Combining Garbage Collection and Region Inference in The ML Kit

Niels Hallenberg (faxe@diku.dk)

Department of Computer Science
University of Copenhagen

Master's Thesis

June 25, 1999

Preface

The ML Kit is a Standard ML compiler where memory management is based on region inference. Region inference inserts, at compile time, allocation and deallocation directives into the target program such that no dynamic memory management is necessary (i.e., garbage collection). However, it is not always possible to insert allocation and deallocation directives such that all dead memory is reclaimed for reuse. It is therefore interesting to see if it is possible to combine region inference with garbage collection. The benefit should be that most of the allocated memory is recycled efficiently by region inference and the garbage collector concentrates on the memory not recycled by region inference. The goal is a system that recycles memory eagerly (i.e., uses less memory) and efficiently (i.e., uses less time on recycling memory).

We present a new backend for the ML Kit which has been designed with a garbage collector in mind. The backend in the ML Kit, as released in version 3, mainly consists of one large module compiling the intermediate lambda language into three address code. We have organized the new backend differently and split it into many smaller modules. Each module is then simpler to comprehend and debug.

We also present a garbage collector that works with regions. Region inference complicates the task of garbage collection in many ways. For instance, the heap is split in many smaller heaps called region pages and some of the heap allocated objects are actually allocated on the machine stack.

Acknowledgements

I would like to thank my advisor Mads Tofte for his inspiration, excellent supervision and ideas without which this project would not have been possible; many aspects of the compiler is the result of his guidance. I will also thank Martin Elsman for his interest in this project and all the discussions we have had about the ML Kit and this project.

Also thanks to the other members of the ML Kit group, that is, Peter Bertelsen, Lars Birkedal, Tommy Højfeld Olesen and Peter Sestoft. It is always a lot of fun to work in the ML Kit group.

Contents

I Overview 7

1 From Manual to Automatic Memory Management 8
- 1.1 Overview of The Compiler 12
- 1.2 Reading Directions 13
- 1.3 What Has Been Implemented 14
- 1.4 Performance 15
- 1.5 Notation 15

II The Backend 17

2 RegExp 18
- 2.1 The Region Inference Allocation Strategy 18
- 2.2 Region Inference 20
  - 2.2.1 Dynamic Semantics for RegExp 21
  - 2.2.2 Example Program 30
- 2.3 Multiplicity Inference 31
  - 2.3.1 Example Program 32
- 2.4 K-normalization 33
- 2.5 Storage Mode Analysis 33
  - 2.5.1 Example Program 35
- 2.6 Physical Size Inference and Drop Regions 36
- 2.7 Application Conversion 37
  - 2.7.1 Example Program 37
  - 2.7.2 Example Program - Tail Recursive 38

3 Closure Conversion 40
- 3.1 From Imperative Languages to Functional Languages 40
  - 3.1.1 Functions in Pascal 41
  - 3.1.2 Functions in C 43
  - 3.1.3 Functions in SML 45
  - 3.1.4 Uniform Representation of Functions 46
  - 3.1.5 Closure Representation 47

3.1.6 A Closure is Not Always Needed . . . 48
3.1.7 Region polymorphic functions . . . 50
3.1.8 Closure Explication . . . 52
3.2 Calculating the Need For Closures . . . 52
3.2.1 Variables and Constants . . . 53
3.2.2 Boxed Expressions . . . 54
3.2.3 Expressions . . . 54
3.3 ClosExp . . . 57
3.3.1 Call convention . . . 57
3.3.2 Functions . . . 59
3.3.3 Storage Modes . . . 64
3.3.4 Grammar for ClosExp . . . 64
3.3.5 Dynamic Semantics for ClosExp . . . 65
3.4 Closure conversion . . . 72
3.4.1 Preserve K-normal form . . . 72
3.4.2 Call Conventions . . . 73
3.4.3 Algorithm $\mathcal{C}$ . . . 74
3.5 Refinements of the Representation of Functions . . . 83

4 Linearization 86
4.1 Scope of Regions and Variables . . . 86
4.2 Return Convention . . . 88
4.3 LineStmt . . . 89
4.4 Example . . . 90
4.5 Algorithm $\mathcal{L}$ . . . 90
4.5.1 Top level declarations . . . 92
4.5.2 Simple Expressions . . . 92
4.5.3 Expressions . . . 92

5 Register Allocation 95
5.1 Revised Grammar . . . 97
5.1.1 Store Type . . . 98
5.1.2 Call Conventions . . . 100
5.1.3 Resolving Call Conventions . . . 101
5.1.4 Rewriting LineStmt . . . 102
5.2 Dummy Register Allocator . . . 107
5.3 Liveness Analysis . . . 107
5.3.1 Def and Use Sets for LineStmt . . . 109
5.4 Interference Graph . . . 109
5.4.1 Unnecessary Copy Related Interferences . . . 110
5.4.2 Move Related Nodes . . . 110
5.4.3 Building IG . . . 110
5.5 Spilling . . . 112
5.5.1 Statements Are Not Like Three Address Instructions . 113

5.6 Graph Coloring With Coalescing . . . 114
5.6.1 Conservative Coalescing . . . 115
5.6.2 Spill Priority . . . 115
5.6.3 Implementation . . . 115
5.7 Caller and Callee Save Registers . . . 115
5.8 Example . . . 117

6 Fetch And Flush **123**
6.1 Revised Grammar . . . 124
6.2 Set Of Variables To Flush . . . 126
6.2.1 Top Level Functions . . . 126
6.2.2 Statements . . . 126
6.3 Insert Flushes and `scope_reg` . . . 128
6.3.1 Top Level Functions . . . 128
6.3.2 Statements . . . 129
6.4 Insert Fetches . . . 131
6.4.1 Top Level Functions . . . 131
6.4.2 Statements . . . 131
6.5 Example . . . 133

7 Calculate Offsets **136**
7.1 Revised Storage Type . . . 136
7.2 Algorithm $\mathcal{CO}$ . . . 137
7.2.1 Top Level Functions . . . 137
7.2.2 Statements . . . 138
7.3 Example . . . 138

8 Substitution and Simplify **141**
8.1 Revised Grammar . . . 142
8.2 Algorithm $\mathcal{SS}$ . . . 142
8.2.1 Top Level Functions . . . 142
8.2.2 Statements . . . 144
8.3 Example . . . 146

9 Code generation **148**
9.1 Kit Abstract Machine . . . 148
9.1.1 Grammar for KAM . . . 149
9.2 Code Generation . . . 151
9.2.1 Constants and Access Types . . . 151
9.2.2 Allocation Points . . . 152
9.2.3 Set Storage Modes . . . 155
9.2.4 Call Convention . . . 156
9.2.5 Functions . . . 158
9.2.6 Applications . . . 159

9.2.7 Statements 163
9.3 Example 165

III The Garbage Collector 167

10 Basic Garbage Collection Algorithms 168
10.1 Fundamentals 168
10.2 Reference Counting 170
10.3 Mark–Sweep Collection 173
10.4 Mark-Compact Collection 174
10.5 Copying Garbage Collection 177
10.5.1 Simple Stop and Copy 178
10.5.2 Cheney's Algorithm 182
10.5.3 Generational Garbage Collection 184
10.6 Comparing The Algorithms 185

11 Garbage Collection of Regions. 188
11.1 Algorithm Using Recursion 188
11.2 Cheney's Algorithm and Regions 189
11.2.1 A Stack of Values 191
11.2.2 A Stack of Regions 191
11.3 A Revised Region and Region Page Descriptor 194
11.4 The Garbage Collection Algorithm 195
11.5 Finite Regions 197
11.5.1 Recursive Data Structures 198
11.6 Garbage Collect a Few Regions Only 199
11.7 Using Only One Global Region 201
11.8 When to Increase The Heap 201

12 Data Representation 203
12.1 Scalar and Pointer Records 204
12.2 Tagging Objects 205
12.3 Eliminating Fragmented Objects 208

13 Root–set and Descriptors 211
13.1 A Function Frame 212
13.2 Call Convention 213
13.3 Callee Save Registers 213
13.4 Frame Descriptors 215
13.5 Stack Layout 215
13.6 Implementation 218

IV Measurements **220**

14 Performance of the ML Kit backend **221**
14.1 Files 222
14.2 Benchmark Programs 222
14.3 Compilation Speed 224
14.4 Effect of Tagging 226

15 Performance of the Garbage Collector **229**
15.1 Cost of Garbage Collection 229
15.1.1 Simple Stop and Copy 231
15.1.2 Region Inference plus Simple Stop and Copy 232
15.1.3 Region Inference contra Garbage Collection 233
15.2 Heap Size 236
15.3 Bit Vector Size 240

16 Conclusion **242**
16.1 Contributions 242
16.2 The ML Kit 244
16.3 Future Work 244

Part I

Overview

Chapter 1

From Manual to Automatic Memory Management

The interaction between the programmer and the computer has gone through a major development since the first computers in the early 1940's. At that time, the programmers used mechanical switches to set the different bits and thereby program the machine.

Later on it was possible to use acronyms for the different machine instructions but explicit addresses and the internal workings of the computer still had to be considered when programming. With assembly languages it was possible to work with a more abstract address space in that the assembler was able to calculate explicit addresses from labeled addresses. It also gradually became simpler to load and execute programs.

As time went on, the computer systems grew considerably in size and they soon became difficult to implement. Programmers had to consider both the functionality of the large computer systems and all the low-level workings of the computer. The solution was to make the computer more abstract by reducing the number of non–application–related issues that the programmer had to consider when implementing an application.

This led to the first programming languages being developed in the 1950's with Algol and Fortran as prominent examples. The overall goal with developing programming languages has always been to make it easier for the programmer to implement different kinds of applications. Today we have a broad range of programming languages:

- **logical** languages as Prolog.
- **functional** languages as Miranda, Haskell, Standard ML, Scheme, Erlang and Lisp.
- **imperative** languages as Fortran, Algol, Pascal and C.
- **object-oriented** languages as Java, C++, and SmallTalk.

- **documentation** languages as TEX, LATEX, PostScript and HTML.
- **parallel** languages (i.e., for parallel computers) as Occam and pH.

The above list in not complete and more groups should be listed in order to include commercial popular languages such as Visual Basic. Each language has its own strenghts and weaknesses making it a good choice for a particular application. For instance, we use Standard ML to implement the ML Kit compiler, C for the runtime system and LATEX for the documentation.

Even though languages are so different, they all have to deal with the problem of memory management which is a main topic of this dissertation.

There are three fundamental different ways to deal with allocation and deallocation of memory [30].

1. The simplest is *static allocation* where each variable is bound to a fixed storage location during the evaluation of the program. This policy is used by Fortran and Occam compilers. There is no difference between a globally declared and a locally declared variable. At any time during evaluation there will be only one instance of each variable and the address of the variable does not change. It is a simple and fast allocation strategy with two main limitations: it is not possible to dynamically allocate data, that is, the size of all data has to be known at compile time. Building dynamically sized lists or tree structures is therefore not possible. The second limitation is recursion, which in general is impossible when all activations of a procedure share the same address space.

2. The most used strategy for block structured languages as C and Pascal is the *stack allocation* strategy. At each procedure invocation a new frame holding local variables for the called procedure is allocated on the stack. This strategy makes recursion possible together with the allocation of some data structures whose size is first known at run time. For instance, a local array of unknown size at compile time may be allocated at runtime on the stack where the size is passed as an argument to the function. With some limitations it is also possible to have procedures as values, see Chapter 3. The stack allocation strategy provides good functionality and recycles memory eagerly (i.e., memory is deallocated as soon as a procedure returns).

3. The indispensable but also error prone memory strategy is *heap allocation*. It is an unrestricted memory strategy where the programmer can create arbitrary sized data which outlives the procedure in which it was created. With heap allocation it is possible to model a wide range of data structures using for instance lists and trees. It is error

> prone because the programmer must allocate and deallocate objects of dynamic data structures manually. For instance, in Pascal the programmer must use `new` and `dispose` (in C `malloc` and `free`) to allocate and deallocate objects. If the heap is not managed correctly, then we may either get *space leaks* or *dangling references*. We have a space leak if we have data that is allocated on the heap but there exists no pointer pointing at the data and the data is never freed, that is, the data will not be referenced for the rest of the computation. A dangling reference is a reference pointing at a memory area whose contents has been deallocated but is not dead yet. If the pointer is dereferenced then the program will behave arbitrarily and will possibly crash. We have all experienced the above kinds of behavior where we for instance are told that the computer is out of system ressources (i.e., the computer is filled with *garbage*). The word garbage is used for data that is still allocated but is dead, that is, never used by the rest of the computation. An *eager* heap allocation strategy is one that relaims garbage as fast as possible and an *exhaustive* heap allocation strategy limits the amount of garbage as much as possible.

We postulate that many modern software projects suffer from bad memory behaviour and even unsafe memory systems. We believe that often the reason for a computer to crash is due to a memory error in the software system; either that the memory ressources are exhausted or that memory has been deallocated too soon. Another, disastrous memory error comes from the lack of array bound checking, that is, an array index accessing memory outside the array bounds. However, this is not directly related to memory management.

With the heap allocation strategy as in C++ and Pascal the programmer has to explicitly allocate and deallocate all data that does not follow the stack discipline (LIFO).

Large applications are normally split into smaller problems with local characteristics and properties. The smaller problems are then implemented by different programmers and put together later in the implementation phase. The manual heap allocation strategy makes the smaller parts less local if they share global data. When implementing a “local” problem the programmer has to figure out how the global data may be allocated and especially when it may be deallocated. Deallocating data in one sub system, *must* not be done if another sub system depends on it. This may lead to system failures that are hard to test for and may first be noticed after several years of usage; the dependency between the two sub systems may arise only in rare situations.

Likewise, it is important that data which is not going to be used by the application again actually is deallocated. Again it may be hard or impossible to justify in which sub system the data should be freed.

In general we believe that the manual heap allocation strategy increases the complexity of programs so much that it is worth looking at automatic heap allocation strategies. With automatic heap allocation strategies the programmer is not concerned about how to allocate and free data. The programmer just create and use data and then the compiler and/or run-time system automatically allocate and deallocate the data in an, hopefully, exhaustive, eager and safe manner, (e.g., the stack allocation strategy).

Another drawback of manual heap allocation is that memory directives are inserted before compile time. When the programmer inserts allocation and deallocation statements in the program she cannot make any assumptions about how the program executes. In one evaluation a data object may be dead at program point p and in another evaluation the same data object may not be dead at program point p. It is then necessary to leave the object allocated and this makes it, in practice, only possible for the programmer to approximate the most eager use of the heap.

In this project we investigate two different automatic allocation strategies. The first is *region inference* [10, 51, 53, 52], which is a *static* memory system. Region inference inserts explicit allocation and deallocation statements in the program when it is compiled. It is an automation of what the programmer does in a manual memory system. Because region inference is automatic and proved to be safe we know that if an object is deallocated, then the object will not be used by the rest of the computation.

Because region inference is resolved at compile time it has the same drawback as manual memory management, (i.e., that the set of allocated objects at a program point is only an approximation to the set of objects needed by the rest of the computation). The amount of garbage may be larger than what we appreciate.

Region inference is already implemented and used in the ML Kit compiler [49, 50]. Measurements show that the memory is in general used eagerly but for some programs the approximation is not good enough to get a satisfactory memory usage. We have a region profiler [49, 27], that identifies the space leaks and for the programs we have tried, the space leaks can be removed by changing the source program. Region inference is often sufficient to make memory efficient programs. However, there will not always be time to tune a program with respect to memory usage and we therefore need an optional tool to remove the extra garbage.

What we aim at is an automatic allocation strategy that is *dynamic*, that is, a strategy that decides what to deallocate when the program executes.

The problem with dynamic allocation strategies (i.e., *garbage collectors*) is that they interrupt program execution and traverse the allocated data structures to find the set of objects that may be reclaimed. This interruption may be annoying for the user, and in some situations disastrous. For example, a monitor program for a nuclear power station may likely not be interrupted.

In this project we combine region inference with garbage collection to achieve an automatic, exhaustive and eager allocation strategy. Region inference gives us a time efficient strategy without interruptions, and the garbage collector makes (hopefully) only small interruptions to reclaim the garbage that region inference does not free.

