay stable and the number of *different* region variables will decrease. This bounds the number of iterations to a small multiple of the size of the skeletons for Π_2 and the Π_3's.

The algorithm for region inference on a whole program begins with creating initial guesses for all procedures by being maximally optimistic: The best possible guess is that each vertex in a Π_2 (the type-graph part of $\mathcal{T}(s)$ where s is a procedure entry) or a Π_3 (the type-graph part of $\mathcal{T}(s)$ when $\mathcal{A}(s) = \mathsf{end}$) is decorated with its own counted region variable and not identical to other variables in the same graph.

Then all procedures in the program are put into a worklist, and the algorithm proceeds as long as there are still pending procedures in the worklist, by selecting one of them, and then run the weakest-precondition algorithm on it.

During the inference, step 3 of Section 7.3.4 does not create fresh Π_3's but instead add fresh restrictions (identities between region variables and or this-must-be-uncounted marks) to the existing guesses for Π_3. If any of the new restrictions were not already satisfied, the callee will be added to the worklist

(if not already there), so that its body will eventually be analyzed with the new postconditions. Then, no matter whether the Π_3's changed, the *old* guess for Π_2 is used. It will still make sense in the context of new Π_3: Any uncounted variables in Π_2 will still appear in Π_3 after new restrictions have been added.

If an "uncounted" mark is added to a position in the Π_2 that has already been unified with something in the same Π_2, the uncountedness should propagate to the other positions in the class – but right after, the class should be dissolved into individual uncounted-variable positions that will only be identified again if necessary because of *internal* circumstances in the procedure.

When an entire procedure body has been analyzed, a check is made to see if its Π_2's and Π_3's differ from the guesses when this iteration of the analysis started; if any of them do, all procedures that contain calls *to* the procedure will be added to the worklist.

Eventually, it will not be possible to restrict the guesses more, and the iteration will end with a consistent set of annotations. This is the initial agent that is the output of basic region inference.

This iterative algorithm does *not* always find an agent that deallocates everything as soon as possible. Indeed, we have seen in Section 2.4.4 that such optimal agents do not always exist. But the basic weakest-precondition method in Section 7.3.2 does find (or is believed to find) optimal agents within its restrictions. What goes wrong when we add procedures?

An analysis of the counterexample in Section 2.4.4 shows what goes wrong. While analyzing a call, if two places in Π_4 have been unified, but not with anything outside Π_4, the rules say that the corresponding places in (the guess for) Π_3 must also be unified. But there is another way to allow the two places in Π_4 to be identical: Mark the corresponding places in Π_3 as uncounted, and utilize the fact that $\mathfrak{Ua}$ need not be injective.

As long as we assumed that there was only one call to the procedure, the method of unifying in Π_3 is always superior over using uncounted variables: The two uncounted variables will always be aliased anyway, and a single counted variable is more flexible than two uncounted ones.

However, when there are more call sites, a unification in Π_3 can propagate to Π_4 of a different call (via step 5 of the call chunk algorithm). In some cases it will harm the region performance there, while the uncounted-variables solution would have allowed a better solution.

Section 2.4.4 tells us that there is no general solution to this problem, so the prototype inference handles it by pretending that it does not exist – it will stick to the unification method until other circumstances forces one of the places in Π_3 to be uncounted anyway, and then switch to the uncounted-variable method.

It is actually the case that the algorithm I outlined here is subtly sensitive to strategy used to select a pending procedure to remove from the worklist. Consider the program

```
fun f () = (sqrt(5),sqrt(7))
fun g (x,y) = (x,y)
fun h () = let val (x,y) = g(0.0,0.0) in [x,y] end
fun k () = let val xy = (sqrt(2),sqrt(3)) in [g(f()),xy]
```

We ignore the region used for the pairs and only look at where the reals are allocated. One analysis history might be:

`h` is analysed. It needs to call textttg such that both reals end up in the same region. The guess for `g` is updated with one counted region for both sides of the return value. This does not immediately propagate to the Π_2 guess for `g`, so the initial optimistic guess for `f` need not be changed.

`g` is analysed. This updates the Π_2 guess to require that the two inputs are in the same (counted) region. Because the guess has changed, `k` and `h` are put into the worklist.

`k` is analysed. It needs the output from `g` to have the same regions as the $\sqrt{2}$ and $\sqrt{3}$ it has already computed, so the output of `g` *must* be in uncounted regions. So the single counted region in `g`'s Π_3 guess is replaced with two uncounted regions. However, the Π_2 guess still says that the input must be delivered in one counted region. This assumption propagates to the Π_3 guess for `f` which now says that `f` must deliver its two outputs in the same region.

`g` is analysed again. This updates the Π_2 guess to expect the input in two different uncounted regions.

`h` is analysed again. The guess for `g` now works in terms of two uncounted regions, which is sufficient for `h` to force them to be equal. So the guess for `g` needs not change.

`k` is analysed again. It is still satisfied with the new guess for `g`, which now means that it does not need `f` to deliver its result in a single region anymore. But the old information in `f`'s cached Π_3 guess cannot just be discarded, because the algorithm does not keep track of where the identities beween the outputs come from.

So in this history `f` ends up being forced to deliver its output in a single region, even though its only caller does not need that anymore. This could be avoided if the first analysis of `k` happened before the first analysis of `h` – then `g` would never be analysed with a postcondition saying that the return values must be in the same region.

It might be interesting in further work to experiment with other and perhaps more intelligent heuristics for this problem, but such experiments will need a large supply of benchmark programs that are known to *have* the problem. In the (granted, somewhat limited) set of benchmark programs that I have tried and reported in [Henglein et al. 2001] and [Makholm and Sagonas 2002], I have not been aware of any region-inference anomaly that could be traced to this effect. This suggests that the problem may be rare in practise.

Eventually the best solution to the dilemma would probably be to provide a reasonable default heuristic and a way for the programmer to artificially force certain positions in the return type of a given function to be implemented by uncounted variables.

7.3.6 Subplacing in the prototype

We still have not discussed how to construct region typings that include $\top$ and $\bot$. Intuitively, we want as many of these as possible – $\top$ because it is are neces-

sary to allow regions to be deallocated so early that they leave dangling pointers, and $\bot$ especially because a $\bot$ instead of a region variable in the Π part of *RTCALL* may prevent the return value from needing to be in an uncounted region.

Therefore we want, conceptually before the inference algorithms in the previous sections run, to locate the places in the type skeleton that *cannot* be either $\top$ or $\bot$, and then consider only these places for region variables.

By inspection of the region type system's rules, we see that the only eventual source of reasons-not-to-be-$\bot$ is the `cons` operation, and the only eventual sources of reasons-not-to-be-$\top$ are read and write operations. Our task is to propagate these reasons along the region type system and see where they coincide. Only places where there is a reason not to put $\bot$ *and* a reason not to put $\top$ need to be decorated with region variables.

I have two proposals for how to do this. The first is the one that is actually implemented in the prototype implementation. It is relatively simple to understand, but depends on the fact that FUN does not support updateable heap locations. Therefore I will describe a more powerful, but yet unimplemented, proposal in the next section.

But first, the solution implemented in the prototype. Because references are not implemented, all "permissions" of the skeleton typing will be `ro` (recall the "permission" concept from Section 5.1.6), and the region type system's subplacing concept collapses to a fully covariant notation. Therefore, nothing on at the left-hand side of a subplacing edge can prevent something on the right from being $\top$, and similarly, nothing on the right can prevent a place on the left from being $\bot$. Reasons-not-to-be-$\bot$ propagate in the direction of the control flow, while reasons-not-to-be-$\top$ propagate against the direction of the control flow.

This means that the set of reasons not to be $\top$ can be computed by a weakest-precondition algorithm, just like the distribution of reasons-for-region-variables-to-be-identical in Sections 7.3.2ff. In fact, in the prototype these two tasks are combined into one fixpoint iteration for both of them. Instead of dividing the places into two cases, counted and uncounted, the implementation in the prototype divides them into three cases: $\top$, counted and uncounted. Anything else than $\top$ means that an actual reason not to use $\top$ has propagated into the place.