It is important that the garbage collector is optional and can be enabled and disabled at will. It is then possible for the programmer to tune a program to be memory efficient without the garbage collector. The program may then be used as a real–time application. On the other hand, it is seldom that a program may not be interrupted for small periods of time and then the garbage collector can remove the extra garbage. We believe that programs compiled with the combination of region inference and garbage collection has the potential to be faster than programs compiled for garbage collection only and this project is a step in that direction. This is reasonable because region inference deallocates most of the dead data efficiently and only a fraction of the data must be freed by the garbage collector, (i.e., the garbage collector has to be activated less often). However, it is not obvious that the combination is faster because region inference gives some limitations that garbage collectors normally do not have. For instance, we do not have one large heap but many, maybe 1000 smaller heaps. As we will see later it is not feasible to combine several garbage collection strategies as for instance generational and mark–sweep collectors which in practive gives excellent results when you have only a few heaps.

The ultimate goal of all memory systems is to have the following invariant:

> *At any time in the execution of program p, the set of objects allocated in memory is exatly the set of objects needed to finish the execution of program p.*

Although one cannot reach the ultimate goal, we can, at least, try to get as close as possible.

1.1 Overview of The Compiler

The backend compiler that we have implemented compiles a region annotated lambda language, presented in Chapter 2, into HP PA-RISC machine code. The task of compiling a lambda language into machine code is complicated and we have chosen to simplify the task by dividing the backend into several separate phases. Each phase adds or refines information to the compiled program such that we end up with an intermediate representation of the compiled program on which we can generate machine code directly.

We use two intermediate languages: ClosExp presented in Chapter 3 and LineStmt presented in Chapter 4. The LineStmt language is used in most of

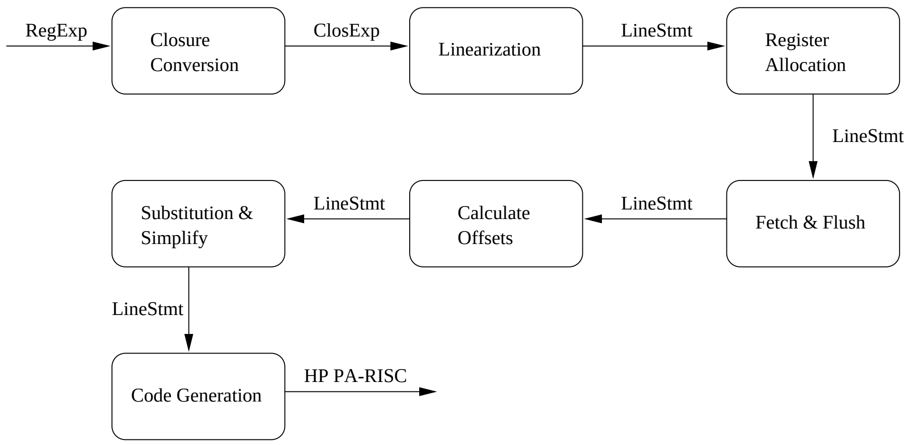

Figure 1.1: Overview of the backend compiler.

the phases. Some phases modify the LineStmt language such that new information can be added. However, the changes are minimal and we believe it would clutter the presentation if we invented a new language for each phase. The implementation also uses ClosExp and LineStmt modified to include all constructs necessary to compile Standard ML. The implementation follows the presentation closely.

Figure 1.1 shows the phases in the compiler and the intermediate language they work on.

The closure conversion phase lifts all functions to top level and rewrites the term such that no function has free variables. The linearization phase converts the expression like language ClosExp into the language LineStmt, in which each function is a sequence of statements. The register allocator adds register mapping information to LineStmt, that is, each variable is either mapped to a machine register or spilled on the stack. The fetch and flush phase inserts fetch and flush statements such that caller and callee save registers are preserved accross function calls. The calculate offsets phase assigns stack offsets to values that are stored on the stack, that is, spilled variables and regions. The substitution and simplify phase rewrites the LineStmt program such that the code generator can generate code for each statement by looking at the statement only.

1.2 Reading Directions

We have included a lot of information in the dissertation and several chapters can be skipped depending on your interests.

Chapter 2 presents the region inference allocation strategy and the intermediate language RegExp being input to our backend compiler. We also give a dynamic semantics for RegExp.

Chapter 3 starts with a discussion about functions in imperative and functional languages and the problems involved in compiling higher order functions. Section 3.3 presents the ClosExp language and the closure conversion algorithm is developed in Section 3.4.

The LineStmt language and linearization algorithm is presented in Chapter 4 and should probably not be skipped because LineStmt is used in the phases after linearization.

Chapter 5 presents the register allocator that is based on graph coloring. If you are fimiliar with graph coloring then you can skip Chapter 5. However, you should probably read Section 5.1 that introduces register allocation information into LineStmt.

Insertion of fetch and flush statements is mostly standard and can be skipped. It may be necessary though, to read Section 6.1 where fetch and flush statements are added to LineStmt.

Chapter 7 can be skipped if you are not interested in the layout of function frames. Chapter 8 rewrites the LineStmt program such that code generation is easier and should probably be read if you intent to read about code generation. The first few sections in the chapter on code generation show how code is generated for allocating in regions and is specific to the ML Kit. Section 9.2.4 – 9.2.6 presents the code generated for functions and applications.

We have four chapters on garbage collection. Chapter 10 is a small survey on garbage collection techniques and can be skipped. Chapter 11 develops the garbage collector for regions. Chapter 12 discusses tagging and is specific to the ML Kit. Chapter 13 discusses the problem of finding the root–set.

Assessments are shown in Chapter 14 and 15.

1.3 What Has Been Implemented

We felt it important to make a complete presentation of the ideas behind the backend compiler. First of all it was possible to evaluate all the ideas and make sure that they were implementable. The presentation greatly reduced the time necessary to do the implementation because most of the problems were already solved.

The presentation left us with limited time for the implementation and it has not been possible to implement the register allocator described in Chapter 5. What has been implemented is the critical parts of the backend compiler and an initial garbage collector.

The backend compiler compiles all parts of the Standard ML basis library and the test suite supplied with the ML Kit. All variables are spilled on the stack.

Martin Elsman has started the implementation of the register allocator as described in Chapter 5. Algorithm $\mathcal{F}$ described in Chapter 3 has been implemented but not tested. The algorithm looks for functions that can be implemented more efficiently without closures and is an optimization only. Also, phases that perform un–curring and un–boxing of functions have not been implemented (and not described in this dissatation). They are required to take full advantage of the ability of functions to take multiple arguments and return multiple results.

We have, so far, used a little more than two month on the backend compiler and two weeks on the garbage collector. We believe it takes another month or two to complete the implementation.

1.4 Performance

Many factors influence the compiler techniques used when compiling with both region inference and garbage collection enabled contra region inference or garbage collection only. For instance, using region inference only makes it possible to skip tags on values which is an important performance gain. Comparing systems using region inference and systems using garbage collection only is difficult.

We do not compare our implementation with other Standard ML compilers because the register allocator has not been implemented and the results would therefore be uninteresting. However, we do evaluate the compiler and garbage collector. For instance, the effect of tagging and garbage collection with and without region inference can be found in Chapter 14 and 15.

1.5 Notation

In this section we make some remarks about the notation used in the presentation.

Let A and B be sets. Then $A \setminus B$ is the set of elements in A that are not in B. The empty set is written $\emptyset$ or $\{\}$. The intersection of two sets A and B is written $A \cap B$ and the union of two sets A and B is written $A \cup B$. The power set of A is written $\mathcal{P}(A)$.

A function f with domain D and range R is written $f : D \to R$. A function $f : D \to R$ is a *partial* function iff f is defined for a subset of the domain only, that is, the set of partial functions for f is: $\bigcup\{E \to R | E \subseteq D\}$. We write a function $f : D \to R$ as a relation: $f \subseteq D \times R$: $f(d) = r$ iff $(d, r) \in D \times R$. The pair (d, r) is also written $d \mapsto r$. The domain of a function f is written $\mathbf{dom}(f)$ and the range is written $\mathbf{ran}(f)$. Two functions f and g are added as follows: $(f + g)(x) =$ if $x \in \mathbf{dom}(g)$ then $g(x)$ else $f(x)$. For instance $\{0 \mapsto 1, 42 \mapsto 2\} + \{42 \mapsto 3, 3 \mapsto 3\} = \{0 \mapsto 1, 42 \mapsto 3, 3 \mapsto 3\}$ (i.e., the function with domain $\{0,42,3\}$ and range $\{1,3\}$).

Function $f : D \to R$ restricted to domain D', written $f|_{D'}$ is the function $\{(x, y)|x \in D \cap D' \wedge f(x) = y\}$. Given a function $f : D \to R$, then we write $f \setminus D'$ for the function $f|_{(D \setminus D')}$.

We use "iff" for "if and only if".

An expression e is *directly within* a function f if there is no lambda between f and e. For instance, the expression 2 is directly within the function $\lambda x.2 + 4$ but not directly within the function $\lambda x.\lambda y.x + y + 2$.

The set of natural numbers $\{0, 1, 2, \dots\}$ is written $\mathbb{N}$.

We have several transformation functions that translates an intermediate language into the same slightly changed language. For instance, in Chapter 8 we have a function $\mathcal{SS}$ that translates the language LineStmt (introduced in Chapter 4) into a version of LineStmt where variables are replaced with what is called access types. When specifying such a function, we take the liberty to use the same set for both domain and range. The function $\mathcal{SS}$ is thus specified as

$$\mathcal{SS} : \mathit{LineStmt} \times \ldots \to \mathit{LineStmt}$$

where *LineStmt* is the set containing statements in both versions of LineStmt.

Part II

The Backend

Chapter 2

RegExp

In this chapter we present the intermediate language RegExp[1] which is the language passed on from region inference and the region representation analyses to the backend of the compiler. By example, we show how region inference introduce regions in the input λ program producing a RegExp program. Then we show how the region representation analyses gradually refine the RegExp program in order to compile it into a machine with a conventional one dimensional address space and a number of word size registers. All analyses discussed in this chapter is the work of other people [10, 55, 49].

The language presented is only a limited version of the RegExp language used in the ML Kit.

We start with a discussion about the allocation strategy that region inference impose at runtime.

2.1 The Region Inference Allocation Strategy

Region inference combines the stack and heap allocation strategy using a *stack of regions* as shown in Figure 2.1. Regions are allocated and deallocated in LIFO order, that is, they follow the stack discipline.

At compile time, allocation and deallocation directives of regions are inserted into the source lambda code based on a type analysis called region inference [52, 51, 53]. All value creating expressions are also annotated with a region in which the value is to be stored.

We have two different kinds of regions: regions with logical size *finite* and regions with logical size *infinite*. The logical size of regions is inferred at compile time with an analysis called Multiplicity Inference [55, 10]. Regions of logical size finite hold only one value at runtime whereas infinite regions may hold an unbounded number of values.

Infinite regions will normally contain recursive datatypes such as lists which may have unbounded size.

[1] *Region Expression.*

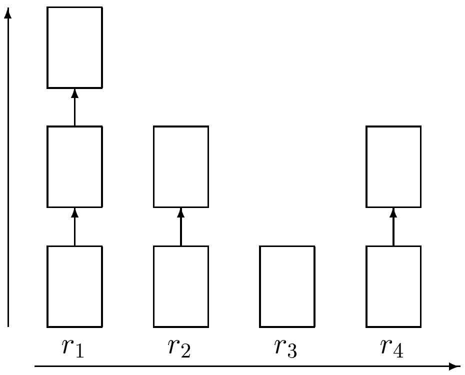

Figure 2.1: A stack of infinite regions. Region r_4 is the topmost region and is the first of the four regions to deallocate. Each region grows vertically. The stack of regions grows horizontally.

Each infinite region works as a restricted heap. You may either allocate at the top of the region or reset the region.

An infinite region consists of a list of *region pages* of fixed size. This framework gives some overhead when allocating objects; an extra range check is necessary. If the region page is full a new region page is allocated from a list of free region pages. Region pages are needed in order to map the one dimensional heap into the two dimensional region stack.

An infinite region r is reset by deallocating the region pages allocated to r except the first one. After a reset of r, the next element allocated in r is allocated at the bottom of the first region page. Infinite regions grow monotonically until they are either reset or deallocated.

Finite regions are split in those containing unboxed values of physical size one word and those containing boxed values of physical size one word or larger by an analysis called Physical Size Inference [10]. Finite regions of size one word are never allocated because the data can be stored in a machine register or spilled on the machine stack. Finite regions of size larger than one word are allocated on the machine stack. Allocating objects in finite regions is equivalent and equally fast as allocating in machine registers or on the machine stack. We even avoid allocating infinite regions containing unboxed values of size one word, that is, the values are stored in machine registers or on the machine stack [10, Section 5].

It is essential that we have only one region stack with both finite and infinite regions and that the region stack is part of the machine stack. Because finite regions have a fixed size they are easily allocated on the machine stack. We cannot allocate the region pages used to build infinite regions on the machine stack, however. Instead we allocate an *infinite region descrip-*

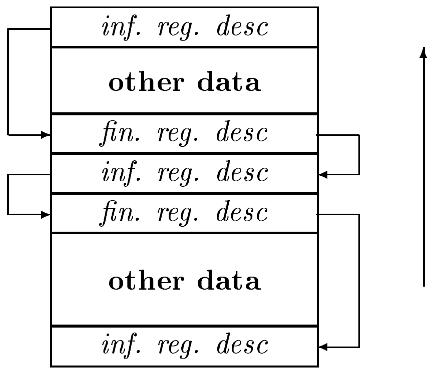

Figure 2.2: Finite and infinite regions are allocated on the machine stack together with other stack allocated data as for instance activation records. The stack grows upwards.

tor on the machine stack and thus make the infinite region visible on the region stack. We store a pointer to the first region page in the region descriptor. It is necessary to traverse the region stack independently of the machine stack in order to implement exceptions. Therefore we also use a *finite region descriptor* when allocating finite regions which together with the infinite region descriptors contain a pointer to the previous region descriptor, either finite or infinite. Starting with the top most region we can traverse all regions from top to bottom.[2] A snapshot of the machine stack is shown in Figure 2.2. The implementation of region descriptors is explained elsewhere [22].

2.2 Region Inference

The front end of the compiler produces a typed lambda term according to Milner's type discipline [17]. Region inference then produces a *region annotated* lambda term where region allocation directives are inserted.

We call the language produced by region inference RegExp. The semantic objects are shown in Figure 2.3. The grammar is shown in Figure 2.5. The grammar resembles a limited version of the core language of SML and share the same properties where applicable. For instance, evaluation is call by value and evaluation order is left to right.[3] We show annotations, such as *free* in function declarations, in example programs only when they are

[2]The discussion of finite region descriptors is not accurate because it turns out that they are not needed at all but it is technical.

[3]The ML Kit implements all features of Standard ML and RegExp in this presentation is a subset of RegExp used in the ML Kit.

$$
\begin{array}{rcl}
x, f, lv & \in & \mathit{LamVar} \\
i & \in & \mathit{Int} \\
\rho & \in & \mathit{RegVar} \\
 & & \mathit{Var} = \mathit{LamVar} \ \cup \ \mathit{RegVar} \\
\mathit{free} & \in & \mathit{FreeVar}\ \mathit{list} \\
\mathit{fv} & \in & \mathit{FreeVar} \ = \ \mathit{Var}
\end{array}
$$

Figure 2.3: The semantic objects used in RegExp

$$
\begin{array}{rcll}
\mathit{list} & ::= & [\,] & \text{empty list} \\
 & | & [x] & \text{singleton list} \\
 & | & [x_1, \ldots, x_n] & \text{list, } n > 1
\end{array}
$$

Figure 2.4: The index of each list element is unique. For instance, the two lists $[x_1, x_2]$ and $[x_2, x_1]$ are not the same.

important for the context.

The syntactic classes are *allocation directives*, a, *region binders*, b, *constants*, c, *boxed expressions*, be, *patterns*, pat, *binary operators*, bop and *expressions*, e. We let *RegExp* be the set of expressions (i.e., $e \in \mathit{RegExp}$). The syntactic constructs are discussed when we present the dynamic semantics in the next section.

Lambda variables, ranged over by x, f and lv are bound in either the `let` construct or in functions (λ and `letrec`). We have the constants: integers ranged over by i and the constant `nil` used to build lists. Region variables, ranged over by ρ, are bound in the `letregion` and `letrec` construct.

To simplify the discussion, we assume that the free variables of ordinary functions (λ) and for `letrec` bound functions have been computed beforehand. The free variables are represented by *free*. We use an *ordered list* for the free variables where the order of the elements in the list is important, see Figure 2.4.

2.2.1 Dynamic Semantics for RegExp

We present RegExp by describing the dynamic (operational) semantics of the language.[4] The semantic objects used are shown in Figure 2.6.

[4]It is a matter of opinion how close the dynamic semantics should resemble the actual evaluation of the compiled program. In a specification, as the definition of Standard ML, it may be done entirely abstract because it is implementation independent. In this presentation we use the dynamic semantics to describe essential points in how the compiled program works. We therefore make allocation of regions explicit together with how values are allocated in regions. This is essential aspects of the region allocation strategy. Unfortunately this approach makes the inference rules a bit more complicated but we believe it is worth the trouble.

$$
\begin{array}{rcl}
a & ::= & \texttt{at}\ \rho \\
b & ::= & \rho \\
c & ::= & i \mid \texttt{nil} \\
be & ::= & (e_1, \ldots, e_n) \mid \lambda^{\mathit{free}}\ \langle x_1, \ldots, x_n\rangle\ \texttt{=>}\ e \\
pat & ::= & c \mid \texttt{::}\ x \\
bop & ::= & \texttt{+},\ \texttt{-},\ \texttt{<},\ \ldots \\
e & ::= & x \\
& \mid & be\ a \\
& \mid & c \\
& \mid & \texttt{::}\ e \\
& \mid & e\ bop\ e \\
& \mid & \texttt{\#}n(e) \\
& \mid & \texttt{letrec}\ f^{\mathit{free}}\ \langle x_1, \ldots, x_n\rangle\ [b_1, \ldots, b_m]\ a\ \texttt{=}\ e\ \texttt{in}\ e\ \texttt{end} \\
& \mid & e\ e \\
& \mid & f\ \langle e_1, \ldots, e_n\rangle\ [a_1, \ldots, a_m]\ a \\
& \mid & \texttt{letregion}\ b\ \texttt{in}\ e\ \texttt{end} \\
& \mid & \texttt{let val}\ \langle x_1, \ldots, x_n\rangle\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\
& \mid & \texttt{case}\ e_1\ \texttt{of}\ pat\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ e_3 \\
& \mid & \langle e_1, \ldots, e_n\rangle
\end{array}
$$

Figure 2.5: The grammar for RegExp.

$$
\begin{array}{rcl}
st & \in & \mathit{Stack} = \mathit{RegDesc}\ \mathit{stack} \\
s & \in & \mathit{Store} = \mathit{RegDesc} \to \mathit{Reg} \\
rd & \in & \mathit{RegDesc} \\
r & \in & \mathit{Reg} = \mathit{Offset} \to \mathit{BoxedVal} \\
o & \in & \mathit{Offset} \\
bv & \in & \mathit{BoxedVal} = \mathit{Record} \cup \mathit{Clos} \cup \mathit{RegVec} \cup \mathit{SClos} \\
ubv & \in & \mathit{UnBoxedVal} = \mathit{Int} \cup \{\texttt{nil}\} \cup \texttt{::}(\mathit{Val}) \\
v & \in & \mathit{Val} = \mathit{Addr} \cup \mathit{UnBoxedVal} \\
rec & \in & \mathit{Record} = \mathit{Val} \times \ldots \times \mathit{Val} \\
\langle \vec{x}, e, E\rangle & \in & \mathit{Clos} = \mathit{LamVar} \times \cdots \times \mathit{LamVar} \times \mathit{RegExp} \times E \\
rv & \in & \mathit{RegVec} = \mathit{Reg} \times \cdots \times \mathit{Reg} \\
\langle \vec{x}, \vec{\rho}, (rd, o), e\rangle & \in & \mathit{LetrecEnv} = \mathit{Var} \times \cdots \times \mathit{Var} \times \mathit{Addr} \times \mathit{RegExp} \\
(rd, o) & \in & \mathit{Addr} = \mathit{RegDesc} \times \mathit{Offset} \\
& & \mathit{VE} = \mathit{Var} \to (\mathit{Val} \cup \mathit{LetrecEnv}) \\
& & \mathit{RE} = \mathit{RegVar} \to \mathit{RegDesc} \\
& & E = \mathit{SClos} = \mathit{VE} \times \mathit{RE}
\end{array}
$$

Figure 2.6: The semantic objects used in the dynamic semantics of RegExp

We have a *stack* (*Stack*) of *regions* ranged over by *st*. We may push and pop from the stack as follows:

$$(rd, st') = \mathbf{pop}(st)$$

and

$$st' = \mathbf{push}(st, rd).$$

The stack is essential for the implementation of exceptions. We discuss the implementation of exceptions as little as possible because it is quite technical and it is discussed elsewhere [38].

The *store* is a map from *region descriptors* to regions and together with the stack it represent the two-dimensional region stack. A region *Reg* is a block of memory (implemented as a list of region pages but the dynamic semantics does not make that explicit). A *region descriptor* (*RegDesc*) or *region name* is a unique name for a region. Allocating a region produces a fresh region and region descriptor. A region is formally a finite map from offsets (*Offset*) into *boxed values* (*BoxedVal*). Boxed values are (with a few exceptions) the values that are larger than one word and cannot be kept in word sized registers.

A new region is allocated by finding a region descriptor not in the domain of the store: $rd \notin \texttt{dom}(s)$, update the store: $s' = s + \{rd \mapsto \{\}\}$ and push the region descriptor on the stack: $st' = \mathbf{push}(st, rd)$.

Allocating a boxed value (bv) in a region r is done by finding an offset not in the domain of r: $o \notin \texttt{dom}(r)$ and then update the region: $r' = r + \{o \mapsto bv\}$. We write

$$s' = s + \{(rd, o) \mapsto bv\}$$

as an abbreviation for

$$s' = s + \{rd \mapsto \{s(rd) + \{o \mapsto bv\}\}\}$$

where $rd \in \mathbf{dom}(s)$.

A closure (*Clos*) contains the argument variables, body and environment with free variables for the function that the closure represent.

A *shared closure* (*SClos*) is used for `letrec` bound functions and contain the free variables of all functions bound in the `letrec`.[5] The `letrec` bound functions are known which is why we need to store a special environment *LetrecEnv* in the variable environment (*VE*) and not in memory. The environment *LetrecEnv* contains the argument variables and formal region variables. The address (rd, o) contains a shared closure.

[5]The grammar only allows one function to be defined in each `letrec` but in the implementation more than one function may be bound in the same `letrec` and they are all mutually recursive. Only one closure is used for all functions which is why we call it a shared closure.

$$\begin{array}{lll} seq & ::= & & \text{empty sequence} \\ & | & x & \text{singleton sequence} \\ & | & x_1, \ldots, x_n & \text{sequence, } n > 1 \end{array}$$

Figure 2.7: A sequence of elements.

An address ($Addr$) in the store (i.e., address in an infinite region) is uniquely determined by a region descriptor and an offset. We write

$$bv = s(rd, o)$$

as an abbreviation for

$$bv = (s(rd))(o)$$

where $rd \in \mathbf{dom}(s)$ and $o \in \mathbf{dom}(s(rd))$.

We may restrict the environment $E = (VE, RE)$ to a list of free variables, $free = [fv_1, \ldots, fv_n]$:

$$(VE, RE)|_{[fv_1, \ldots, fv_n]} = (VE|_{\{fv_1, \ldots, fv_n\}}, RE|_{\{fv_1, \ldots, fv_n\}})$$

We write $\vec{x}$ for a possibly empty sequence of objects x as defined in Figure 2.7.

The dynamic semantics is represented as a collection of inference rules of the form

$$st, s, E \vdash e \Rightarrow v, s'.$$

With stack st, store s and environment E the expression e evaluates to value v and a new store s'.

Constants $\boxed{\vdash ubv \Rightarrow ubv}$

We have the constants: integers (i) and `nil`. They all create an unboxed value.

$$\frac{}{\vdash i \Rightarrow i} \tag{2.1}$$

$$\frac{}{\vdash \texttt{nil} \Rightarrow \texttt{nil}} \tag{2.2}$$

Allocation Points $\boxed{s, (VE, RE) \vdash a \Rightarrow (rd, o), s'}$

For each allocation into a region ρ we need a new offset in the region.

$$\frac{RE(\rho) = rd \quad o \notin \texttt{dom}(s(rd))}{s, (VE, RE) \vdash \texttt{at}\ \rho \Rightarrow (rd, o), s} \tag{2.3}$$

Boxed Expressions $\boxed{st, s, E \vdash be \Rightarrow bv, s'}$

Records (tuples) are created with the expression

$$(e_1, \ldots, e_n) \texttt{ at } \rho$$

and the i'th component is selected by

$$\texttt{\#}i(e)$$

The record is stored in region ρ. In RegExp the first component is numbered 0 whereas the definition of Standard ML number the first component 1.

An ordinary function value is created by

$$\lambda^{free} \langle x_1, \ldots, x_n \rangle \texttt{ => } e \texttt{ at } \rho$$

where a closure for the function with all its free variables *free* is stored in region ρ. The function may take one or more arguments which will either be passed in machine registers or on the stack.

The inference rules for the boxed expressions are as follows:

$$\frac{st, s_{i-1}, E \vdash e_i \Rightarrow v_i, s_i \qquad i = 1, \ldots, n}{st, s_0, E \vdash (e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n), s_n} \tag{2.4}$$

$$\frac{}{st, s, E \vdash \lambda^{free} \langle x_1, \ldots, x_n \rangle \texttt{ => } e \Rightarrow \langle x_1, \ldots, x_n, e, E|_{free} \rangle, s} \tag{2.5}$$

Expressions $\boxed{st, s, E \ \vdash e \Rightarrow v, s'}$

A variable x is looked up in the variable environment

$$\frac{VE(x) = v}{st, s, (VE, RE) \vdash x \Rightarrow v, s} \tag{2.6}$$

A boxed expression is allocated in a region:

$$\frac{\begin{array}{c} st, s, E \vdash be \Rightarrow bv, s_1 \\ s_1, E \vdash a \Rightarrow (rd, o), s_2 \\ s' = s_2 + \{(rd, o) \mapsto bv\} \end{array}}{st, s, E \vdash be \ a \Rightarrow (rd, o), s'} \tag{2.7}$$

Constants are not allocated in a region:

$$\frac{\vdash c \Rightarrow ubv}{st, s, E \vdash c \Rightarrow ubv, s} \tag{2.8}$$

Binary operations work on integers only. We use the function $eval_{bop} : Int \times Int \rightarrow Int$ to evaluate the function bop given the two integer arguments. The values true and false are represented as 0 and 1 respectively.

$$\frac{\begin{array}{c} st, s, E \vdash e_1 \Rightarrow i_1, s_1 \\ st, s_1, E \vdash e_2 \Rightarrow i_2, s' \\ eval_{bop}(i_1, i_2) = i \end{array}}{st, s, E \vdash e_1 \ bop \ e_2 \Rightarrow i, s'} \tag{2.9}$$

Selection fetches the nth element from the record:

$$\frac{\begin{array}{c} st, s, E \vdash e \Rightarrow (rd, o), s' \\ s'(rd, o) = (v_0, \ldots, v_m) \quad 0 \leq n \leq m \end{array}}{st, s, E \vdash \texttt{\#}n(e) \Rightarrow v_n, s'} \tag{2.10}$$

Functions and Applications $\boxed{st, s, E \vdash e \Rightarrow v, s'}$

Mutually recursive functions are declared by the `letrec` construct:

$$\texttt{letrec}\ f^{free}\ \langle x_1, \ldots, x_n \rangle\ [b_1, \ldots, b_m]\ a \ = \ e_f\ \texttt{in}\ e\ \texttt{end}$$

The function f expects arguments $x_1, \ldots, x_n$ and a *region vector* to be passed when called. The arguments and region vector are passed in either machine registers or on the machine stack. Scope of the function f is the function itself e_f and the body e of the `letrec`.

The region vector binds the *formal* region variables $b_1, \ldots, b_m$ to *actual* region variables when the function is called. The actual region variables are passed in a region vector (record).

The function f has its free variables annotated as *free*. Because all `letrec` bound functions are known, we store only the environment in the closure and not the code e_f. The representation of closures are explained in Chapter 3. Note, that (rd, o) is the address of the shared closure.

$$\frac{\begin{array}{c} s, E \vdash a \Rightarrow (rd, o), s_1 \quad E = (VE, RE) \\ s_2 = s_1 + \{(rd, o) \mapsto \langle E|_{free} \rangle\} \\ VE' = VE + \{f \mapsto \langle x_1, \ldots, x_n, b_1, \ldots, b_m, (rd, o), e_f \rangle\} \\ st, s_2, (VE', RE) \vdash e \Rightarrow v, s' \end{array}}{\begin{array}{c} st, s, (VE, RE) \vdash \texttt{letrec}\ f^{free}\ \langle x_1, \ldots, x_n \rangle\ [b_1, \ldots, b_m]\ a \\ = \ e_f\ \texttt{in}\ e\ \texttt{end} \Rightarrow v, s' \end{array}} \tag{2.11}$$

Applications to ordinary and `letrec` bound functions are handled by the two rules:

$$\frac{\begin{array}{c} st, s, E \vdash e \Rightarrow (rd, o), s_0 \\ s_0(rd, o) = \langle x_1, \ldots, x_n, e_b, (VE_b, RE_b) \rangle \\ st, s_{i-1}, E \vdash e_i \Rightarrow v_i, s_i \quad i = 1, \ldots, n \\ st, s_n, (VE_b + \{x_i \mapsto v_i\}_{i=1,\ldots,n}, RE_b) \vdash e_b \Rightarrow v', s' \end{array}}{st, s, E \vdash e \ \langle e_1, \ldots, e_n \rangle \Rightarrow v', s'} \tag{2.12}$$

and

$$\frac{\begin{array}{c} VE(f) = \langle x_1, \ldots, x_n, b_1, \ldots, b_m, (rd_f, o_f), e_f \rangle \\ st, s, (VE, RE) \vdash \langle e_1, \ldots, e_n \rangle \Rightarrow \langle v_1, \ldots, v_n \rangle, s_n \\ s_n, (VE, RE) \vdash a \Rightarrow (rd, o), s'_n \\ rd_j = RE(\rho_j)_{j=1,\ldots,m} \quad s_f = s'_n + \{(rd, o) \mapsto \langle rd_1, \ldots, rd_m \rangle\} \\ (VE_f, RE_f) = s_f(rd_f, o_f) \\ VE' = VE_f + \{x_i \mapsto v_i\}_{i=1,\ldots,n} + \{f \mapsto VE(f)\} \\ RE' = RE_f + \{b_j \mapsto rd_j\}_{j=1,\ldots,m} \\ st, s_f, (VE', RE') \vdash e_f \Rightarrow v', s' \end{array}}{st, s, (VE, RE) \vdash f \ \langle e_1, \ldots, e_n \rangle \ [\rho_1, \ldots, \rho_m] \ a \Rightarrow v', s'} \tag{2.13}$$

The above rule may seem complicated but only elementary operations happen in each sub phrase or side condition. We look up f in the variable environment to get the letrec closure and then evaluate the arguments. The next two sub phrases allocate the region vector. The environment in which the body of f evaluates is then built. After closure conversion (Chapter 3) we separate the allocation of the region vector from this construct which simplifies the above rule.

A later phase (*Application Conversion*, Section 2.7) annotates a *call kind* at the application points in order to differentiate tail from non tail calls.