Reasons not to be $\bot$ propagate in the other direction, so weakest preconditions will not help much here. Instead, the prototype uses a separate **strongest postconditions** fixpoint iteration to identify possible $\bot$'s before the weakest-precondition runs. This pass is conceptually similar to the algorithms that have already been presented – except for running in the opposite direction – but is actually simpler because it does not have to reason about equality between different places but simply propagates a binary attribute on each type vertex.

I will leave it to the reader to imagine the details of this pass.

7.3.7 Subplacing by propositional networks

With host languages that have primitives to update heap-allocated values, the region typing may need to contain `rw` marks. Then the easy subplacing strategy in the previous section does not work anymore, because under a `rw` mark subplacing is replaced with strict type matching, and then reasons to be $\top$ or $\bot$ can flow in both directions across the subplacing edge. In the general case we need stronger methods for subplacing.

A part of these stronger methods should be an attempt to change `rw`'s to `ro`'s in the skeleton typing. The syntax-driven methods for constructing skeleton typings in Section 7.2 will usually decorate all type vertices that represent an updatable type according to the host language's type system. However, if at some point in the program it can be shown that the pointer will not actually be used for destructive updates its decoration can and should change to `ro`. An important special case is when the pointer will not be used at all anymore. Then changing its permission to `ro` will allow the decorations on the pointed-to type to be subplaced to $\top$ such that the pointer and what it points to can all be deallocated.

Minimizing the number of `rw`'s in the program invoves reasoning about "reasons not to be `ro`". It turns out that these propagate in mostly the same way as "reasons not to be $\top$, so I propose a single analysis that locate prospective $\top$ and $\bot$ locations simultaneously with changing `rw` to `ro`.

The analysis works by a techique that I call "propositional networks", which I originally developed for doing binding-time analysis for C [Makholm 1999, under the name "logic-based BTA"] and later used in a subphase of my TT-like region inference for Prolog [Makholm 2000b, Section 8.8.1], coincidentally for a purpose much related to the placement of `ro`'s.[4] Systematically it can be characterized as constraint solving over a simple Boolean domain, with constraints in the shape of Horn clauses. But that does not convey much of the underlying intution, so here is an introduction I like better:

The idea is to build a little formal theory[5] about how the given program can be annotated. For each type vertex in the program, the theory contains propositions meaning, "there must *not* be a $\top$ here" and "there must *not* be a $\bot$ here", such that each truth assignment for all propositions describes a (not necessarily well-) typing for the program. The also contain other auxiliary propositions (to be described in a moment), but the total number of propositions is finite.

A finite number of inference rules and axioms are then derived from the program, such that a truth assignment describes a correct typing if and only if it is a **model**, that is, is satisfies the axioms and inference rules. Considering a proposition to be true iff it is provable in the theory (which is decidable because there are only finitely many propositions and rules in the theory) yields a model[6] which clearly is the "least true" model – which again means that it describes the typing with the most $\top$'s and $\bot$'s.

This is how the formal theory is generated. First, for each vertex

$$\Pi(\pi) = \langle \cdots \mid \ldots, \nu_{ij}{:}\pi_{ij}, \ldots \mid \cdots \rangle_?$$

[4] I am reluctant to claim to be the inventor of the general technique, since it is so simple that it has surely been thought of a dozen times before I was born. The algorithmic core is stuff that logic-programming people eat for breakfast. However, simple and useful as it is, I have never seen other people use it for program analysis, at least not with the heterogenous statement sets and multiary constraints that I see as its strength.

[5] People have different expectations about what concepts the phrase "formal theory" entails. We use the term here in the minimal sense of Mendelson [1997, Section 1.4], where all that is required is some set of "propositions" (which Mendelson calls "well-formed formulas") and a way to deduce propositions from other propositions.

[6] This is true because we do not include implicit connections between the truth values of proposition. Compare, *e.g.*, with propositional logic which requires that the truth assignment of propositions such as $\mathcal{A}$ and $\neg\mathcal{A}$ are related in a particular way. In such theories, taking the provable propositions to be true does not necessarily yield a well-formed truth assignment, let alone a model.

in any type graph in the skeleton typing, create propositions reading

$$\text{“}\Pi(\pi) \text{ is not } \bot\text{”} \qquad \text{and} \qquad \text{“}\Pi(\pi) \text{ is not } \top\text{”}$$

(as already described). Furthermore, for each $\mathtt{i}$ and $\mathtt{j}$, create the proposition

$$\text{“}\Pi(\pi) \text{ has } \mathtt{rw} \text{ as } \nu_{\mathtt{ij}}\text{”}$$

Now compute a “global subplacing certificate” – the least subplacing certificate (Definition 5.10) that would be able to certify all of the subplacing relations in all of the *RTSUB* steps in the skeleton typing, if all permissions were imagined to be `ro`. (This is a formal way of saying “the set of all connections of the kind that are shown as dotted lines in the graph on page 168, except that we also consider subplacing steps in non-annotatable edges”). For each element of this certificate, $\big((\Pi,\pi),(\Pi',\pi')\big)$, where

$$\Pi(\pi) = \langle \cdots \mid \ldots, \nu_{\mathtt{ij}}{:}\pi_{\mathtt{ij}}, \ldots \mid \cdots \rangle_? \qquad \text{and} \qquad \Pi'(\pi') = \langle \cdots \mid \ldots, \nu'_{\mathtt{ij}}{:}\pi'_{\mathtt{ij}}, \ldots \mid \cdots \rangle_?,$$

create the proposition

$$\text{“}(\Pi,\pi) \equiv (\Pi',\pi')\text{”}$$

and the inference rules

$$\frac{\Pi(\pi) \text{ is not } \bot}{\Pi'(\pi') \text{ is not } \bot} \qquad \frac{\Pi'(\pi') \text{ is not } \bot \qquad (\Pi,\pi) \equiv (\Pi',\pi')}{\Pi(\pi) \text{ is not } \bot}$$

$$\frac{\Pi'(\pi') \text{ is not } \top}{\Pi(\pi) \text{ is not } \top} \qquad \frac{\Pi(\pi) \text{ is not } \top \qquad (\Pi,\pi) \equiv (\Pi',\pi')}{\Pi'(\pi') \text{ is not } \top}$$

and for each $\mathtt{i}$, $\mathtt{j}$:

$$\frac{\Pi'(\pi') \text{ has } \mathtt{rw} \text{ as } \nu_{\mathtt{ij}}}{\Pi(\pi) \text{ has } \mathtt{rw} \text{ as } \nu_{\mathtt{ij}}} \qquad \frac{\Pi(\pi) \text{ has } \mathtt{rw} \text{ as } \nu_{\mathtt{ij}} \qquad (\Pi,\pi) \equiv (\Pi',\pi')}{\Pi'(\pi') \text{ has } \mathtt{rw} \text{ as } \nu_{\mathtt{ij}}}$$

$$\frac{(\Pi,\pi) \equiv (\Pi',\pi')}{(\Pi,\pi_{\mathtt{ij}}) \equiv (\Pi',\pi'_{\mathtt{ij}})} \qquad \frac{\Pi'(\pi') \text{ has } \mathtt{rw} \text{ as } \nu_{\mathtt{ij}}}{(\Pi,\pi_{\mathtt{ij}}) \equiv (\Pi',\pi'_{\mathtt{ij}})}$$

For each non-subplacing rule that requires the places or permissions in different type vertices to be identical, generate inference rules that enforce this (or, more efficiently, represent the propositions that refer to the two vertices by the same object in the implementation).

For each cons operation in the program (to be typed by either *RTCONS* or *RTNOWHERE*), generate a

$$\frac{}{\Pi(\pi) \text{ is not } \bot}$$

axiom for the appropriate type vertex.