Letregion $\boxed{st, s, E \ \vdash e \Rightarrow v, s'}$

The `letregion` ρ `in` e `end` construct declares the region variable ρ. The region is allocated on the region stack and e is evaluated. The region (represented by ρ) is popped from the region stack after e is evaluated. The syntactic construction enforces region allocation to follow the stack discipline. Note, that region variables are also bound to regions when calling `letrec` bound functions, rule 2.13.

$$\frac{\begin{array}{c} rd \notin \mathtt{dom}(s) \\ s_1 = s + \{rd \mapsto \{\}\} \\ st_1 = \mathbf{push}(st, rd) \\ st_1, s_1, (VE, RE + \{\rho \mapsto rd\}) \vdash e \Rightarrow v, s_2 \\ (\_, st) = \mathbf{pop}(st') \\ s' = s_2 \setminus \{rd\} \end{array}}{st, s, (VE, RE) \vdash \mathtt{letregion} \ \rho \ \mathtt{in} \ e \ \mathtt{end} \Rightarrow v, s'} \tag{2.14}$$

In example programs we write

`letregion` $\rho_1, \ldots, \rho_n$ `in` e `end`

as an abbreviation for

```
letregion ρ1 in
  ...
    letregion ρn in e end
  ...
end
```

We make a distinction between region variables, region names (descriptors) and regions:

- a region variable (*RegVar*) is a syntactic object in the code as for instance *r37* in the program on page 30. A region variable can be bound to several regions at runtime.
- a region (*Reg*) is a piece of memory at runtime.
- a region name (*RegDesc*) is a name (number) which uniquely identify each region at runtime.

Declaring Lambda Variables $\boxed{st, s, E \vdash e \Rightarrow v, s'}$

Lambda variables are declared with the

`let val` $\langle x_1, \ldots, x_n \rangle$ `=` e_1 `in` e_2 `end`

construct. The expression e_1 is evaluated and the result is bound to $\langle x_1, \ldots, x_n \rangle$ and then the body e_2 is evaluated. The lambda variables have scope e_2.

$$\frac{\begin{array}{c} st, s, (VE, RE) \vdash e_1 \Rightarrow \langle v_1, \ldots, v_n \rangle, s_1 \\ st, s_1, (VE + \{x_i \mapsto v_i\}_{i=1,\ldots,n}, RE) \vdash e_2 \Rightarrow v', s' \end{array}}{st, s, (VE, RE) \vdash \texttt{let val } \langle x_1, \ldots, x_n \rangle \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end} \Rightarrow v', s'} \tag{2.15}$$

The unboxed record $\langle e_1, \ldots, e_n \rangle$ is evaluated by evaluating each element left to right.

$$\frac{st, s_{i-1}, E \vdash e_i \Rightarrow v_i, s_i \quad i = 1, \ldots, n}{st, s_0, E \vdash \langle e_1, \ldots, e_n \rangle \Rightarrow \langle v_1, \ldots, v_n \rangle, s_n} \tag{2.16}$$

We omit the brackets in examples if there is only one variable (i.e., ⟨x⟩ is written x).

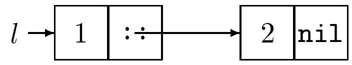

Figure 2.8: Representation of the list [1,2] in memory.

Case $\boxed{st, s, E \vdash e \Rightarrow v, s'}$

The case construct

$$\texttt{case}\ e_1\ \texttt{of}\ pat\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ e_3$$

evaluates e_1 and compares the result with the pattern *pat*. The default branch is chosen if no match is found, that is, e_3 is evaluated. There will always be a default branch, that is, cases with no default branch have been translated into cases *with* a default branch. The grammar supports a total of only two branches (i.e., conditionals) but the extension to n branches is trivial.

$$\frac{\begin{array}{c} st, s, (VE, RE) \vdash e_1 \Rightarrow \texttt{::}(v), s_1 \\ st, s_1, (VE + \{x \mapsto v\}, RE) \vdash e_2 \Rightarrow v', s' \end{array}}{st, s, (VE, RE) \vdash \texttt{case}\ e_1\ \texttt{of}\ \texttt{::}x\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ e_3 \Rightarrow v', s'} \tag{2.17}$$

The pair v is bound to x in the above rule.

$$\frac{\begin{array}{c} st, s, E \vdash e_1 \Rightarrow c, s_1 \\ st, s_1, E \vdash e_2 \Rightarrow v', s' \end{array}}{st, s, E \vdash \texttt{case}\ e_1\ \texttt{of}\ c\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ e_3 \Rightarrow v', s'} \tag{2.18}$$

$$\frac{\begin{array}{c} st, s, E \vdash e_1 \Rightarrow c', s_1 \qquad c' \neq c \\ st, s_1, E \vdash e_3 \Rightarrow v', s' \end{array}}{st, s, E \vdash \texttt{case}\ e_1\ \texttt{of}\ c\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ e_3 \Rightarrow v', s'} \tag{2.19}$$

$$\frac{\begin{array}{c} st, s, E \vdash e_1 \Rightarrow \texttt{nil}, s_1 \\ st, s_1, E \vdash e_3 \Rightarrow v', s' \end{array}}{st, s, E \vdash \texttt{case}\ e_1\ \texttt{of}\ \texttt{::}x\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ e_3 \Rightarrow v', s'} \tag{2.20}$$

Constructors $\boxed{st, s, E \vdash e \Rightarrow v, s'}$

The cons constructor :: takes a pair (e_1, e_2) where e_2 is a list and constructs a new list with e_1 as the first element. The list $l = \texttt{::}(1,\texttt{::}(2,\texttt{nil}))$ is stored as shown in Figure 2.8.

In the pair (v_1, v_2), v_1 is the head of the list and v_2 is the tail.

$$\frac{st, s, E \vdash e \Rightarrow (v_1, v_2), s'}{st, s, E \vdash \,::e \Rightarrow \,::(v_1, v_2), s'} \tag{2.21}$$

Note that :: has no runtime cost, that is, the pair $(v1, v2)$ is already constructed.[6]

2.2.2 Example Program

We show, by example, the result of a program being fed through the region inference analysis. Consider the SML program:

```
fun gen_list (0,acc) = acc
  | gen_list (n,acc) = gen_list(n-1,n::acc)
```

The result program from the region inference analysis is as follows:

```
letrec gen_list ⟨v515⟩ [r34, r35] at r1 =
  (case #0(v515) of
    0 => #1(v515)
    _ =>
      let
        val acc = #1(v515)
        val n = #0(v515)
      in
        letregion r37, r39
        in
          gen_list ⟨(n - 1,
                    let
                      val v41037 = (n,acc) at r35
                    in
                      :: ⟨v41037⟩
                    end) at r39⟩ [r39, r35] at r37
        end (*r37,r39*)
      end)
```

All boxed value creating expressions have `at` annotations and regions are introduced by the `letregion` construct.

The shared closure for *gen_list* is allocated in region *r1*. If the first component in the argument pair *v515* is 0 then we return the accumulator,

[6] We assume that lists are unboxed which is the case when the garbage collector is disabled. If the garbage collector is enabled then it is necessary to box lists and :: does have a runtime cost. Tagging is discussed in Chpater 12.

$$
\begin{array}{rcl}
b & ::= & \rho : m \\
m & ::= & 0 \mid 1 \mid \infty
\end{array}
$$

Figure 2.9: The grammar for RegExp after multiplicity inference. The multiplicity is denoted by m.

that is, the second component. In the default branch, we retrieve the two components from the argument and then recursively call *gen_list*. The region vector for *gen_list* is allocated in region *r37*; it contains the two region names denoted by *r39* and *r35*.

2.3 Multiplicity Inference

The multiplicity inference analysis solves the problem of how many values are allocated in a region [55, 10]. The analysis finds an upper bound of the number of allocations into each region. It turns out, in practice, that it is sufficient to consider the multiplicities 0, 1 and ∞. It is seldom that, for instance three values are allocated in a region. Regions containing lists get multiplicity ∞ because a list may contain an unbounded number of elements.

After multiplicity inference, every region binder b is annotated with a multiplicity m, see Figure 2.9. The `letregion` construct is then

$$\texttt{letregion}\ \rho : m\ \texttt{in}\ e\ \texttt{end}$$

and the `letrec` construct is

$$\texttt{letrec}\ f^{free}\ \langle x_1, \ldots, x_n \rangle\ [\rho_1 : m_1, \ldots, \rho_l : m_l]\ a \ = \ e\ \texttt{in}\ e\ \texttt{end}.$$

The regions have been split in *unbounded* regions called infinite regions and *write once* regions called finite regions. When allocating a region with `letregion` we know whether the region is finite and can be allocated on the machine stack or infinite and has to be implemented with region pages. For simplicity, we do not make the distinction in the dynamic semantics except that we record the multiplicity in the region environment:$RE = \mathit{RegVar} \rightarrow (\mathit{RegDesc} \times m)$, where the set of multiplicities *Mult* is ranged over by m (see Figure 2.6 on page 22).

The `letregion` rule 2.14 now becomes:

$$
\frac{\begin{array}{c}
rd \notin \texttt{dom}(s) \\
s_1 = s + \{rd \mapsto \{\}\} \\
st_1 = \mathbf{push}(st, rd) \\
st_1, s_1, (VE, RE + \{\rho \mapsto (rd, m)\}) \vdash e \Rightarrow v, s_2 \\
(\_, st) = \mathbf{pop}(st_1) \\
s' = s_2 \setminus \{rd\}
\end{array}}{st, s, (VE, RE) \vdash \texttt{letregion}\ \rho : m\ \texttt{in}\ e\ \texttt{end} \Rightarrow v, s'} \tag{2.22}
$$

The multiplicities on the formal region variables of a `letrec` bound function f is an upper bound on how many times the function f allocates a value in each region including calls to other functions. It may be the case that a formal region variable ρ has multiplicity 1 but an actual region has multiplicity ∞ in one call to f and multiplicity 1 in another call to f. This is called *multiplicity polymorphism* and each region ρ is at runtime annotated with its multiplicity. If ρ is either a free or formal region variable with multiplicity 1 then the multiplicity of ρ is looked up at runtime before allocating because the code to allocate in infinite and finite regions are different. If ρ is a formal region variable with multiplicity 1 and it is bound to an actual region with multiplicity ∞ then at runtime ρ has multiplicity ∞.

The `letrec` rule 2.11 does not change but rule 2.13 for letrec application does:

$$\frac{\begin{array}{c} \mathit{VE}(f) = \langle x_1, \ldots, x_n, (\rho'_1, m'), \ldots, (\rho'_l, m'_l), (\mathit{rd}_f, o_f), e_f \rangle \\ \mathit{st}, s, (\mathit{VE}, \mathit{RE}) \vdash \langle e_1, \ldots, e_n \rangle \Rightarrow \langle v_1, \ldots, v_n \rangle, s_n \\ \mathit{st}, s_n, (\mathit{VE}, \mathit{RE}) \vdash a \Rightarrow (\mathit{rd}, o), s'_n \\ (\mathit{rd}_j, m_j) = \mathit{RE}(\rho_j)_{j=1,\ldots,l} \quad s_f = s'_n + \{(\mathit{rd}, o) \mapsto \langle \mathit{rd}_1, \ldots, \mathit{rd}_l \rangle\} \\ (\mathit{VE}_f, \mathit{RE}_f) = s_f(\mathit{rd}_f, o_f) \\ \mathit{VE}' = \mathit{VE}_f + \{x_i \mapsto v_i\}_{i=1,\ldots,n} + \{f \mapsto \mathit{VE}(f)\} \\ \mathit{RE}' = \mathit{RE}_f + \{\rho'_j \mapsto (\mathit{rd}_j, m_j)\}_{j=1,\ldots,l} \\ \mathit{st}, s_f, (\mathit{VE}', \mathit{RE}') \vdash e_f \Rightarrow v', s' \end{array}}{\mathit{st}, s, (\mathit{VE}, \mathit{RE}) \vdash f\ \langle e_1, \ldots, e_n \rangle\ [\rho_1, \ldots, \rho_l]\ a \Rightarrow v', s'} \tag{2.23}$$

We store the multiplicity of each actual region together with the region descriptor in the region environment.

We also adjust rule 2.3 for allocation points:

$$\frac{\mathit{RE}(\rho) = (\mathit{rd}, m) \qquad o \notin \mathtt{dom}(s(\mathit{rd}))}{\mathit{st}, s, (\mathit{VE}, \mathit{RE}) \vdash \mathtt{at}\ \rho \Rightarrow (\mathit{rd}, o), s} \tag{2.24}$$

The dynamic semantics does not distinguish between finite and infinite regions when we allocate. However, if m is 1 we allocate into a finite region and if m is ∞ we allocate into an infinite region.

2.3.1 Example Program

After multiplicity inference our running example (page 30) becomes:

```
letrec gen_list ⟨v515⟩ [r34:0, r35:∞] at r1 =
  (case #0(v515) of
    0 => #1(v515)
    _ =>
      let
```

```
    val acc = #1(v515)
    val n = #0(v515)
  in
    letregion r37:1, r39:1
    in
      gen_list ⟨(n - 1,
                 let
                   val v41037 = (n,acc) at r35
                 in
                   :: ⟨v41037⟩
                 end) at r39⟩ [r39, r35] at r37
    end (*r37,r39*)
  end)
```

We have underlined the changes. We have one unbounded region and three write once regions. The unbounded region *r35* contains the list cells.

2.4 K-normalization

The K-normalization phase inserts extra `let` bindings such that every non atomic value is bound to a lambda variable [10]. The atomic values are the constants. The inference rules for the dynamic semantics are the same.

Although it is convenient to have K normalized code in the analyses, it makes even simple examples large because of an excessive number of let bindings. We therefore omit some let bindings in the presentation when they are not important for the context.

2.5 Storage Mode Analysis

After the storage mode analysis all allocation points are annotated with a storage mode [10].

$$\begin{array}{rcl} sma & ::= & \mathtt{attop} \mid \mathtt{atbot} \mid \mathtt{sat} \\ a & ::= & sma\ \rho \end{array}$$

Storage mode `attop` is used when a region contains live data at the allocation point and we therefore have to allocate at–top in the region. Storage mode `atbot` is used when the region does not contain live data and it may be reset before we allocate. Storage mode `sat` (*somewhere at*) is used when the decision is deferred to runtime. This happens within `letrec` bound functions where it is possible to store at–bot for some actual regions and necessary to store at–top for other actual regions. The set of *storage mode annotations* is *SMA* ranged over by *sma*.

The storage modes (either `atbot` or `attop`) for actual region names passed to a letrec bound function f are annotated on the region names at runtime before f is called. Inside f, the storage mode of region ρ is tested before allocating in ρ if the storage mode at the allocation point is `sat`. This is called *storage mode polymorphism.* It is not necessary to check the storage mode of a region if the storage mode at the allocation point is either `attop` or `atbot`. Note that region names never have storage mode `sat`; only allocation points do!

We extend the region environment to include the *storage mode* (*sm*): $RE = RegVar \to (RegDesc \times m \times sm)$, where $sm \in StorageMode = \{\texttt{attop}, \texttt{atbot}\}$.

In the `letregion` rule we use the storage mode `attop` which is chosen arbitrarily; it is only when calling a `letrec` bound function that a storage mode (either `attop` or `atbot`) is set in RE and used inside the `letrec` bound function.

$$\frac{\begin{array}{c} rd \notin \texttt{dom}(s) \\ s_1 = s + \{rd \mapsto \{\}\} \\ st_1 = \textbf{push}(st, rd) \\ st_1, s_1, (VE, RE + \{\rho \mapsto (rd, m, \texttt{attop})\}) \vdash e \Rightarrow v, s_2 \\ (\_, st) = \textbf{pop}(st_1) \\ s' = s_2 \setminus \{rd\} \end{array}}{st, s, (VE, RE) \vdash \texttt{letregion}\ \rho : m\ \texttt{in}\ e\ \texttt{end} \Rightarrow v, s'} \tag{2.25}$$

When calling a `letrec` bound function we store the storage modes found at the *actual* regions in the region environment:

$$\frac{\begin{array}{c} VE(f) = \langle x_1, \ldots, x_n, (\rho'_1, m'), \ldots, (\rho'_l, m'_l), (rd_f, o_f), e_f\rangle \\ st, s, (VE, RE) \vdash \langle e_1, \ldots, e_n\rangle \Rightarrow \langle v_1, \ldots, v_n\rangle, s_n \\ st, s_n, (VE, RE) \vdash a \Rightarrow (rd, o), s'_n \\ (rd_j, m_j, sm_j) = RE(\rho_j)_{j=1,\ldots,l} \quad s_f = s'_n + \{(rd, o) \mapsto \langle rd_1, \ldots, rd_l\rangle\} \\ (VE_f, RE_f) = s_f(rd_f, o_f) \\ VE' = VE_f + \{x_i \mapsto v_i\}_{i=1,\ldots,n} + \{f \mapsto VE(f)\} \\ RE' = RE_f + \{\rho'_j \mapsto (rd_j, m_j, \texttt{resolve\_sm}(sm_j, sma_j))\}_{j=1,\ldots,l} \\ st, s_f, (VE', RE') \vdash e_f \Rightarrow v', s' \end{array}}{st, s, (VE, RE) \vdash f\ \langle e_1, \ldots, e_n\rangle\ [sma_1\ \rho_1, \ldots, sma_l\ \rho_l]\ a \Rightarrow v', s'} \tag{2.26}$$

Note that m_j is the multiplicity of the actual region and is stored in the region environment. The `resolve_sm` function is necessary to get the right storage mode annotation (sm). If a storage mode sma_j is `atbot` then ρ_j is `letregion` bound and sma_j is recorded in RE. If sma_j is `attop` then ρ_j is either `letregion` bound or a formal region parameter and sma_j is recorded in RE; the storage mode is resolved even in the case that ρ_j is a formal region parameter. If sma_j is `sat` then ρ_j is a formal region parameter and we look up the storage mode in RE which will either be `attop` or `atbot`.

$$\begin{array}{l}
\texttt{resolve_sm}\text{: } \mathit{StorageMode} \times \mathit{SMA} \rightarrow \mathit{StorageMode} \\
\texttt{resolve_sm}(_, \texttt{atbot}) = \texttt{atbot} \\
\texttt{resolve_sm}(_, \texttt{attop}) = \texttt{attop} \\
\texttt{resolve_sm}(\texttt{atbot}, \texttt{sat}) = \texttt{atbot} \\
\texttt{resolve_sm}(\texttt{attop}, \texttt{sat}) = \texttt{attop}
\end{array}$$

The dynamic semantics for allocation points is extended to include resetting of regions depending on the storage mode at the allocation point and the storage mode recorded in the region environment. Again we use the function `resolve_sm`:

$$\frac{\begin{array}{c} RE(\rho) = (rd, m, smre) \\ \texttt{resolve\_sm}(smre, smap) = \texttt{attop} \\ o \notin \texttt{dom}(s(rd)) \end{array}}{st, s, (VE, RE) \vdash smap\ \rho \Rightarrow (rd, o), s} \tag{2.27}$$

$$\frac{\begin{array}{c} RE(\rho) = (rd, m, smre) \\ \texttt{resolve\_sm}(smre, smap) = \texttt{atbot} \\ s' = s + \{rd \mapsto \{\}\} \\ o \notin \texttt{dom}(s'(rd)) \end{array}}{st, s, (VE, RE) \vdash smap\ \rho \Rightarrow (rd, o), s'} \tag{2.28}$$

2.5.1 Example Program

After the storage mode analysis our running example (page 32) becomes:

```
letrec gen_list ⟨v515⟩ [r34:0, r35:∞] attop r1 =
  (case #0(v515) of
    0 => #1(v515)
    _ =>
      let
        val acc = #1(v515)
        val n = #0(v515)
      in
        letregion r37:1, r39:1
        in
          gen_list ⟨(n - 1,
                    let
                      val v41037 = (n,acc) attop r35
                    in
```

```
                    :: ⟨v41037⟩
                  end) atbot r39⟩ [atbot r39, sat r35] atbot r37
      end (*r37,r39*)
    end)
```

We have underlined the changes. We allocate `attop` in region *r35* containing the list. At the application to *gen_list* we have the allocation point `sat` *r35* inside the region vector. At that point *r35* is recorded in *RE* and mapped to either `atbot` or `attop` because *r35* is a formal region parameter to *gen_list*.

2.6 Physical Size Inference and Drop Regions

The physical size inference analysis infers the physical size of each finite region [10]. Region binders and multiplicities are now defined as

$$b ::= \rho : m \quad \text{and} \quad m ::= n \mid \infty$$

where the multiplicity may be an integer n or ∞. Allocating finite regions of size n is done by reserving n words on the stack. The dynamic semantics does not distinguish between allocating in finite and infinite regions.

A drop region phase is run before physical size inference which removes all formal and actual regions ρ from `letrec` bound functions and applications if nothing is ever written in ρ. In the example below we see that region *r34* has been removed; compare to the example program above.

```
letrec gen_list ⟨v515⟩ [_ r35:∞] attop r1 =
  (case #0(v515) of
    0 => #1(v515)
    _ =>
      let
        val acc = #1(v515)
        val n = #0(v515)
      in
        letregion r37:1, r39:2
        in
          gen_list ⟨(n - 1,
                  let
                    val v41037 = (n,acc) attop r35
                  in
                    :: ⟨v41037⟩
                  end) atbot r39⟩ [sat r35] atbot r37
        end (*r37,r39*)
```

$$
\begin{array}{rcl}
ck & ::= & \texttt{funjmp} \mid \texttt{funcall} \mid \texttt{fnjmp} \mid \texttt{fncall} \\
e & ::= & e_{ck}\ e \\
 & \mid & f_{ck}\ \langle e_1, \ldots, e_n \rangle\ [a_1, \ldots, a_m]\ a
\end{array}
$$

Figure 2.10: The grammar for RegExp after application conversion. We use *ck* for the call kind.

```
end)
```

Region *r39* has physical size two words containing the argument pair.

2.7 Application Conversion

The application conversion analysis annotates each ordinary and letrec application with a *call kind* [49, page 129].

We have four call kinds:

- tail calls to letrec bound functions have call kind `funjmp`. The requirements for a call to a letrec bound function to be a tail call are a bit tricky because what seems to be a tail call in the original SML program may turn out to be a non tail call after region inference. A region is needed for the region vector, so often a letregion construct is inserted around the call. We then have to return from the call and deallocate the region and the call is not a tail call anymore. However, special cases exists where it is possible to implement tail calls anyway; consult [49] for details.
- non tail calls to letrec bound functions have call kind `funcall`.
- tail calls to ordinary functions (defined with the λ construct) have call kind `fnjmp`.
- non tail calls to ordinary functions have call kind `fncall`.

The call kind is annotated at the application points as shown in Figure 2.10.

We do not make use of the call kind in the dynamic semantics so the inference rules are the same.

2.7.1 Example Program

After application conversion our example program is as follows:

```
letrec gen_list ⟨v515⟩ [r35:∞] attop r1 =
 (case #0(v515) of
   0 => #1(v515)
   _ =>
     let
       val acc = #1(v515)
       val n = #0(v515)
     in
       letregion r37:1, r39:2
       in
         gen_list_funcall ⟨(n - 1,
                           let
                             val v41037 = (n,acc) attop r35
                           in
                             :: ⟨v41037⟩
                           end) atbot r39⟩ [sat r35] atbot r37
       end (*r37,r39*)
     end)
```

This is the program that is passed on to the closure conversion phase which is the first transformation in the new backend for the ML Kit. Two regions *r39* and *r37* have to be deallocated after *gen_list* is called so the call is not a tail call.

2.7.2 Example Program - Tail Recursive

Consider the slightly changed example program below:

```
fun gen_list (p as (0,acc)) = p
  | gen_list       (n,acc)  = gen_list(n-1,n::acc)
```

The function *gen_list* has been turned into a region endomorphism [49, 8, 27] and the recursive call to *gen_list* is now a tail call:

```
letrec gen_list ⟨v512⟩ [r34:∞,r35:∞] attop r1 =
 (case #0(v512) of
   0 => v512
   _ =>
     let
       val acc = #1(v512)
       val n = #0(v512)
     in
       gen_list_funjmp ⟨(n - 1,
                         let
```

```
                              val v40882 = (n,acc) attop r35
                            in
                              :: ⟨v40882⟩
                            end) attop r34⟩
    end)
```

The region vector passed to *gen_list* is reused in the recursive call so we have not written it in the above example.

Chapter 3

Closure Conversion

An important difference in how a functional language like SML is compiled compared to traditional Pascal like languages is in the treatment of functions. In SML, functions may be nested and are first order values. A function as a first order value means that it can be passed around as any other value like numbers and strings. A function can be stored and later retrieved and applied in a context different from where it was created.

In this chapter we discuss how the implementation of SML functions using *closures* [33] allocated on the heap differs from Pascal like languages where *activation records* (*function frames*) [1, 6] allocated on the stack are sufficient.

In order to simplify the phases in the backend we implement a *closure conversion* algorithm [6, Chapter 15], which translates the RegExp language into a ClosExp language in which closures are explicit. A closure of a function contains the free variables of the function and inside its body, each free variable is accessed through the closure that is given as argument to the function. Thus, the free variables of a function have been turned into local variables to the function. Now that the functions have no free variables it is possible to lift the functions to top level, that is, to split the single large RegExp expression into a list of function declarations where the body of each function is one ClosExp expression.

3.1 From Imperative Languages to Functional Languages

Functions are not first order values in imperative languages such as Fortran and Pascal [57]. However, with some limitations, it is possible, in Pascal, to pass a function g as an argument to another function f [57, page 121]. The restriction is that g must be in scope when passed to f. It is not possible to store a function in a variable, thus, functions are not first class values in Pascal.

In the next two sections we explain how functions in Pascal and C are compiled using activation records on the machine stack. Then we treat SML functions in greater detail and explain how they are represented in the ML Kit.

3.1.1 Functions in Pascal

It is allowed to nest functions in Pascal. Lexical scoping rules apply where a function `h` declared inside another function `f` may access the local variables declared in the outer function `f`:

```
program scope(input,output);
  var a : integer; {(2)}

  function g(x : integer) : integer;
  begin
    g:=x+a
  end;

  procedure f();
    var a : integer; {(1)}

    procedure h();
    begin
      write('Result of g(1): ', g(a)); {(3)}
      writeln
    end;

  begin
    a := 1;
    h()
  end;

begin
  a := 41;
  f()
end.
```

Inside procedure `h` we access the local variable `a` defined in procedure `f` (`(1)`) and not the global declared variable `a` (`(2)`). This is implemented by storing the local variable `a` in the *activation record* for `f` and then store an *access link* [1, Chapter 7] in the activation record for `h`, which points at the activation record for `f`. The access link is stored in the activation record for `h` when `h` is called. Inside `h` we fetch variable `a` by dereferencing the access

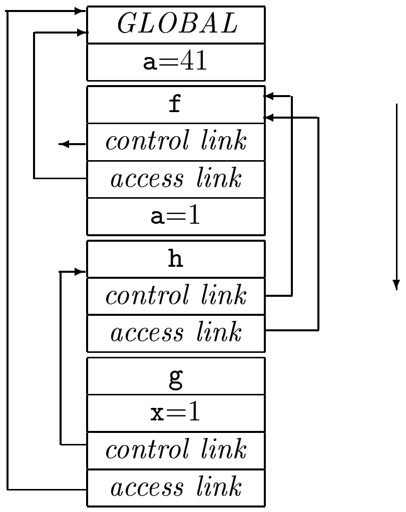

Figure 3.1: The control link points at the callers activation record. The access link of a function f, say, points at the activation record for the closest enclosing function of f. For instance, the closest enclosing activation record for `g` is the global data holding `a`=41 whereas the access link for `h` points at the activation record for `f`. We put the argument `x`=1 before the control link. The stack grows downward.

link once giving the activation record for `f`. This scheme works because a function never escape from the context (function) where it is defined. If function `h` could escape then `h` could be called in a context where function `f` is not activated and the local variable `a` would not be accessible.

When `g` is called (`(3)`) it is important that `g` uses the global declared variable `a` (`(2)`) and not the one declared inside `f`. This is again controlled by the access link stored in the activation record for `g`. An example stack is shown in Figure 3.1. For simplicity, we have put the global address space at the bottom of the stack.

Consider the following Pascal program where the two functions `add` and `mul` are passed as arguments to procedure `apply_f`.

```
program apply(input, output);

  procedure apply_f(function f(n : integer):integer,
                    arg: integer);
  begin
    write(f(arg));  {(3)}
  end; { apply_f }

  procedure p();
    var local : integer;
```

```
    function add(n : integer):integer;
    begin
      add:=local+n
    end; { add }

    function mul(n : integer):integer;
    begin
      mul:=local*n
    end; { mul }
  begin
    local:=41;
    apply_f(add,1); {(1)}
    apply_f(mul,0); {(2)}
    writeln
  end; { p }
begin
  p()
end. { apply }
```

The variable `local` is not accessible within procedure `apply_f` but it is accessible at the program points `(1)` and `(2)`. At the time we pass the functions `add` and `mul` to `apply_f` the variable `local` is allocated in the activation record for `p` and accessible. At program point `(1)` both the function `add` and an access link to the activation record for `p` is passed. When `add` is activated (through `f`) at program point `(3)`, an activation record for `add` is created and the access link stored is the one pointing at the activation record for `p` giving access to the variable `local`. So even though we have nested functions, it is possible to pass functions as arguments and still implement function calls on the stack using activation records. Figure 3.2 shows an example stack.

3.1.2 Functions in C

In C it is possible to take the address of a function g, store it in a variable, retrieve it from the variable and then pass it to a function, f, (i.e., functions can be used as first class values).

```
void *temp_f = NULL;

void apply_arg(int (*f)(int)) {
  printf("The value of succ(41): %d\n", (*f)(41));
  return;
}
```

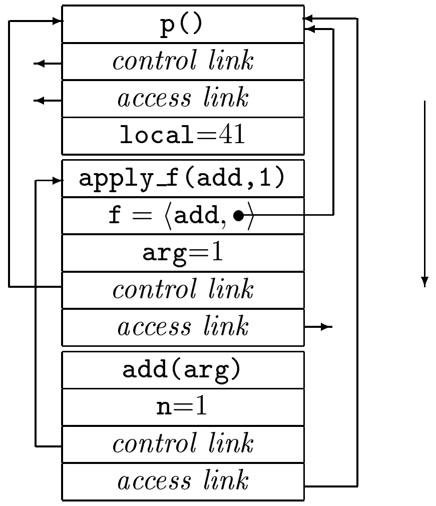

Figure 3.2: There are no global data so the access links of `p` and `apply_f` points at `nil`. When `add` is passed as argument to `apply_f` we pack the access link that should be used by `add` together with the code pointer for `add`: ⟨`add`, •⟩. When `add` is called we fetch the access link from the argument ⟨`add`, •⟩ and put it in the activation record for `add`. The stack grows downward.

```
void call_temp_f() {
  apply_arg(*(*(int (*)(int))temp_f));
  return;
}

int succ(int i) {
  return i+1;
}

void main() {
  temp_f = (void *)&succ;
  call_temp_f();
  return;
}
```

The function `succ` is passed as argument to function `apply_arg`. The type `(int (*f)(int))` is read: `f` is a pointer to a function with one argument of type `int` returning an `int` as result. When the function `f` is applied in the `printf` statement we first dereference `f` in order to get the function and then call `f` with 41 as argument.

All variables in a C function may be viewed as either *global* (defined at top level) or *local* (defined inside a function). Fetching global data is done through a *global data pointer*. To store local data each function is equipped with a stack allocated activation record (as for the Pascal programs above), and the record contains all local declared data. Local data is fetched through the stack pointer. Access links are not necessary since data is either local to the function itself or global. The fact that functions may not be nested in C but are always declared at top level makes it possible to implement C functions using activation records allocated on the stack. This holds even though we can use functions as return values because we do always have access to the variables used by the function which are either local for the function itself or global.

To sum up we may use functions as return values in C but may not nest functions. In Pascal we may nest functions but not use functions as return values. We have shown that both languages can be implemented using traditional compiler techniques with activation records allocated on the stack.