For each read or write operation in the program (to be typed by *RTREAD* pr *RTWRITE*), generate a

$$\frac{}{\Pi(\pi) \text{ is not } \top}$$

axiom for the appropriate type vertex (or vertices, if $\mathcal{T}(\mathtt{s})$ includes multiple possibilities). For write operations, also generate

$$\frac{}{\Pi(\pi) \text{ has } \texttt{rw} \text{ as } \nu_{il}}$$

for all relevant i's.

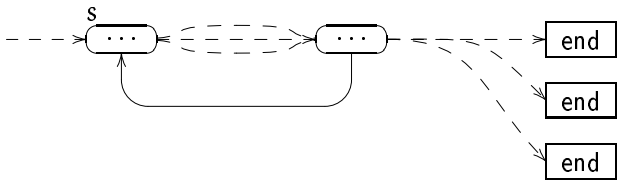
Once the propositions and rules have all been generated it is a simple matter[7] to compute which propositions are provable, starting with the axioms and iteratively adding propositions that are immediate consequences of the already-proved propositions. Then the assignment of $\top$'s and $\bot$'s can be derived from this set of provable propositions.

This analysis can be completed in time and space linear in the size of the skeleton typing. It can be optimized by not generating "$\Pi(\pi)$ has rw as ν_{ij}" propositions for permissions that are already known to be ro in the original skeleton typing (supposing that the skeleton typing itself is correct, these propositions will not be provable), and by generating "$(\Pi, \pi) \equiv (\Pi', \pi')$" propositions (and the rules that have them as premises) lazily instead of all at once.

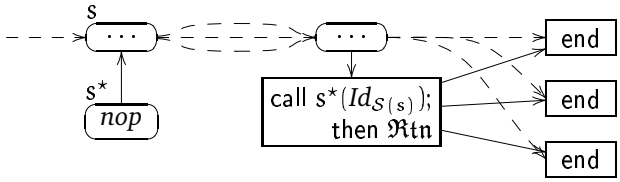
7.3.8 Loops in procedure bodies

The techniques described so far for the prototype all depend on the weakest-precondition algortihm described in Section 7.3.2, which itself depends on being able to traverse the chunks of each procedure body in topological order, which prevents it from working for uniform mutators with loops in procedure bodies. In general, of course, we want to be able to use our techiques for programming languages where intra-procedure loops are ubiquitous, so a solution to this has to be found.

One somewhat hacky solution to this would be to convert loops to tail recursion. Because UHL supports multiple entries to a procedure, we could identify a "back edge" in each loop and convert the loop from



to



where $s^\star$ is a fresh entry point to the procedure and $\mathfrak{Rtn}$ is a return map that makes the call a tail call by jumping to the very same end node that the recursive call returned from.

However, this is not actually a good strategy, because the region inference techniques just might, in pathological cases, decide that it would be a good idea with an agent that required region annotations on the edges from the tail call to the end nodes. Then the call would be a tail call no more, and it would have to

[7] An imperative linear-time algorithm is easily constructed [Makholm 1999].

be *implemented* as an ordinary call. (It could greatly confuse the programmer if his while loop caused a "stack overflow"!)

In fact, one would prefer to do the reverse transformation and convert tail-recursive calls[8] to back edges, precisely to prevent region operations from being inserted after the "returning" from the tail call. In the case of mutual tail recursion, this will entail merging the two (or more) procedures into one and unifying their `end` nodes (which will match in number and types if the calls are indeed tail calls). Therefore this transformation also depends on the ability to have more than one entry in a procedure.

The conclusion is that we do want to handle loops natively in the basic inference phase. This means that we need to leave the idea of considering the chunks in order – so let us toss out the idea completely and handle the chunks *all at once*! In practical terms this means that we have to maintain separate local union-find structures for all of the procedure's chunks simultaneously, in addition to the global union-find structure which all chunks share.

During the analysis, each class in a local union-find must be marked with a list of the incoming annotatable edges where it has LHS partners (together with references to the parters). When two classes are unified, their lists must be merged; if the same incoming edge appears in both of them, the unification propagates to the local union-find for the chunk's predecessor. A stack or queue of pending unifications can be used to do this. Likewise, when a class first gets marked with a reference to the global union-find (meaning that it refers to an uncounted variable), the list must be scanned and the assignment propagated to the predecessor chunks. Thus there must also be a list of pending local-to-global links.

Once all of the pending unifications and linkings have been resolved, region-variable names can be assigned to the entire procedure at once, and the region-operation selection algorithm from Section 7.3.1 ran for all of the annotatable edges. (But it is not necessary to do this until a fixpoint for the entire program has been found, of course).

Surprisingly, calls require little special attention. The region-variable identities in the existing guess for Π_3 and Π_2's become unifications in the call chunk's local union-find, and that is all that is necessary until the unifications for the entire procedure have settled down. Then step 3 of the algorithm in Section 7.3.4 can be executed to find out whether it is necessary to update the Π_3 guesses.

This strategy, which has a certain flavour of constraint solving even though there are no explict constraint syntax, still uses the fixpoint iteration of Section 7.3.5 to handle interprocedural connections. It is tempting to try to extend the "everything at once" principle to the interprocedural level and region-infer the entire uniform mutator with a single sound of constraint solving. This would nicely bound the running time of the basic region inference to O(the size of the skeleton typing).

Alas, this plan founders on the heuristics used when a caller needs a callee to return several pieces of its output in the same region. As shown by the example at the end of Section 7.3.5, the fixpoint iteration initially tries to solve this by

[8]Tail calls that are not recursive (even indirectly) are best left as ordinary UHL calls. There may be good reasons to need region operations after such a call, and it would only need bounded stack space to convert such calls to ordinary calls.

unifying two counted variables in the Π_3 guess. If, later, a compelling reason to make this counted variable uncounted arises, the two places are treated as separate in the next analysis of the callee. This would not be possible if the callee already had an active set of local and global union-finds – one would need to propagate the *negation* of an identity through its flowchart, and the usual union-find structure does not support that.

Doing a whole-program basic inference in a single pass might be possible if another heuristic was used – for example to prefer the solution with several uncounted variables from the beginning (but too many uncounted variables might make it hard to exploit the flexibility of the HMN model) or to make only a single uncounted variable when an already-unified counted variable needs to become uncounted (but that would mean that very few instances of aliasing-through-uncounted variables would be generated, and the result might risk becoming inferior to a TT-style inference, even for first-order programs).

More exprerimentation with different heuristics (which, luckily, can be done within the fixpoint-iteration paradigm) will be necessary to know wheter it is worthwhile to try to develop one of these ideas.

7.4 Region optimizations

As mentioned in the overview in the beginning of the chapter, the basic region inference cares only about deallocating memory as soon as it can. It uses the features of the agent programming language quite indiscriminately, not worrying about the cost of manipulating reference counts or passing region parameters around.

Keeping these internal costs of the agent under control is the job of **region optimizations** that are run after the basic region inferences. These are source-to-source program transformations of the agent programming language and do not care about the host language at all, working exclusively on the level of flowchart chunks rather than individual control states. They must work under the constraint that they must not extend the lifetimes of the of the mutator's memory blocks, at least not significantly.

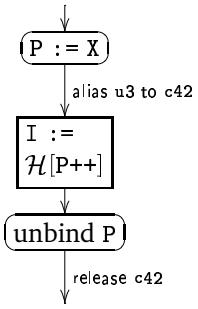
Of course, the region optimizations must preserve the *soundness* of the agent, because the principal goal of region inference is to produce a sound agent. In general it can be argued that each of the optimizations is *conservative* in the sense of Theorem 3.53 and thus preserves soundness.

For most of the optimizations it also holds, coincidentally, that they preserve typeability in a region type system such as the one from Chapter 5. This is good news for people who use region-based memory management to certify the memory safety of compiled code, but I will not have time to go into details about this.

This section briefly describe the region optimizations that are present in the prototype region inference. The descriptions will be somewhat sketchy, due to time constraints (see the Preface). Perhaps I will later have time to publish a fuller account of them. Readers are advised not to hold their breath.

7.4.1 Local alias propagation

The most important region optimization is **alias propagation**. It can be viewed as a kind of copy propagation for the agent programming language. Basically, the problem it solves is this: The basic region inference, since it works backwards, has a tendency to create lots of little short-lived aliases for regions whenever something is read from them. A common idiom is



for reading a non-pointer value from a cell in `u3`. The basic region inference constructs a typing where `P` refers to a fresh counted variable `c42` at the time of the read instruction. Only when it backs up to the preceding chunk does it learn that `P` is actually the same as `X` which is already knows to be in `u3`, and then emits an alias instruction to patch things up.

This is completely acceptable given the restricted goals of the basic region inference – the short-lived alias does not actually extend any lifetimes. But the reference-count manipulations it entails still have some cost at run time, in addition to bloating the agent.

Of course, in this simple example a quick cure would be to make sure that `P := X` was in the same chunk as the read operation. But the same problem appears on a larger scale, too. We need a more far-ranging solution.

The idea of the alias propagation is to try to defer alias instructions to as late as possible in the control flow. If eventually the alias ends up next to a release of the alias, the two operations dissolve each other. If the alias ends up next to a release of the source region-variable, it becomes a rename operation, which is cheaper to execute. In fact, the alias propagation will also try to propagate rename operations, in the hope that it meets another rename operation that it can be fused with (since renames are naturally transitive).

What can stop an alias from propagating further, except reaching a release operation to annihilate with? The canonical example is when it reaches a call chunk where one of the aliases is passed as a region parameter and the other is not. As shown by the *twolife* example in Section 2.4.1, this situation is the primary motivating example for having a alias operations at all, so it is OK that it gets blocked.

An alias operation can also be blocked by a call chunk even if both aliases are passed as region parameters. Global alias propagation, to be described shortly, can sometimes push the aliasing into the callee, but if there are other calls to the procedure (entry) in question which may pass two different regions in the same region parameters, the alias must happen before the call.

Similarly, an aliasing can be blocked on entry to a chunk that has another other incoming edge from a chunk where the two region variables are not nec-

essarily aliased. And if the aliasing eventually reaches a chunk with an end node, it will have to stop there – until we introduce global alias propagation in a little while.

The actual implementation, however, does not work in terms of moving actual alias operations around in the agent. Its working is more like a dual of basic region inference. Where basic region inference works by backwards-propagating reasons for region variables to be identical, alias propagation works by *forwards*-propagating reasons for region variables to be *different*.

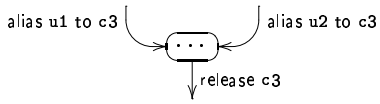
The algorithm computes, for each chunk, an **aliasing map** from "old" to "new" region variable names. The map needs not be transitive; two "old" region variable names that map to the same "new" name represent an aliasing operation that has been moved past the chunk.

As the aliasing map "moves along" an annotatable edge, the "old" region operations on it interact with the map, possibly changing (or annihilating) the operations as well as the map. new operations introduce new "new" names in the map; release operations disappear unless the "old" name in the operation is the only one that refers to its "new" name. alias and rename operations always disappear in favor or manipulating the map.

When a chunk has multiple incoming annotatable edges, a "joint" aliasing map need to be constructed by combining the maps that result from the old annotations on the incoming edges. In the joint map, two "old" variables map to different "new" names iff at least one of the incoming edges have them different. The difference between the incoming maps and the joint map is translated to alias and/or rename operations at the end if the incoming edges.

The prototype implementation of alias propagation works on acyclic flowcharts only. If the flowchart has loops, the incoming maps cannot always be available when a joint map is constructed. Then a fixpointing algorithm must be used, starting with the optimistic assumption that all region variables are aliased to each other.

There is still some room for improvement of this algorithm. For example, consider the situation



Here the creation (and subsequent release) of c3 is not necessary for the safety or soundness of the agent, because it does not change any memory-block lifetimes (whatever it keeps alive is kept alive by u1 or u2 anyway). But the current alias propagation is not strong enough to discover this.

7.4.2 Global alias propagation

It is not difficult to imagine an interprocedural variant of the local alias propagation. Aliasings that reach a end node can readily be recreated in the outgoing alias maps for the call chunks that receive returns from it. And alias maps for a procedure-entry chunks can be seeded as joint versions of the incoming alias maps in the call chunks that jump to it.

This has the consequence that it is not possible anymore to consider the chunks in order, even if procedure bodies are acyclic. The prototype implementation solves this by keeping a cache of gradually improved guesses for each procedure, like for the basic region inference in Section 7.3.5. If loops must be supported it may be cleaner to maintain a set of guesses for each chunk in the entire program, scheduling a chunk for reconsideration whenever one of its predecessors change.

This is well and simple as long as we stick to the convention that alias maps always map uncountable region variables to themselves (but countable region variables may map to uncountable ones, symbolizing a deferred "alias uN to cM" operation). This allows countable region parameters to be "fused" if all call sites agree that they will always be aliased anyway.

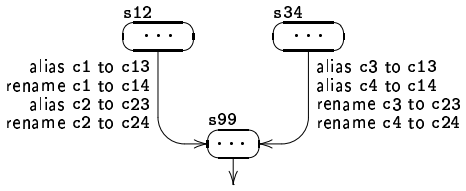
However, quite a can of worms opens if we want to handle uncounted region variables too. Ideally we could want to be able to

- Fuse uncounted parameters if they are always passed as aliases.
- Convert a counted parameter to an uncounted one if its source it always aliased to something that is *not* passed as a parameter (and is therefore known to be available as the source of an uncounted parameter). This will save an alias operation as well as a release s omewhere in the callee. Possibly it will also make the new uncounted parameter available for fusing with an existing one.
- Eliminate an output region if it is found to be aliased with an uncounted region variable at the end node. Then instead of passing a region back, the caller's name for the output parameter can be considered an alias for the source for the uncounted parameter.

This does not seem too difficult, but there are some subtle problems.

Fusion of uncounted parameters is easy enough, except that whenever a caller finds a difference between two uncounted parameters, they must be un-fused at *all* of the procedure's entry points, not just the one that particular call site leads to.

Conversion of counted region parameters to uncounted ones are easy too, as long as each procedure has only one entry point (as in the case in the prototype). With more entry points, things get difficult when new uncounted names may be invented. As an example, consider a procedure that begins with



where s12 and s34 are entry points and the region parameters c1, c2, c3, c4 all satisfy the conditions for being lifted to uncounted parameters. If four different uncounted names are used, c13, c14, c23, and c23 must all stay as separate counted variables. However, if c1 and c3 were both mapped to the same uncounted variable u13, the alias map for s99 could map c13 to u13. But that would prevent c14 from being treated in the same way. In general one would

Box 7.1—Fusing counted inputs with counted outputs

A really subtle trick (but actually occurring quite often in the benchmark program I have studied) in the prototype's global copy propagation is the ability to eliminate an output region in favor of an ordinary *counted* region parameter, if it can be shown that the returned region at runtime will always be identical to a particular input region. In this case the two counted regions (one input, one output) will be replaced with a single uncounted region parameter.

There are some conceptual problems with generalizing this from the single-entry, single-exit world of the prototype to the full UHL model. In particular, it will only work if the particular counted input region corresponds to at least one output region in *each* of the procedures `end` nodes, lest the caller-variable that got converted to an uncounted parameter get "orphaned" in the other return jumps. Thus mere ability for a procedure to throw an exception that does not contain any data (in which case the exceptional exit should return no regions either) will prevent it from working.

The feature should probably be considered superseded by the parameter lifting of Section 7.4.5

need some heuristic to chose between the various options here, and I do not have good ideas for one.