3.1.3 Functions in SML

In Standard ML we have both nesting of functions and functions as first class values. A computation may result in a function which may be applied in a context different from where it was defined. A function may access all variables declared before the function itself. Consider the following program

```
fun gen_f() =
  let
    val c = 41
    fun f(arg) = arg + c
  in
    f
  end

val g = gen_f()
val res = g(1)
```

The function `gen_f` returns a function `f`, that depends on the local value `c`. When `f` is called (through variable `g`) the activation of `gen_f` is removed and if `c` is stored in the activation record for `gen_f` then it is not available when `f` is called. We must keep access to the local value `c` even though the function in which it is created has returned. The combination of nested functions and functions as first class values makes it impossible to store all local values on the stack.

The solution is to use a *closure* [33], which in addition to the code for the function, holds the environment in which the function was declared. The environment maps all free variables (variables defined before the function itself) to their values. Given the closure (which in general is allocated on the heap and not on the stack) it is possible to apply the function everywhere in the program because all information needed to evaluate the function is held in the closure.

In the above example the function `f` has `c` as free variable and the closure for `f` is a *closure record* written: $(code_{\mathtt{f}}, v_{\mathtt{c}})$, where $code_{\mathtt{f}}$ is a pointer to the code for `f` and $v_{\mathtt{c}}$ is the value of `c`. When we call `f`, we fetch the code pointer from the closure (the first element of the closure record) and pass `f` the closure record as an argument. The closure record is allocated on the heap and is accessible for as long time as needed.[1]

3.1.4 Uniform Representation of Functions

With functions as first class values, it is possible to call several functions from the same application point. Therefore, a uniform representation of closures is needed. Consider the program:

```
fun f(x) =
  let
    val a = 41
    val b = 42
  in
    if x then
      (fn y => y+a)
    else
      (fn z => z+2*b)
  end

val g = f(true) 1 (*1*)
```

The result function from `f(true)` called at `(*1*)` depends on the argument to `f`. However, the code performing the call is the same so the two functions (closures) returned from the if expression must have the same representation.

To call a function we must know where the code for the function is in addition to the environment needed to evaluate the function (in this case either `a` or `b`). We use the first address in the closure for the code pointer and the following addresses for free variables. The code calling the function then knows where to jump but not necessarily how the free variables are organized in the closure which is fine because they are first used in the body of the called function. The closure records for the two functions are $(code_{fny}, \mathtt{a})$ and $(code_{fnz}, \mathtt{b})$. A closure can be viewed as a tuple where

[1]In a region based system, region inference decides whether the closure record is allocated on the stack or on the heap. The point is that the closure record should stay allocated for as long time as it may be used.

the first element is the code pointer and the succeeding elements are free variables of the function. The arguments `y` or `z` are also passed when the functions are called.

3.1.5 Closure Representation

There are many ways to represent the closures [5, page 112]. We have two basic methods: *flat* and *linked* representation and they can be combined in several ways. We have used flat representation in the above examples.

With flat representation we copy the free variables into the closure when the closure is created. This works only if all variables are immutable, (i.e., the contents cannot change after declaration). This is indeed the case in SML contrary to C and Pascal. Consider the program

```
let
  val w=1
  fun g(x) =
    let
      fun f(y) = w+x+y
    in
      f(42)
    end
in
  g(3)
end
```

With linked representation the closure for `g` contains all free variables in `g` (i.e., `w`) that are not free in the outer function (i.e., still `w`). The closure for `f` contains `x` (`x` is not free in `g`) but not `w` because `w` is free in `g`. Instead we put a closure pointer to the closure for `g` in the closure for `f` and access `w` by dereferencing the two closure pointers.

Linked representation works for mutable variables because at any time there is only one instance of the variable. Flat representation may be adopted to work with mutuable variables; if there will ever be more than one instance of a variable it may be implemented as a reference to a cell in memory.

The flat representation tends to create larger closures (i.e., time used on creation) but optimizes fetching values to only one fetch operation given the closure pointer. Linked representation minimizes the size of closures and has better creation times but fetching may now be done through several closure pointers which is a serious disadvantage. Many combinations of flat and linked representation can be used. Using either flat or linked representation influence the life range for closure records. With linked representation, a closure record may be live even though the function it represents is never called again. This is not the case with flat representation. Region inference, that infers life ranges for closure records, assumes that flat representation is used, thus, we use flat rerpesentation in the ML Kit.

3.1.6 A Closure is Not Always Needed

Even though Standard ML (contrary to Pascal) offers functions as first class values, functions are not always used as a first class value. Actually, it seems that most functions are used as in Pascal. Consider the program at left.[2]

```
let
  val q = 1
in
  let
    val f = fn x => x + q
  in
    f 4 (*1*)
  end
end
```

```
let
  val q = 1
in
  let
    val f = fn <x,q> => x + q
  in
    f <4,q>
  end
end
```

The function `f` does not escape and the free variable `q` needed to evaluate `f` is available at the application point `(*1*)`. It is therefore not necessary to build a closure for `f`. Instead we can pass the free variable as an extra argument to `f`. We then avoid building a closure and the free variable will probably be in a machine register in `f` which is faster that fetching it from a closure. The above code at right has the free variable `q` as an extra argument to `f`.

The following rules may be used to find the function declarations that do not need to be represented by closures. Let f be the identifier to which the function is bound as in the above program.

1. The identifier f must appear only in applications and only as the operator, (i.e., $f\ e_2$).

2. All applications of f must have the free variables of f accessible (i.e., either as local or free variables in the function containing the application).

The first rule says that f is a known function, that is, *all* application points calling f only call f and the code to jump to is therefore known at compile time. The second rule makes sure that we can actually access all the free variables to f at the call site. A variable is *accessible* in a function g if it is

[2] It is important to note that even though the functionality offered by Standard ML is not always used by the programmer then it is still a very important functionality and it is worth the trouble dealing with.

either a local declared variable or a free variable to g. We have to be a bit careful, though, because we can nest the declaration of variables and reuse names. A free variable to f, say v, may be overridden by a more nested declaration of v which is less nested than the call site. Consider the code at left:

```
let
  val free = 42
in
  let
    fun f(x) = free+x
  in
    let
      val free = 43
    in
      f(2)
    end
  end
end
```

```
let
  val free = 42
in
  let
    fun f<x,free> = free+x
  in
    let
      val free = 43
    in
      f <2,free>
    end
  end
end
```

If we just pass the variable `free` to `f` as an extra argument as in the code at right then we get 45 and not 44 which is the right result. We must rename all variables such that variables on the left side of `=` are different. This has already been done on RegExp.

The function f in the above program satisfies the two rules but the mostly nested function g below does not; which rules are not satisfied?

```
let
  val a = 1
in
  let
    val g =
      let
        val b = 2
        val g = fn x => a+b+x
      in
        g
      end
  in
    g 3
  end
end
```

The first rule is not satisfied because the identifier g is the result in the inner let expression which is not an application. Rule two is satisfied even though it may not seem so. In the application $g\ 3$ the free variable b to g is not in scope. This, however, is not a violation of the rule because g in $g\ 3$ is not the same identifier as g to which the function is bound.

We need a closure for the inner function bound to the inner g because it escapes. The closure only has to contain the free variable b because the application $g\ 3$ is in scope of variable a. The code pointer is not needed because g is the only function which can be applied. There is a variety of ways to specialize functions as described elsewhere [56].

We note that a function f may still be implemented without a closure even though there are lambda abstractions in between the declaration of f and use of f (i.e., f may escape through another function g). Consider the code:

```
let
  val a = 42
  val f = fn x => x - 2 + a
  val g = fn x => f x + a
in
  g
end
```

The application to `f` in the body of `g` escapes because `g` escapes. However, no matter where `g` is applied, the application to `f` inside `g` will always be to `f` and the free variable `a` of `f` will always be accessible inside `g` because `a` is also free in `g`. The function `g` needs to be closure implemented. We note that if `a` is not free in `g` then rule 2 is not satisfied because `a` is not accessible at the call to `f`.

3.1.7 Region polymorphic functions

A region polymorphic function differs from an ordinary function in three ways [38]:

1. in every application $f\ \langle e_1, \ldots, e_n \rangle\ [a_1, \ldots, a_m]\ a$ (see Figure 2.5 on page 22) the function called (e.g., f) will always be known. It is not necessary to store the code pointer in the closure for f. The first rule in Section 3.1.6 is always satisfied.

2. in addition to the ordinary arguments $\langle e_1, \ldots, e_n \rangle$, f also has a region parameter (i.e., *region vector*) $[a_1, \ldots, a_n]$

3. region polymorphic functions may be mutually recursive.

Consider the program with `g` and `f` as mutually recursive functions.

```
let
  fun g [p1] x = (x+a+f[p1]x)
  and f [p2] y = (y+b+f[p2]y + g[p2]y)
in
  g[r1]4 + f[r2]5
end
```

In rule 1 above we noticed that no code pointer is necessary in the closures for the two functions. The following closures may then be used for the two functions:

> The function `g` has `a` and `f` as free variables: $(\mathtt{a}, \mathtt{f})$.
>
> The function `f` has `b`, `f` and `g` as free variables: $(\mathtt{b}, \mathtt{f}, \mathtt{g})$.

where `f` and `g` are pointers to the two closures respectively. The mutually recursiveness of the two functions is obvious in the closures. However, the closures for mutually recursive functions may be merged into one shared closure [5, 9]. This is possible because the closures are built at the same time anyway and they have the same lifetime. Merging the two closures gives the *shared closure record*: $(\mathtt{a}, \mathtt{b}, \mathtt{f}, \mathtt{g})$. We choose to use shared closures rather than separate closures.

The next observation is that the two functions `f` and `g` may be removed from the shared closure. There is no need to have the functions `f` and `g` inside the shared closure to point at the same shared closure (i.e., `f` and `g` are not considered free in the body of the functions). The shared closure record then becomes: $(\mathtt{a}, \mathtt{b})$.

We end the discussion about shared closures by looking at how a shared closure is accessed in different situations [38, page 44]. Let $f_1, \ldots, f_n$ be n mutually recursive functions with bodies $e_1, \ldots, e_n$, respectively, and scope e. Let the shared closure be sc.

- Accessing any $f_i, i \in 1, \ldots, n$ yields the same closure sc in $e_1, \ldots, e_n$ and e. Now assume an application $f_i \ \langle e_{arg1}, \ldots, e_{argn} \rangle \ [a_1, \ldots, a_m] \ a$ directly within e, that is, not within a function in e. This yields the shared closure sc being built for $f_1, \ldots, f_n$ before evaluating e.

- If the application $f_i \ \langle e_{arg1}, \ldots, e_{argn} \rangle \ [a_1, \ldots, a_m] \ a$ is within e but not directly within then the shared closure is accessed through the closure for an in between function (i.e., the shared closure is a free variable of the in between function).

- Assuming the application $f_i \ \langle e_{arg1}, \ldots, e_{argn} \rangle \ [a_1, \ldots, a_m] \ a$ is directly within $e_i, i \in 1, \ldots, n$ then the shared closure bound to f_i is the same as the current one, (i.e., no code is needed to find the closure).

3.1.8 Closure Explication

Several compilers, including the former versions of the ML Kit, decides the representation of closures at the time they compile from lambda calculus (or continuation passing style) into machine code (i.e., during code generation). However, it is possible to separate the representation of closures (closure conversion) from other compilation tasks because the representation of closures is not machine dependent and it is not difficult to express closures explicitly in intermediate languages.

There are several advantages of representing closures explicitly. Code generation becomes simpler because all functions are at top level exactly as they will be after code generation. Moreover, it is advantageous to include machine independent information in the intermediate languages because it gives more information for the optimizations performed in the backend.

We have implemented a closure conversion phase in the ML Kit that converts the RegExp language into a language called ClosExp where function declarations are substituted for closures being the values of the functions. Furthermore, we give each function the closure as an explicit argument and all accesses to values in the closure are changed into selects into the closure. We also make the region vectors explicit in the program. There are no free variables in the functions after closure conversion and they can be lifted to top level.[3] The program has been translated from one expression into a list of function declarations. Having closures made explicit in ClosExp makes the succeeding phases, including code generation, simpler and we have made it possible (though not used yet) to apply more advanced analyses on the representation of closures.

3.2 Calculating the Need For Closures

In this section we present the algorithm that finds the functions (either ordinary or `letrec` bound) that may be implemented without a closure, that is, with all free variables passed as arguments.

We use the two rules laid out in Section 3.1.6 and the additional requirement that the maximum number of free variables may at most be *max_free_args*. The limit *max_free_args* makes sure that we do not pass huge closures as arguments where the total number of arguments exceeds the number of machine registers available. The number *max_free_args* depends on the register allocator used. If the closure arguments have to be passed on the stack anyway then it is likely that it is better to build the closure in order to limit the register pressure.

The algorithm performs a forward scan of the RegExp expression and

[3] Variables imported in the module may still be considered free but they are referenced through a global label.

computes a partially defined function

$$fe \in FE = LamVar \rightarrow \{\texttt{fn}\ free\ list,\ \texttt{fix}\ free\ list\}.$$

Given a variable f to which a function is bound then if f is not in the domain of fe then the function bound to f should be implemented with a closure. If f is in the domain of fe then it may be implemented without a closure and $fe(f)$ gives the free variables of the function. We let `fn` denote ordinary functions and `fix` denote letrec bound functions.

At application points we need to make sure that the free variables of the called function are accessible (rule two). The accessible variables are recorded in a variable environment

$$VE = RegVar \cup LamVar$$

The algorithm uses two return types

$$rtnType ::= \texttt{func}\ free\ list \mid \texttt{other}$$

We let $RtnType$ denote the set of return types. If a subexpression evaluates to a function then the return type is `func` and otherwise the return type is `other`. Consider the expression:

```
let
  val f = fn x => x+2
in
  e
end
```

The return type returned from expression `fn x => x+2` is `func` and we insert the relation $\texttt{f} \mapsto \texttt{func}$ in fe.

The algorithm inserts a function in fe when the function is declared. The function is removed from fe if one of the requirements are not satisfied.

Algorithm $\mathcal{F}$ does one forward traversal of the program.

$$\mathcal{F} : RegExp \rightarrow FE \rightarrow VE \rightarrow FE \times RtnType\ seq$$

3.2.1 Variables and Constants

$$\begin{array}{l}
\mathcal{F}\ [\![x]\!]\ fe\ ve = \\
\quad \texttt{if}\ x \in \mathbf{dom}(fe)\ \texttt{then} \\
\qquad (fe \setminus \{x\}, \langle \texttt{other} \rangle) \\
\quad \texttt{else} \\
\qquad (fe, \langle \texttt{other} \rangle)
\end{array}$$

The algorithm handles applications specifically so the variable x in the above case will always appear as a use, that is, not as an operator in an application. If x represents a function then x has to be implemented with a closure and is therefore removed from fe (i.e., rule one).

$\mathcal{F}$ $[\![i]\!]$ *fe ve* = (*fe*, $\langle$`other`$\rangle$)

$\mathcal{F}$ $[\![$`nil`$]\!]$ *fe ve* = (*fe*, $\langle$`other`$\rangle$)

3.2.2 Boxed Expressions

$\mathcal{F}$ $[\![(e_1, \ldots, e_n)\ a]\!]$ fe_0 *ve* =
  `let`
    `val` $(fe_i,\_) = \mathcal{F}$ $[\![e_i]\!]$ fe_{i-1} *ve* $i = 1, \ldots, n$
  `in`
    $(fe_n$, $\langle$`other`$\rangle$)
  `end`

If e_i for some i $\in 1, \ldots, n$ evaluates to a function then the function is implemented with a closure because it is stored in the record.

$\mathcal{F}$ $[\![\lambda^{free}\ \langle x_1, \ldots, x_n\rangle$ `=>` $e\ a]\!]$ *fe ve* =
  `let`
    `val` $(fe',\_) = \mathcal{F}$ $[\![e]\!]$ *fe* (`setof`(*free*) $\cup \{x_1, \ldots, x_n\}$)
  `in`
    (fe', $\langle$`func` *free*$\rangle$)
  `end`

The function `setof` converts the list of variables into a set of variables. The set of accessible variables in the body of the function is precisely the arguments and the free variables of the function. The return type is a function indicated by `func`.