The last wish, eliminating an output region in favor of an uncounted parameter, is problematic even in the limited case treated by the prototype implementation (acyclic flowcharts, single-entry procedures). Consider a procedure with two uncounted input variables `u7` and `u9` and one output region `c13`. The optimistic initial assumption is that we will be able to fuse `u7` and `u9` to `u79` and eliminate `c13` in favor of `u79`. Now, while we are analyzing a caller, we find the `u7` and `u9` cannot be fused anyway, so we split them into their original names in the assumption. But what should the assumption about `c13` be now? Eliminating it in favor of `u79` is meaningless when `u79` does not exist anymore, and anything we can assume of it *might* be belied once we complete the analysis of the callee. This threatens the monotonicity of the assumptions and means that we cannot be sure that the alias propagation will terminate, which is bad.

The prototype implementation solves this by adding an exceptional alternative to an alias map during the analysis. This alternative means, "no information due to mention of obsolete constant parameters", and whenever a joint alias map must be computed, predecessors that are in the no-information state can be ignored. The no-information state can also be used as an initial assumption that for all alias-map positions. If any of these "holes" are still left after the analysis has reached a fixed point, it means that the chunks in question are actually unreachable; they can be removed from the mutator completely.

7.4.3 Correctness of alias propagation

Alias propagation preserves correctness, because the "new" agent is always conservative with respect to the "old" one. This can be shown formally by converting an arbitrary managed execution trace with the "new" agent to a managed execution trace with the "old" agent. In general this can be achieved by concatenating the `R` part of the configuration with the aliasing map that corresponds to the current control state.

Such a simulation also proves that alias propagation does not extend memory-

block lifetimes. Because the simulation does not change heap addresses it follows that whenever the "new" agent keeps something alive it is possible for the "old" agent to keep it alive, too.

Similarly, alias propagation can be seen to preserve typeability in the region type system: A typing for the "old" agent can be converted to a typing for the "new" agent by applying the relevant alias map to all region variables in each type graph. The prototype actually does does such rewriting on a large part of the typings during region optimizations. This has been identified as a source of bad performance of the region optimizer. It is probably faster to transform only the minimal parts of the typing necessary to recreate it (i.e., the annotations on the procedure entry and exit type graphs) and reconstruct the rest from this extract and the skeleton typing once all region optimizations have been been performed.

7.4.4 Region merging

The alias propagation does a good job of reducing the agent's use of different region variables that will just contain the same values at run time. But it leaves all new operations in the agent unchanged, except for the concrete name of the region variable that receives the new region.

There are certain fixed costs are associated with having a region. It takes time at run time to create and deallocate it, and to maintain reference counts. There is also a space cost, both for the management data and for yet-unallocated slack in the region's newest payload card. Therefore, we are interested in creating as few regions at runtime as possible without extending data lifetimes so much that it would offset the space saving in the region manager.

The central idea behind **region merging** is

> *If two run-time regions are deallocated at the same time, they really ought to have been the same region from the beginning (which would have resulted in the same data lifetimes but saved the overhead of one region).*

This principle was, as far as I am aware, an original contribution of my M.Sc. thesis [Makholm 2000b, Section 10.3.1]. I am not aware of it being mentioned in any of the previous work on automatic region inference, and I have it on reliable sources that the ML Kit does not do anything towards it[9].

It is harder to adhere strictly to this principle in an HMN-like agent programming language than in the TT-derived model I originally formulated it for. I the HMN model it is possible that it was not yet certain when the two regions in question were created, that they would eventually be deallocated at the same time. In that case, we need to keep them separate to be prepared for the possibility that they are *not* deallocated together. And after the regions have been created, there is nothing to do; the region manager's interface does not support the coalescing of two already existing regions into one (nor does its intended implementation).

Instead, we must settle for a weaker principle: If we can deduce, for a given new operation, that a specific other region variable will always be bound to a

[9]This is not surprising, because merging two regions *before* the Kit's storage-mode analysis may forfeit possibilities for resetting, and it is hard to see *after* the storage-mode analysis whether it is safe to merge them, especially if they may be reset within another function than the one that contains the letregion expression

region that will be deallocated at approximately the same time as the freshly created one, then the new should be replaced by an alias operation. We refer to this rewriting as **merging** the two regions.

If we take care to do the merging *before* alias propagation, the new alias instruction will propagate throughout the lifetime of the two regions (except in certain pathological cases), so we can get the benefit of less region-handle manipulation in the agent in addition to more efficient use of the region manager.

It is immediately that replacing a new with an alias will not cause a well-typed agent to become less well-typed. It takes some active reasoning (in the form of a direct simulation and an appeal to Theorem 3.53) to see that it also preserves soundness in general. Whether it also preserves the memory footprint of the agent depends on what we mean by "approximately the same time" and how we show that the two deallocation points are indeed that close.

Of course, one interpretation of "approximately the same time" would be "on the same annotatable edge". Another, more liberal but still fully safe would be that the two deallocations are close enough to each other if it is impossible for any (non-nowhere) cons operation to happen between them. Then no possibility for memory reuse would be lost by postponing deallocating the data in the first region until the merged region can be deallocated.

In the M.Sc. thesis I went a bit further and allowed merging two regions even if allocations happened between their deallocation points, as long as no *procedure calls* happened in between. The reasoning was that because the host language had no loops other than recursion, this would bound the number of reuse opportunities that could be forfeited because of the merging, and the merging in itself would be likely to free some overhead space, to there was a good chance that it would come out even, and in any case only a constant increase in memory footprint could result. The M.Sc. implementation also allowed merging of a region that was deallocated immediately before a procedure's return with one deallocated in the caller immediately after the call, under certain restrictions designed to avoid a region being merged across unboundedly many allocations on the way up from a deep recursion.

The HMN prototype is more liberal yet: It allows two regions to be merged if their deallocation points are not separated by any *entry* to a procedure – but returning from one is fair game. The rationale is that even if a deallocation *may* be postponed across unboundedly many allocations, the number of allocations will still not exceed a constant factor times the stack size – so the increase in memory footprint will not be more than linear in the stack (which is usually considered, by some sort of practical axiom, to be "small" by definition).

The common feature of these definition is that the identify certain points in the mutator as "checkpoints" across which deallocations must not be postponed. It is still an open question which strategy for selecting checkpoints is the most desirable, but the basic algorithms for merging regions should work with any arbitrary set of checkpoints.

The general procedure for doing region merges has three steps:

1. Compute, for each chunk, an approximation (from above) of the possible futures of the chunk's bound region variables – assuming each of them holds the only reference to some runtime region. Such a description could read something like
    a. It is possible to from here to have a future where first one or more

checkpoints are passed, then the regions currently bound to `c5` and `c2` are deallocated "approximately at the same time" (*i.e.*, without any intervening checkpoints), then some checkpoints are passed, then the region currently bound to `c7` is deallocated, then (...)

b. It is also possible to have a future where first some checkpoints are passed, then the region bound to `c5` is deallocated, then (...)

... (...)

During the computation, the intermediate results are parameterized with what happens to the output regions after the procedure returns; this prevents cross-contamination between the futures of different calls to the same procedure, and was found to be practically important for producing the intuitively expected amount of merges.

The future-descriptions, as can be seen from the example description above, tend to be quite lengthy, and it is easy to construct simple examples where the number of them grows exponentially in the number of region variables. I have plans for a reimplementation where the descriptions are instead structured like

a. It is possible in the future to reach a checkpoint where the regions currently bound to `c1`, `c2`, `c5`, and `c7` are still live.

b. It is possible in the future to reach a checkpoint where the regions currently bound to `c1` and `c7` are still live.

c. It is possible in the future to reach a checkpoint where the region currently bound to `c1` is still live.

... (...)

Such a description is simply a set of subsets of the bound region variables; it can be represented by a Binary Decision Diagram [Bryant 1992]. BDDs naturally supports most of the operations needed in the future approximation, but it remains to be seen whether they are robust enough to represent common sets of futures compactly.