3.2.3 Expressions

The interesting cases are the `let`, `letrec` and application constructs.

Constructors

$\mathcal{F}$ $[\![::\ e]\!]$ *fe ve* = $\mathcal{F}$ $[\![e]\!]$ *fe ve*

Binary Operations

$\mathcal{F}$ $[\![e_1\ bop\ e_2]\!]$ *fe ve* =
  `val` $(fe_1,\_)$ `=` $\mathcal{F}$ $[\![e_1]\!]$ *fe ve*
  `val` $(fe_2,\_)$ `=` $\mathcal{F}$ $[\![e_2]\!]$ fe_1 *ve*
`in`
  $(fe_2$, $\langle$`other`$\rangle$)
`end`

The result of a binary operation is an integer so the return type is `other`.

Selection

$\mathcal{F}$ $[\![\texttt{\#}n(e)]\!]$ fe $\mathit{ve} = \mathcal{F}$ $[\![e]\!]$ fe ve

If $\texttt{\#}n(e)$ evaluates to a function then the function is closure implemented (i.e., the function is bound to a variable that has been used in an expression creating a record.)

`letrec` bound functions

$\mathcal{F}$ $[\![\texttt{letrec}\ f^{\mathit{free}}\ \langle x_1, \ldots, x_n\rangle\ [\rho_1 : m_1, \ldots, \rho_l : m_l]\ a\ =\ e_1\ \texttt{in}\ e_2\ \texttt{end}]\!]$ fe $\mathit{ve} =$

```
let
  val fe₀ = if |free| < max_free_args then
              fe + {f ↦ (fix free)}
            else
              fe
  val (fe′,_) = F ⟦e₁⟧ fe₀ (setof(free) ∪ {f, x₁, ... , xₙ, ρ₁, ... , ρₗ})
in
  F ⟦e₂⟧ fe′ (ve ∪ {f})
end
```

Initially we assume f not to be closure implemented by inserting f into fe iff the number of free variables is less than $\mathit{max\_free\_args}$. If f should be closure implemented then f is removed from fe when computing e_1 and e_2. The function f is accessible in both e_1 and e_2. The function $|\cdot| : \alpha\ \mathit{list} \to \mathit{Int}$ computes the number of elements in a list.

Applications

Ordinary applications and `letrec` applications are handled similarly. The `letrec` application is:

$\mathcal{F}$ $[\![f\ e\ [a_1, \ldots, a_m]\ a]\!]$ fe $\mathit{ve} =$

```
let
  val fe₀ = if (f ∈ dom(fe) and
                fe(f).free ⊆ ve) then
              fe
            else
              fe \ {f}
  val (fe′,_) = F ⟦e⟧ fe₀ ve
in
  (fe′, ⟨other⟩)
end
```

If $\mathit{fe}(f) = (\_\ \mathit{free}\ \mathit{list})$, then we write $\mathit{fe}(f).\mathit{free}$ for free. The second requirement in the condition ensures that all free variables are accessible at the application point (i.e., rule two). The return type is $\langle$`other`$\rangle$ because a function is not declared.

Letregion

$\mathcal{F}$ ⟦`letregion` $\rho : m$ `in` e `end`⟧ $fe\ ve =$
  $\mathcal{F}$ ⟦e⟧ $fe\ (ve + \{\rho\})$

The region variable ρ is accessible in the body e.

Declaring variables

```
F [[⟨e1, ... , en⟩]] fe0 ve =
  let
    val (fei,rtn_typei) = F [[ei]] fei−1 ve      i = 1, ... , n
  in
    (fen, ⟨rtn_type1, ... , rtn_typen⟩)
  end
```

The set of accessible variables in each sub expression e_i is the same because no variable is declared between the sub expressions. We use the newest environment *fe* in each sub expression because if a function has to be implemented with a closure in one of the sub expressions then it has to be deleted from the *fe* that we return.

```
F [[let val ⟨x1, ... , xn⟩ = e1 in e2 end]] fe ve =
  let
    val (fe0,⟨rtn_type1, ... , rtn_typen⟩) = F [[e1]] fe ve
    val fei = case rtn_typei of      i = 1, ... , n
                func free => fei−1 + {xi ↦ (fn free)}
              | other => fei−1
  in
    F [[e2]] fen (ve + {x1, ... , xn})
  end
```

The functions (if any) bound to a variable x_i are inserted into *fe* as an ordinary function `fn`. The variables $x_1, \ldots, x_n$ are distinct.

Case

```
F [[case e1 of c => e2 | _ => e3]] fe ve =
  let
    val (fe1,_) = F [[e1]] fe ve
    val (fe2,_) = F [[e2]] fe1 ve
    val (fe3,_) = F [[e3]] fe2 ve
```

```
  in
    (fe₃,⟨other⟩)
  end
```

$\mathcal{F}$ [[`case` e_1 `of ::`x `=>` e_2 `|` `_` `=>` e_3]] $fe\ ve$ =

```
  let
    val (fe₁,_) = F [[e₁]] fe ve
    val (fe₂,_) = F [[e₂]] fe₁ (ve+{x})
    val (fe₃,_) = F [[e₃]] fe₂ ve
  in
    (fe₃,⟨other⟩)
  end
```

If e_2 or e_3 return a function then the function has to be implemented with a closure because the application sites do not know which function is applied. The return type is therefore `other`.

3.3 ClosExp

In this section we present ClosExp which is a refinement of RegExp where functions are lifted to top level. ClosExp is the result of closure conversion.

3.3.1 Call convention

A call convention specifies how arguments to a function are passed. We use a call convention with five entries:

$$
\begin{aligned}
cc = \{&\text{clos:} && \mathit{LamVar\ option},\\
&\text{free:} && \mathit{LamVar\ list},\\
&\text{args:} && \mathit{LamVar\ list},\\
&\text{reg\_vec:} && \mathit{LamVar\ option},\\
&\text{reg\_args:} && \mathit{LamVar\ list}\ \}
\end{aligned}
$$

A function f may either have its free variables passed in a closure or as extra arguments. If a closure is used then the closure is passed as an argument to the function and the closure argument is written in the clos entry. If a closure is not used then the free variable arguments are written in the entry free. It is possible, in the same call, to pass some of the free variables in free variable arguments and some of the free variables in a closure argument. However, we do not use that option, that is, the entry clos and the entry free cannot, at the same time, be non empty.

Regular arguments are written in the entry args. As with free variables we can either pass region variables in a region vector or as region variable arguments. We always pass region variables in a region vector argument and the region vector argument is written in the entry reg_vec. The entry reg_args is never used and is included in the call convention for completeness. It is likely that we, in the future, may extend the call conventions to include functions receiving region variables as extra arguments and thereby eliminate the need for a region vector; this is especially useful for functions with a low register pressure.

We note that a *list* may be empty (see Figure 2.4) and *LamVar option* means that either there is a variable or not.

The call convention is designed to maximize flexibility in calling functions. The call convention fully describes how both ordinary and `letrec` functions are called. The closure conversion phase lifts all functions to top level and with the call convention specified above we get the following grammar for top level declarations:

$$\begin{array}{rcl} \mathit{top\_decl} & ::= & \lambda^{\texttt{fun}}_{lab}\ cc\ \texttt{=>}\ e \\ & | & \lambda^{\texttt{fn}}_{lab}\ cc\ \texttt{=>}\ e \end{array}$$

The two constructs are identical except that we syntactically differentiate between `letrec` (abbreviated `fun`) and ordinary functions (abbreviated `fn`). The label *lab* is a unique ident (name) for the function; $lab \in Label$. We let the set *TopDecl* denote the set of top level declarations ranged over by *top_decl*.

The order of arguments passed to the function is uniquely determined by the register allocator that allocates the variables to either machine registers or the machine stack.

Application points must follow the call convention for the called function. We use the following grammar at application points:

$$\begin{array}{rcl} e & ::= & e_{ck}\ \langle e_1, \ldots, e_n\rangle\ \langle e_{clos}\rangle\ \langle e_{f_1}, \ldots, e_{f_m}\rangle \\ & | & lab_{ck}\ \langle e_1, \ldots, e_n\rangle\ \langle e_{reg}\rangle\ \langle e_{\rho_1}, \ldots, e_{\rho_l}\rangle\ \langle e_{clos}\rangle\ \langle e_{f_1}, \ldots, e_{f_m}\rangle \end{array}$$

For ordinary function calls the regular arguments $e_1, \ldots, e_n$ are written in the first bracket and the optional closure and free values in the second and third bracket. We never use the second and third bracket at the same time.

The `letrec` application is similar except that the region vector and region values have been squeezed in as the second and third bracket. We note that ClosExp is in K–normal form and all values are either bound to a variable ranged over by x and f or the value is a constant.

In examples, we only write the non empty entries in a call convention.

We did not write the region vector that we reuse in the recursive call to *gen_list* in the example on page 38. In ClosExp we explicitly write all the arguments.

3.3.2 Functions

Now that all functions are lifted to top level the two expressions

$$\lambda^{free}\ \langle x_1, \ldots, x_n\rangle \texttt{ => } e$$

and

$$\texttt{letrec}\ f^{free}\ \langle x_1, \ldots, x_n\rangle\ [b_1, \ldots, b_m]\ a\ =\ e\ \texttt{in}\ e\ \texttt{end}$$

disappear from the grammar.

Instead of the λ expression we introduce a new boxed expression:

$$be\ ::=\ \lambda^{lab}\ [e_{f_1}, \ldots, e_{f_n}]$$

that evaluates to a closure record: $(lab, v_{f_1}, \ldots, v_{f_n})$ where v_{f_i} is the value of e_{f_i}, $i = 1, \ldots, n$. The label lab refers to the function at top level.

The `letrec` construct is replaced by three new constructs. The boxed expressions

$$\begin{array}{rcl} be & ::= & [a_1, \ldots, a_n]_{\text{regvec}} \\ & | & [e_{f_1}, \ldots, e_{f_n}]_{\text{sclos}} \end{array}$$

are the region vector and shared closure respectively. They evaluate to a region vector record $(v_{\rho_1}, \ldots, v_{\rho_n})$ and a shared closure record $(v_{f_1}, \ldots, v_{f_n})$. We also need a `letrec` construct to make the scope for f explicit.

$$e\ ::=\ \texttt{letrec}\ f_{lab}\ \texttt{=}\ be\ a\ \texttt{in}\ e\ \texttt{end}$$

The label lab connects the top level function $\lambda^{\texttt{fun}}_{lab}$ with f. We often omit the label in examples where f and the label are identical. The boxed expression, be, is always a shared closure which is bound to the variable f.

Examples

Consider the RegExp program:

```
let
  val b = 5
  val f = λ{b} ⟨x⟩ =>
             (let
                val fn_y = (λ{b,x} ⟨y⟩ => x + y + b) attop r1
              in
                fn_y
              end) attop r1
in
  f_fncall ⟨5⟩
end
```

In ClosExp we write the same program as:

$\lambda^{\texttt{fn}}_{main}$ { } `=> let`
  `val` $b = 5$
  `val` $f = \lambda^{fn\_x}$ $[b]$ `attop` *r1*
`in`
  $f_{\texttt{fncall}}$ $\langle 5\rangle$ $\langle f\rangle$ $\langle\rangle$
`end`

$\lambda^{\texttt{fn}}_{fn\_x}$ {clos=c, args=$[x]$} `=>`
`let`
  `val` $fn\_y = \lambda^{fn\_y}$ $[\texttt{\#}1(c),\ x]$ `attop` *r1*
`in`
  fn_y
`end`

$\lambda^{\texttt{fn}}_{fn\_y}$ {clos=c, args=$[y]$} `=>` $\texttt{\#}2(c)$ `+` y `+` $\texttt{\#}1(c)$

Three functions are created at top level where *main* is the function that initiates computation. All functions are closed because access to arguments and free variables are described in the call convention. We insert explicit selections into the closure records. The closure record passed to *fn_y* is $c = (fn\_y, b, x)$ where *fn_y* is the code label and the expression $\texttt{\#}2(c)$ denotes x in the original program and $\texttt{\#}1(c)$ denotes b. The closure record for *fn_x* is $c = (fn\_x, b)$.

We note that ClosExp is not a typed intermediate language. However, it is possible to give the closure record for *fn_y* a simple type, for instance: $c : Code \times Int \times Int$ where *Code* is a type donoting code. It would greatly improve the reliability of the compiler if the intermediate languages could be type checked even though the type system might not be sound, that is, we cannot prove that the source and translated program evaluate to the same result even though the translated program type checks. It is also possible to engineer sound type systems where we can prove that if the closure converted program type checks then it evaluates to the same result as the original program [36]. Unfortunately, this is beyond the scope of this project.

The running example for the next chapters is shown below:

```
fun foldl f b xs =
  case xs of
    [] => b
  | x::xs' => foldl f (f x b) xs'
```

The same program in RegExp is shown in Figure 3.3.

The translation to ClosExp is shown in Figure 3.4 and Figure 3.5. Four top level functions are created.

```
letrec foldl[ ] ⟨f⟩ [r7 : 4, r8 : 4] attop r1 =
  let
    val fn_b = λ[f,r8,foldl] ⟨b⟩ attop r7 =>
    let
      val fn_xs = λ[b,f,foldl] ⟨xs⟩ attop r8 =>
        (case xs of
           nil => b
         | :: (v942) =>
             let
               val x = #0(v942)
               val xs' = #1(v942)
             in
               letregion r22:4 in
               let
                 val k80 =
                 letregion r24:4 in
                 let
                   val k77 =
                     letregion r25:2 in
                       foldl_funcall ⟨f⟩ [atbot r24, atbot r22] atbot r25
                     end (*r25*)
                   val k79 =
                     let
                       val k78 = f_fncall ⟨x⟩
                     in
                       k78_fncall ⟨b⟩
                     end (*let*)
                 in
                   k77_fncall k79
                 end (*let*)
                 end (*r24*)
               in
                 k80_fncall ⟨xs'⟩
               end (*let*)
               end (*r22*)
             end (*let*))
    in
      fn_xs
    end (*let*)
  in
    fn_b
  end (*let*)
in
  foldl_funcall ⟨sum⟩ [atbot r4, atbot r5] atbot r6
end (*letrec*)
```

Figure 3.3: Our running example program written in RegExp. We assume *sum*, $r1$, $r4$, $r5$ and $r6$ to be previously defined. We have $k77 = foldl\ f$, $k78 = f\ x$, $k79 = f\ x\ b$ and $k80 = foldl\ f\ (f\ x\ b)$.

$\lambda^{\texttt{fun}}_{foldl}$ {args=[f], recvec=rv, clos=c} `=>`
  `let`
    `val` fn_b `=` λ^{fn_b} [f, `#1`(rv), c] `attop_ff` `#0`(rv)
  `in`
    fn_b
  `end`

$\lambda^{\texttt{fn}}_{fn\_b}$ {args=[b], clos=c} `=>`
  `let`
    `val` fn_xs `=` λ^{fn_xs} [b, `#1`(c), `#3`(c)] `attop_ff` `#2`(c)
  `in`
    fn_xs
  `end`

$\lambda^{\texttt{fn}}_{main}$ { } `=>`
  `letrec` $foldl_{foldl}$ `=` $[\ ]_{sclos}$ `attop_li` $r1$
  `in`
    `let`
      `val` rv `=` [`attop_lf` $r4$, `atbot_lf` $r5$]$_{regvec}$ `atbot_lf` $r6$
    `in`
      $foldl_{\texttt{funcall}}$ $\langle sum\rangle$ $\langle rv\rangle$ $\langle\rangle$ $\langle foldl\rangle$ $\langle\rangle$
    `end`
  `end`

Figure 3.4: The first three top level functions in our running example program translated into ClosExp. In the `letrec` binding we introduce both the label *foldl* and the variable *foldl* (holding the empty shared closure). In the application the first term *foldl* is the label and the shared closure is accessed in the fourth bracket. Actually the shared closure is empty so it should be omitted and will be in the implementation. The closure record c for fn_b is (fn_b, f, $r5$, $foldl$).

```
λ^fn_fn_xs {args=[xs], clos=c} =>
  (case xs of
     nil => #1(c)
   | :: (v942) =>
       let
         val x = #0(v942)
         val xs' = #1(v942)
       in
         letregion r22:4 in
           let
             val k80 =
               letregion r24:4 in
                 let
                   val k77 =
                     letregion r25:2 in
                       let
                         val rv = [atbot_lf r24, atbot_lf r22] atbot_lf r25
                       in
                         foldl_funcall ⟨#2(c)⟩ ⟨rv⟩ ⟨⟩ ⟨#3(c)⟩ ⟨⟩
                       end
                     end (*r25*)
                   val k79 =
                     let
                       val k78 = #2(c)_fncall ⟨x⟩ ⟨#2(c)⟩ ⟨⟩
                     in
                       k78_fncall ⟨#1(c)⟩ ⟨k78⟩ ⟨⟩
                     end (*let*)
                 in
                   k77_fncall ⟨k79⟩ ⟨k77⟩ ⟨⟩
                 end (*let*)
               end (*r24*)
           in
             k80_fncall ⟨xs'⟩ ⟨k80⟩ ⟨⟩
           end (*let*)
         end (*r22*)
       end (*let*))
```

Figure 3.5: The main function *fn_xs*. The label in the call to *foldl* is known at compile time and is therefore not free in the function, however, the shared closure *foldl* is free in the function. The closure record c is (*fn_xs*, b, f, *foldl*).