2. Do a forward propagation through the agent to find out which region variables have "trustworthy" future information. A variable is trustworthy if we can be sure that the point at which the future information says it will be deallocated is in fact the right deallocation point, even in the presence of aliases.

   By definition the region variable the holds a newly created region is always trustworthy. After a

   $$\text{alias } \rho \text{ to } \rho^c$$

   operation ρ is trustworthy if ρ was trustworthy before and there is no future in which ρ^c lives longer than ρ. Conversely ρ^c will be trustworthy if ρ was trustworthy before and there is no future in which ρ lives longer than ρ^c.

   Both of ρ and ρ^c can be trustworthy if they happen to have the same future information.

3. Go through the mutator and consider each `new` operation. If there is another already bound region variable that is trustworthy and has the same future information as the newly created region, then replace the `new` with

an alias of the other variable. (Because the future information is identical, both region variables will stay trustworthy, so the order in which new operations are considered does not matter).

A new inside a procedure cannot directly be merged with a region variable in the caller of the procedure, even if the two regions have the same lifetime. To work around this, the region-merge phase in the prototype region inference has a special setting, "speculative merging" where it will add an artificial (counted) input region for each output region and pretend that the artificial shadow will be deallocated at the same time as the real output region. The hope is that the creation of the output region inside the procedure can be merged with the speculative parameter and the speculative parameter can be merged outside the call with a region variable that were not visible inside the callee.

However, it is possible that the inner merge will happen only on some paths through the procedure and the outer merge will be possible only at some call sites. Call sites *without* an outer merge will need to construct a fresh region to pass as the speculative parameter, and if the callee finds that it will not reach an inner merge anyway, it must release it explicitly. Thus there is a reason that speculative merges will lead to a lot of empty regions being created and then destroyed to no avail. Our (anecdotal) experience with the prototype is that this happens often enough that speculative merging is more likely to harm the (time and space) performance of the agent than to improve it.

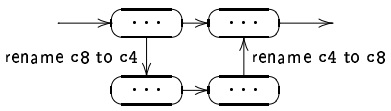
7.4.5 Parameter lifting

The **parameter lifting** optimization runs between region merging and alias propagation and builds on data from the first part of region merging. It goes over all procedures (all entrypoints) and whenever there is a counted region parameter that the future approximation says will always stay allocated for each checkpoint until the procedure returns, the parameter is marked as "long-lived".

Then all call chunks are inspected. Each time a long-lived parameter is passed, an alias for the actual value is created just before the call. The alias is released again after the call. Because the parameter is long-lived, this will not extend the region's lifetime by more than what is accepted for region merging. But the extra aliases will give the the alias propagation the opportunity to promote the parameter to an uncounted parameter, which is cheaper in general (and which will also cause the parameter lifting's new alias operations to disappear again.

7.4.6 The "unrename" transformation

Usually we should not worry much about whether the name of counted region variables match up from chunk to chunk. If they do not, rename operations will be inserted to fix the gap; such operations are assumed to be cheap. In general I have assumed that eventually the detailed management of region variables themselves will be handled by the existing implementation's register allocator, such that in situations like



it will be recognized that `c4` and `c8` are "really" the same variable and the renamings do not need to actually happen. Of course, specific algorithms for recognizing and preventing such code in agents could be developed, but the problem is really a standard register-allocation problem, and should be solved together with register allocation for the mutator.

The HMN prototype, however, does not have a register allocator as such, so it includes a optimization that targets a single case where the lack of a general solution is especially disturbing, namely when the return edge of a *tail call* is annotated with a rename operation.

Any region operation on the return edge will prevent the prototype's code generator from recognizing the call as a tail call, so the **unrename** optimization runs as the last phase in the region inference and tries to move rename operations earlier in the flowchart. They are stopped by almost every operation on the region variable they rename, but usually they will at least be able to move past the tail call, allowing it to stay tail.

Chapter 8

Conclusion

In this thesis I have developed and presented a general framework for reasoning about region-based memory management and region inference in a language-independent fashion – that is, with a minimum of dependence on the language and implementation that use the memory-management services of the region inference and the agent it generates.

The framework is based on an abstract flowchart language, the *universal host language* UHL, which has been designed to be sufficiently expressible to represent the low-level memory-use patterns of conventional implementations of a wide range of programming languages. To this language I have added a generic *agent programming language* (*i.e.*, a framework for region annotation) based on the "HMN model" that I have codeveloped with Fritz Henglein and Henning Niss. I have defined a generic *region type system* that allows the safety of agents for UHL to be certified. Finally, I have described general algorithms for *creating an agent mechanically* given a program in a host language that has a suitable UHL representation.

I have demonstrated, as a running example, how the REGFUN calculus and its associated region type system of Henglein et al. [2001] can be *systematically derived* from the general framework. But REGFUN was already known, and it is not difficult so specialize a generalization to the case from which it was originally generalized. The fire test would be to derive a working system of region-based memory management for a different setting than FUN, but I have not had time to actually do this. I can point, however, at the attempt I made for Prolog Makholm and Sagonas [2002], which however – as described in Section 7.1.1 – was not completely succesful[1] because the UHL model was not completely developed at the time of the experiment and because I lacked sufficient understanding of how the host language is typically used (as opposed to just how it works) and the existing body of analyses to help understand Prolog programs.

It is my hope that the general model will be useful for people who want to experiment with automatic region inference for other languages than the traditional ML-like languages described in the standard literature. Whether that will be the case remains to be seen, however, as does whether or not those

[1] That is, the *region inference* part of the experiment was not quite succesful. The *run-time* support for regions in the Prolog implementation – which was the subject matter of the conference paper Makholm and Sagonas [2002] – turned out to entirely satisfactory.

intended readers will be scared off by the sheer length of the description.

In addition to the UHL model itself and its associated examples and algorithms, I wish to direct special attention to the following new ideas in the thesis. Even if you are going to forget the details of the model, please at least try to hold on to one or two of the folllowing:

- The discovery that *optimal region annotations do not always exist* in a number of region-inference formalisms and for a natuaral definition of "optimal" (Sections 2.2.3, 2.3.1, and 2.4.4). This result was also published in Henglein et al. [2001].

  It is unknown and an interesting problem for further work whether this non-optimiality property still holds if we consider region-sound (or region-safe) agents for UHL in general rather than just the ones accepted by the region type system.
- The idea that (automatic) region-based memory management must be formulated for a *particular implementation of a programming language* rather than for a programming language in genereal (Section 1.4.1).
- The distinction between *region safety* – that a region-annotated program will not try to access deallocated data – and *region soundness* – that it exhibits the same extensional behavior as one would expect without memory reuse (Section 3.3).
- The idea that region annotations can be viewed as a sepearate coprocess, the *agent*, written in a special *agent programing language* (Section 1.1), and that the agent can be subject to program transformations and program analyses as a separate program, such as the region optimizations in Section 7.4.
- The identification of the *agent programming language*, the *region type system* and the *agent-construction algorithm* as the three conceptual parts of the region inference problem (Section 1.3).

References

Alexander Aiken, Manuel Fähndrich, and Raph Levien [1995]. Better static memory management: Improving region-based analysis of higher-order languages (extended abstract). In *Programming Language Design and Implementation (ACM SIGPLAN Conference, PLDI '95, La Jolla, CA, USA)*, special issue of *ACM SIGPLAN Notices*, 30(6):174–185.
⟨http://http.cs.berkeley.edu/~aiken/ftp/region.ps⟩. *(Cited on pp. 14, 22, 35, 36, and 82).*

Hassan Aït-Kaci [1991]. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, MA, USA, ISBN 0-262-01123-9 (hardcover), 0-262-69146-9 (paperback).
⟨http://www.isg.sfu.ca/~hak/documents/wam.html⟩. *(Cited on p. 156).*

Roberto M. Amadio and Luca Cardelli [1993]. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631.
⟨http://research.microsoft.com/Users/luca/Papers/SRT.A4.pdf⟩. *(Cited on p. 131).*

Anindya Banerjee, Nevin Heintze, and Jon G. Riecke [1999]. Region analysis and the polymorphic lambda calculus. In *Logic in Computer Science (14th Annual IEEE Symposium, LICS '99, Trento, Italy)*, pages 88–97. IEEE Computer Society. ⟨http://guinness.cs.stevens-tech.edu/~ab/Publications/fsharp.ps.gz⟩. *(Cited on pp. 29 and 82).*

Lars Birkedal and Mads Tofte [2001]. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392.
⟨http://www.it-c.dk/people/birkedal/papers/conria.ps.gz⟩. *(Cited on pp. 14, 22, and 30).*

Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner [1993]. The ML Kit. Technical Report DIKU-TR-93/14, Department of Computer Science, University of Copenhagen. *(Cited on p. 31).*

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup [1996]. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages (23rd ACM SIGPLAN-SIGACT Symposium, POPL '96, St. Petersburg Beach, FL, USA)*, pages 171–183. ACM Press, New York, NY, USA, ISBN 0-89791-769-3.
⟨http://www.diku.dk/users/tofte/publ/popl96.ps.gz⟩. *(Cited on pp. 14, 15, 22, 31, 32, 34, and 98).*

Michael Brandt and Fritz Henglein [1998]. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338. ⟨ftp://ftp.diku.dk/diku/semantics/papers/D-353.ps.gz⟩. *(Cited on p. 131).*

Randal E. Bryant [1992]. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318. *(Cited on p. 189).*

Cristiano Calcagno, Simon Helsen, and Peter Thiemann [2002]. Syntactic type soundness results for the region calculus. *Information & Computation*, 173(2):199–221. ⟨http://www.informatik.uni-freiburg.de/~helsen/calcagno-helsen-thiemann-iandc-2001.ps.gz⟩. *(Cited on pp. 29, 75, and 82).*

Cristiano Calcagno [2001]. Stratified operational semantics for safety and correctness of region calculus. In *Principles of Programming Languages (28th ACM SIGPLAN-SIGACT Symposium, POPL '01, London, UK)*, pages 155–165. ACM Press, New York, NY, USA, ISBN 1-58113-336-7. ⟨http://www.dcs.qmw.ac.uk/~ccris/ftp/popl01.ps⟩. *(Cited on pp. 29, 75, 82, and 83).*

Morten Voetmann Christiansen and Per Velschow [1998]. Region-based memory management in Java. Master's thesis, Department of Computer Science, University of Copenhagen. ⟨ftp://ftp.diku.dk/diku/semantics/papers/D-395.ps.gz⟩. *(Cited on pp. 78, 82, and 149).*

Karl Crary, David Walker, and Greg Morrisett [1999]. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (26th ACM SIGPLAN-SIGACT Symposium, POPL '99, San Antonio, Texas, US)*, pages 262–275. ACM Press, New York, NY, USA, ISBN 1-58113-095-3. ⟨http://simon.cs.cornell.edu/home/jgm/papers/capabilities.ps⟩. *(Cited on pp. 15 and 60).*

Silvano Dal Zilio and Andrew D. Gordon [2000]. Region analysis and a π-calculus with groups. In Mogens Nielsen and Branislav Rovan (eds), *Mathematical Foundations of Computer Science (International Symposium, MFCS '00, Bratislava, Slovakia)*, volume 1893 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-67901-4. ⟨http://www.research.microsoft.com/copyright/accept.asp?path=/users/adg/publications/regions-mfcs.ps&pub=15⟩. *(Cited on pp. 29 and 82).*

Robert DeLine and Manuel Fähndrich [2001]. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (ACM SIGPLAN Conference, PLDI '01, Snowbird, UT, USA)*, special issue of *ACM SIGPLAN Notices*, 36(5):59–69. ACM Press, New York, NY, USA, ISBN 1-58113-414-2. ⟨http://research.microsoft.com/vault/learn/papers/pldi01.pdf⟩. *(Cited on p. 15).*

Martin Elsman and Niels Hallenberg [1995]. An optimizing backend for the ML kit using a stack of regions. Student project, Department of Computer Science, University of Copenhagen. ⟨http://www.it.edu/people/mael/mypapers/backend.ps⟩. *(Cited on p. 22).*

Martin Elsman [1999]. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, Department of Computer Science, University of Copenhagen. Published as technical report DIKU-TR-99/3. ⟨http://www.itu.dk/research/mlkit/kit_general/phd.ps⟩. *(Cited on p. 22).*

Martin Gardner [1970]. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123. *(Cited on p. 25).*

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney [2002]. Region-based memory management in Cyclone. In PLDI [2002], pages 282–293. ⟨http://www.research.att.com/projects/cyclone/papers/cyclone-regions.pdf⟩. *(Cited on p. 15).*

Niels Hallenberg, Martin Elsman, and Mads Tofte [2002]. Combining region inference and garbage collection. In PLDI [2002], pages 141–152. *(Cited on p. 31).*

Niels Hallenberg [1999]. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen. ⟨http://www.itu.dk/people/nh/mypapers/root260699-a4.ps.gz⟩. *(Cited on p. 31).*

Simon Helsen and Peter Thiemann [2000]. Syntactic type soundness for the region calculus. In Alan Jeffrey (ed.), *Higher-Order Operational Techniques in Semantics (ACM Workshop, HOOTS '00)*, special issue of *Electronic Notes in Theoretical Computer Science*, 41(3):1–20. Elsevier. ⟨http://www.elsevier.nl/locate/entcs/volume41.html⟩. *(Cited on pp. 29, 75, and 82).*

Fritz Henglein, Henning Makholm, and Henning Niss [2001]. A direct approach to control-flow sensitive region-based memory management. In *Principles and Practice of Declarative Programming (3rd International ACM SIGPLAN Conference, PPDP '01, Firenze, Italy)*, pages 175–186. ACM Press, New York, NY, USA. ⟨http://www.diku.dk/~makholm/ppdp-henglein-makholm-niss.ps.gz⟩. *(Cited on pp. 1, 12, 14, 18, 20, 21, 25, 37, 43, 78, 159, 176, 192, 193, and 200).*

Fritz Henglein, Henning Makholm, and Henning Niss [2005?] $^{\text{Effect}}$types and region-based memory management. In Benjamin C. Pierce (ed.), *Advanced Topics in Types and Programming Languages*. MIT Press. To appear. *(Cited on pp. 1 and 29).*

Michael Hind [2001]. Pointer analysis: Haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering (ACM SIGPLAN-SIGSOFT Workshop, Snowbird, UT, USA)*, pages 54–61. ACM Press, New York, NY, USA, ISBN 1-58113-413-4. *(Cited on p. 146).*

Richard Kelsey, William Clinger, and Jonathan Rees (eds) [1998]. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1). ⟨http://www.schemers.org/Documents/Standards/R5RS/⟩. *(Cited on p. 18).*

Henning Makholm and Kostis Sagonas [2002]. On enabling the WAM with region support. In Peter J. Stuckey (ed.), *Logic Programming (18th International Conference, ICLP '02, Copenhagen, Denmark)*, volume 2401 of *Lecture*

Notes in Computer Science, pages 163–178. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-43930-7. *(Cited on pp. 1, 18, 20, 21, 153, 155, 156, 160, 176, and 192).*

Henning Makholm [1999]. Specializing C: an introduction to the principles behind C-Mix. Student project 99-1-2, Department of Computer Science, University of Copenhagen. To appear as a DIKU Technical Report in 2000. ⟨`http://www.diku.dk/~makholm/cmixintro.ps.gz`⟩. *(Cited on pp. 178 and 180).*

Henning Makholm [2000a]. On Jones-optimal specialization for strongly typed languages. In Walid Taha (ed.), *Semantics, Applications and Implementation of Program Generation (International Workshop, SAIG '00, Montréal, Canada)*, volume 1924 of *Lecture Notes in Computer Science*, pages 129–148. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-41054-6. *(Cited on p. 1).*

Henning Makholm [2000b]. Region-based memory management in Prolog. Master's thesis, Department of Computer Science, University of Copenhagen. ⟨`ftp://ftp.diku.dk/diku/semantics/papers/D-421.ps.gz`⟩. *(Cited on pp. 16, 17, 18, 31, 152, 153, 178, and 187).*

Henning Makholm [2000c]. A region-based memory manager for Prolog. In Antony Hosking (ed.), *International Symposium on Memory Management (ISMM '00, Minneapolis, MN, USA)*, special issue of *ACM SIGPLAN Notices*, 36(3):25–34. ACM Press. *(Cited on pp. 1, 17, and 153).*

Elliott Mendelson [1997]. *Introduction to Mathematical Logic*. Chapman & Hall, London, UK, fourth edition, ISBN 0-412-80830-7. *(Cited on p. 178).*

Robin Milner, Mads Tofte, Robert W. Harper, and David MacQueen [1997]. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, USA, ISBN 0-262-63181-4. *(Cited on pp. 17 and 116).*

Robin Milner [1978]. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17(3):348–375. *(Cited on p. 14).*

Greg Morrisett, David Walker, Karl Crary, and Neal Glew [1998]. From system F to typed assembly language. In *Principles of Programming Languages (25th ACM SIGPLAN-SIGACT Symposium, POPL '98, San Diego, CA, USA)*, pages 85–97. ACM Press, New York, NY, USA, ISBN 0-89791-979-3. ⟨`http://www.cs.cornell.edu/talc/papers/tal-popl.pdf`⟩. *(Cited on p. 60).*

Greg Morrisett, David Walker, Karl Crary, and Neal Glew [1999]. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569. ⟨`http://www.cs.cornell.edu/talc/papers/tal-toplas.pdf`⟩. *(Cited on p. 60).*

Christian Mossin [1997]. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, Department of Computer Science, University of Copenhagen. ⟨`ftp://ftp.diku.dk/diku/semantics/papers/D-330.ps.gz`⟩. *(Cited on p. 101).*

Henning Niss [2002]. *Regions are Imperative: Unscoped Regions and Control-Flow Sensitive Memory Management*. PhD thesis, Department of Computer Science, University of Copenhagen.

⟨http://www.diku.dk/~hniss/thesis/thesis-submitted.ps.gz⟩. *(Cited on pp. 38, 43, 69, 72, 78, 82, 122, and 135).*

PLDI [2002]. *Programming Language Design and Implementation (ACM SIGPLAN Conference, PLDI '02, Berlin, Germany)*, special issue of *ACM SIGPLAN Notices*, 37(5). ACM Press, New York, NY, USA, ISBN 1-58113-463-0. *(Cited on p. 196).*

John C. Reynolds [2002]. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (17th Annual IEEE Symposium, LICS '02, Copenhagen, Denmark)*, pages 55–74. IEEE Computer Society, Los Alamitos, CA, USA, ISBN 0-7695-1483-9. *(Cited on pp. 55 and 144).*

Walid Taha, Henning Makholm, and John Hughes [2001]. Tag elimination and jones-optimality. In Olivier Danvy and Andrzej Filinski (eds), *Programs as Data Objects (Second Symposium, PADO '01, Århus, Denmark)*, volume 2053 of *LNCS*, pages 257–275. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-42068-1. ⟨http://cs-www.cs.yale.edu/homes/taha/publications/preprints/pado00.dvi⟩. *(Cited on p. 1).*

Mads Tofte and Lars Birkedal [1998]. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767. ⟨http://www.itu.dk/research/mlkit/kit_general/toplas98.ps.gz⟩. *(Cited on pp. 14, 22, and 30).*

Mads Tofte and Jean-Pierre Talpin [1994]. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Principles of Programming Languages (21st ACM SIGPLAN-SIGACT Symposium, POPL '94, Portland, OR, USA)*, pages 188–201. ACM Press, New York, NY, USA, ISBN 0-89791-636-0. ⟨ftp://ftp.diku.dk/diku/semantics/papers/D-235.dvi.gz⟩. *(Cited on pp. 13, 14, 26, 27, 29, 75, and 83).*

Mads Tofte and Jean-Pierre Talpin [1997]. Region-based memory management. *Information and Computation*, 132(2):109–176. ⟨http://www.itu.dk/research/mlkit/kit2/infocomp97.ps⟩. *(Cited on pp. 26, 29, 75, and 83).*

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Hjfeld Olesen, Peter Sestoft, and Peter Bertelsen [1997]. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen. ⟨http://www.diku.dk/research-groups/topps/activities/kit2/diku97-12.a4.ps.gz⟩. *(Cited on p. 31).*

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen [1998]. Programming with regions in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, Department of Computer Science, University of Copenhagen. ⟨http://www.it-c.dk/research/mlkit/kit3/manual.ps.gz⟩. *(Cited on pp. 31 and 34).*

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft [2002]. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen. ⟨http://www.it.edu/research/mlkit/dist/mlkit-4.1.0.pdf⟩. *(Cited on p. 31).*

David Walker and Kevin Watkins [2001]. On regions and linear types. In *6th International Conference on Functional Programming (ICFP '01, Florence, Italy)*, pages 181–192. ACM Press, New York, NY, USA, ISBN 1-58113-415-0. ⟨`http://www.cs.princeton.edu/~dpw/papers/lr.pdf`⟩. *(Cited on pp. 15 and 28).*

David Walker, Karl Crary, and Greg Morrisett [2000]. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771. ⟨`http://www.cs.princeton.edu/~dpw/capabilities-toplas.pdf`⟩. *(Cited on pp. 15, 60, and 82).*

Daniel C. Wang [2001]. *Managing Memory with Types*. PhD thesis, Princeton University. TR-640-01. ⟨`http://ncstrl.cs.princeton.edu/expand.php?id=TR-640-01`⟩. *(Cited on pp. 78 and 82).*

David H. D. Warren [1983]. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A. *(Cited on p. 156).*

Dansk sammenfatning

Regionsbaseret lagerstyring er en strategi til automatisk styring af lagergenbrug i computerprogrammer, hvor den velkendte spildopsamling, som træffer beslutninger på kørselstidspunktet efter en analyse af lagerets indhold, erstattes af et specielt programmodul, en *agent*, som er konstrueret af oversætteren til at styre lager netop for det kørende program.

Processen at konstruere en agent automatisk på grundlag af det program den skal styre lager for, kaldes *regionsinferens*. Hidtil har resultater om metoder til regionsinferens kun foreligget for programmeringssprog i ML-familien, og disse metoder er ikke simple at overføre til andre familier af programmeringssprog.

I denne afhandling fremlægger jeg et generelt begrebsapparat for og en generel teori om regionsinferens som kan anvendes til mange forskellige programmeringssprog, og som kan hjælpe med at overføre teknikker fra ét programmeringssprog (eller én realisering af et programmeringssprog) til et andet.

Teorien er bygget op om et abstrakt rutediagramsprog, et *unverselt værtssprog*, som er konstueret til at have tilstrækkelig udtrykskraft til at repræsentere lageranvendelsesmønstre på lavt niveau for et bredt sprektrum af forskellige programmeringssprog. Jeg har udviklet et generelt, indlejret *agentprogrammeringssprog* for det universelle værtssprog baseret på "HMN-modellen" som jeg har udviklet sammen med Henning Niss og Fritz Henglein. Jeg har udviklet et *regionstypesystem* som gør det muligt at bevise at agenter i agentprogrammeringssproget er sikre. Endelig har jeg givet algoritmer til at *konsturere en agent mekanisk* udfra et program i et programmeringssprog der kan repræsenteres på passende vis i det universelle værtssprog.

Som løbende eksempel har jeg vist hvordan REGFUN-kalkulen fra Henglein et al. [2001] og dens tilhørende regionstypesystem kan *udledes systematisk* fra den generelle teori.