3.3.3 Storage Modes

We extend the set of storage modes, see Section 2.5, in order to make code generation easier, see Chapter 9. The storage modes `attop`, `atbot` and `sat` are replaced by eight storage modes:

`attop_ff`	`attop_fi`
`attop_lf`	`attop_li`
`atbot_lf`	`atbot_li`
`sat_ff`	`sat_fi`

We have added the letter combinations:

`ff`: a formal region parameter with finite multiplicity.

`fi`: a formal region parameter with infinite multiplicity.

`lf`: a `letregion` bound region variable with finite multiplicity.

`li`: a `letregion` bound region variable with infinite multiplicity.

Consider the allocation point: e `attop_ff` ρ. The storage mode `attop_ff` says that region variable ρ is bound as a formal region parameter with finite multiplicity. The new storage modes do not add new information to the program but moves the information to places where the information is used by the code generator. Knowing how a region variable is bound and its multiplicity at the application point makes code generation easier. The set *SMA* is the set of storage mode annotations ranged over by *sma*.

The function **resolve_sm** defined on page 35 is redefined to work on the new storage modes:

`resolve_sm`: *StorageMode* $\times$ *SMA* $\rightarrow$ *StorageMode*
`resolve_sm(_, atbot_lf) = atbot`
`resolve_sm(_, atbot_li) = atbot`
`resolve_sm(_, attop_ff) = attop`
`resolve_sm(_, attop_fi) = attop`
`resolve_sm(_, attop_lf) = attop`
`resolve_sm(_, attop_li) = attop`
`resolve_sm(atbot, sat_ff) = atbot`
`resolve_sm(atbot, sat_fi) = atbot`
`resolve_sm(attop, sat_ff) = attop`
`resolve_sm(attop, sat_fi) = attop`

3.3.4 Grammar for ClosExp

The grammar for ClosExp is shown in Figure 3.6 and semantic objects for the grammar in Figure 3.7.

For ordinary applications, e_{ck} is either a closure or a label. For `letrec` applications we always know which function to call. We let the set *CC* denote call conventions. The translation into ClosExp (Section 3.4.3) introduces extra let bindings and it is possible that a region descriptor is bound to a lambda variable *lv* and not a region variable ρ. This happens when region descriptors are stored in region vectors and then selected from the region vectors or simply appear as a free variable to the function. We let the set *ClosExp* be the set of expressions ranged over by e.

3.3.5 Dynamic Semantics for ClosExp

The semantic objects used in the dynamic semantics for ClosExp is shown in Figure 3.8. A region name *rn* is used to identify a region in the heap. A region descriptor contains the region name together with the multiplicity and storage mode. We have turned a region descriptor into a value because they are stored in region records. Because region descriptors are values we have no need for the region environment *RE* as used in the dynamic semantics for RegExp. Note the difference between a storage mode *sm* and a storage mode annotation *sma* defined in the grammar, Figure 3.6. A storage mode is either `atbot` or `attop` and never somewhere–at. A label *lab* is considered a constant used when calling functions.

Top level declarations $\boxed{FE \vdash top\_decl \Rightarrow FE'}$

We create a function environment *FE* containing all the functions declared at top level.

$$\frac{}{FE \vdash \lambda^{\mathtt{fun}}_{lab}\ cc \texttt{ => } e \Rightarrow FE + \{lab \mapsto \langle cc, e\rangle\}} \tag{3.1}$$

$$\frac{}{FE \vdash \lambda^{\mathtt{fn}}_{lab}\ cc \texttt{ => } e \Rightarrow FE + \{lab \mapsto \langle cc, e\rangle\}} \tag{3.2}$$

$$\frac{FE \vdash top\_decl_1 \Rightarrow FE_1 \quad FE_1 \vdash top\_decl_2 \Rightarrow FE_2}{FE \vdash top\_decl_1\ ;\ top\_decl_2 \Rightarrow FE_2} \tag{3.3}$$

Begin evaluation $\boxed{st, h, E \vdash e \Rightarrow v, h'}$

To start evaluation we call the function *main* with an empty call convention. The stack and heap are empty. The function environment *FE* is obtained from the top level declarations.

$$\frac{}{\{\}, \{\}, (\{\}, FE) \vdash main_{fncall}\langle\rangle\ \langle\rangle\ \langle\rangle \Rightarrow v, h'} \tag{3.4}$$

$$
\begin{array}{rcl}
\mathit{top\_decl} & ::= & \lambda^{\texttt{fun}}_{lab}\ cc\ \texttt{=>}\ e \\
& \mid & \lambda^{\texttt{fn}}_{lab}\ cc\ \texttt{=>}\ e \\
& \mid & \mathit{top\_decl}_1\ \texttt{;}\ \mathit{top\_decl}_2 \\
\\
ck & ::= & \texttt{funjmp} \mid \texttt{funcall} \mid \texttt{fnjmp} \mid \texttt{fncall} \\
sma & ::= & \texttt{attop\_li} \mid \texttt{attop\_lf} \mid \texttt{attop\_fi} \mid \texttt{attop\_ff} \\
& \mid & \texttt{atbot\_li} \mid \texttt{atbot\_lf} \mid \texttt{sat\_fi} \mid \texttt{sat\_ff} \\
a & ::= & sma\ xv \\
m & ::= & n \mid \infty \\
b & ::= & \rho : m \\
\\
c & ::= & i \mid \texttt{nil} \mid lab \\
pat & ::= & c \mid \texttt{::}\ x \\
bop & ::= & \texttt{+},\ \texttt{-},\ \texttt{<},\ \ldots \\
\\
be & ::= & (e_1, \ldots, e_n) \\
& \mid & \lambda^{lab}\ [e_{f_1}, \ldots, e_{f_n}] \\
& \mid & [a_1, \ldots, a_n]_{\mathrm{regvec}} \\
& \mid & [e_{f_1}, \ldots, e_{f_n}]_{\mathrm{sclos}} \\
\\
e & ::= & x \\
& \mid & be\ a \\
& \mid & c \\
& \mid & \texttt{::}\ e \\
& \mid & e_1\ bop\ e_2 \\
& \mid & \texttt{\#}n(e) \\
& \mid & \texttt{letrec}\ f_{lab}\ \texttt{=}\ be\ a\ \texttt{in}\ e\ \texttt{end} \\
& \mid & e_{ck}\ \langle e_1, \ldots, e_n \rangle\ \langle e_{clos} \rangle\ \langle e_{f_1}, \ldots, e_{f_m} \rangle \\
& \mid & lab_{ck}\ \langle e_1, \ldots, e_n \rangle\ \langle e_{reg} \rangle\ \langle a_{\rho_1}, \ldots, a_{\rho_l} \rangle\ \langle e_{clos} \rangle\ \langle e_{f_1}, \ldots, e_{f_m} \rangle \\
& \mid & \texttt{letregion}\ b\ \texttt{in}\ e\ \texttt{end} \\
& \mid & \texttt{let val}\ \langle x_1, \ldots, x_n \rangle\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\
& \mid & \texttt{case}\ e_1\ \texttt{of}\ pat\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ e_3 \\
& \mid & \langle e_1, \ldots, e_n \rangle
\end{array}
$$

Figure 3.6: The grammar for ClosExp. The call convention *cc* is defined in Section 3.3.1. Note that an allocation point a may be of the form *sma* ρ and *sma lv*.

$$
\begin{array}{rcl}
top\_decl & \in & TopDecl \\
e & \in & ClosExp \\
x, f, lv & \in & LamVar \\
lab & \in & Label \\
i & \in & Int \\
\rho & \in & RegVar \\
cc & \in & CC \\
xv & \in & Var = LamVar \ \cup \ RegVar
\end{array}
$$

Figure 3.7: The semantic objects used in ClosExp. We let CC denote the set of call conventions.

$$
\begin{array}{rcl}
st & \in & Stack \ = RegDesc\ stack \\
h & \in & Heap \ = RegName \ \to Reg \\
s & \in & Store \ = Stack \ \times Heap \\
rd & \in & RegDesc \ = RegName \ \times Mult \ \times StorageMode \\
rn & \in & RegName \\
r & \in & Reg \ = Offset \ \to BoxedVal \\
o & \in & Offset \\
bv & \in & BoxedVal \ = Record \\
ubv & \in & UnBoxedVal \ = Int \ \cup \{\texttt{nil}\} \cup \texttt{::}(Val) \cup Label \\
v & \in & Val \ = Addr \ \cup UnBoxedVal \ \cup RegDesc \\
rec & \in & Record \ = Val \times \ldots \times Val \\
(rd, o) & \in & Addr \ = RegName \ \times Offset \\
m & \in & Mult \ = Int \ \cup \ \{\infty\} \\
sm & \in & StorageMode \ = \{\texttt{atbot},\ \texttt{attop}\} \\
\langle cc, e\rangle & \in & FEelem \ = (CC \times ClosExp) \\
 & & VE = Var \ \to Val \\
 & & FE = Label \ \to FEelem \\
 & & E = SClos = VE \ \times FE
\end{array}
$$

Figure 3.8: The semantic objects used in the dynamic semantics of ClosExp. The function environment FE maps labels into a call convention $cc \in CC$ and the code for the function. The record rec is used for ordinary records, closure records and region records.

Constants $\boxed{\vdash ubv \Rightarrow ubv}$

$$\frac{}{\vdash i \Rightarrow i} \tag{3.5}$$

$$\frac{}{\vdash \texttt{nil} \Rightarrow \texttt{nil}} \tag{3.6}$$

$$\frac{}{\vdash lab \Rightarrow lab} \tag{3.7}$$

We have the constants: integers (i), `nil` and labels lab. They all create an unboxed value.

Allocation Points $\boxed{st, h, E \ \vdash a \Rightarrow (rn, o), h'}$

$$\frac{\begin{array}{c} VE(xv) = (rn, m, sm) \\ \texttt{resolve\_sm}(sm, sma) = \texttt{attop} \\ o \notin \texttt{dom}(h(rn)) \end{array}}{st, h, (VE, FE) \vdash sma\ xv\ \Rightarrow (rn, o), h} \tag{3.8}$$

$$\frac{\begin{array}{c} VE(xv) = (rn, m, sm) \\ \texttt{resolve\_sm}(sm, sma) = \texttt{atbot} \\ h' = h + \{rn \mapsto \{\}\} \\ o \notin \texttt{dom}(h(rn)) \end{array}}{st, h, (VE, RE) \vdash sma\ xv\ \Rightarrow (rn, o), h'} \tag{3.9}$$

If we allocate `atbot` then the heap is updated with the region xv being reset. The function `resolve_sm` is defined on page 64.

Call Conventions $\boxed{st, h, E \ \vdash_{ap} e \Rightarrow \langle cc, e' \rangle, h'}$

$$\frac{\begin{array}{c} st, h, E \ \vdash e \Rightarrow (rn, o), h' \\ h'(rn, o) = (lab, v_1, \ldots, v_n) \\ FE(lab) = \langle cc, e' \rangle \end{array}}{st, h, E \ \vdash_{ap} e \Rightarrow \langle cc, e' \rangle, h'} \tag{3.10}$$

$$\frac{FE(lab) = \langle cc, e \rangle}{st, h, E \ \vdash_{ap} lab\ \Rightarrow \langle cc, e \rangle, h} \tag{3.11}$$

The first rule covers closure implemented functions and the second rule known functions. The call convention and body of function is looked up in *FE*.

We have a rule that matches arguments with a call convention and returns a variable environment where arguments in the call convention are mapped into the argument values.

$$\frac{\begin{array}{l} st, h_{i-1}, E \vdash e_i \Rightarrow v_i, h_i \quad i = 1, \ldots, l \\ \quad cc = \{\text{args} = [x_1, \ldots, x_n], \\ \qquad\quad \text{reg\_vec} = x_{n+1}, \\ \qquad\quad \text{reg\_args} = [x_{n+2}, \ldots, x_m], \\ \qquad\quad \text{clos} = x_{m+1}, \\ \qquad\quad \text{free} = [x_{m+2}, \ldots, x_l]\} \\ \qquad\quad VE' = \{x_i \mapsto v_i\}_{i=1,\ldots,l} \end{array}}{\begin{array}{c} st, h_0, E \vdash_{cc} (cc, \langle e_1, \ldots, e_n\rangle \; \langle e_{n+1}\rangle \; \langle e_{n+2}, \ldots, e_m\rangle \\ \langle e_{m+1}\rangle \; \langle e_{m+2}, \ldots, e_l\rangle) \Rightarrow VE', h_l \end{array}} \tag{3.12}$$

It is not mandatory that all brackets are non empty as long as the call convention matches the supplied arguments. For instance, if a region record is passed as argument then there must be a corresponding variable in the call convention.

Boxed Expressions $\boxed{st, h, E \vdash be \Rightarrow bv, h'}$

$$\frac{st, h_{i-1}, E \vdash e_i \Rightarrow v_i, h_i \quad i = 1, \ldots, n}{st, h_0, E \vdash (e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n), h_n} \tag{3.13}$$

$$\frac{st, h_{i-1}, E \vdash e_{f_i} \Rightarrow v_{f_i}, h_i \quad i = 1, \ldots, n}{st, h_0, E \vdash \lambda^{lab} \; [e_{f_1}, \ldots, e_{f_n}] \Rightarrow (lab, v_{f_1}, \ldots, v_{f_n}), h_n} \tag{3.14}$$

$$\frac{\begin{array}{c} VE(xv_i) = (rn_i, m_i, sm_i) \\ sm'_i = \texttt{resolve\_sm}(sm_i, sma_i) \quad i = 1, \ldots, n \end{array}}{\begin{array}{c} st, h, (VE, FE) \vdash [sma_1 \; xv_1, \ldots, sma_n \; xv_n]_{regvec} \Rightarrow \\ ((rn_1, m_1, sm'_1), \ldots, (rn_n, m_n, sm'_n)), h \end{array}} \tag{3.15}$$

$$\frac{st, h_{i-1}, E \vdash e_{f_i} \Rightarrow v_{f_i}, h_i \quad i = 1, \ldots, n}{st, h_0, E \vdash [e_{f_1}, \ldots, e_{f_n}]_{sclos} \Rightarrow (v_{f_1}, \ldots, v_{f_n}), h_n} \tag{3.16}$$

An ordinary function closure (rule 3.14) has the function label as first component in the closure. A shared closure (rule 3.16) does not have labels in the record fields. A region vector (rule 3.15) is a record of region descriptors containing both the region name, multiplicity and storage mode. The storage mode is used by the allocation rules (rule 3.8 and 3.9). We do not use the multiplicity in the rules, that is, the semantics does not differentiate between finite and infinite regions.

Expressions $\boxed{st, h, E \vdash e \Rightarrow v, h'}$

$$\frac{VE(x) = v}{st, h, (VE, FE) \vdash x \Rightarrow v, h} \tag{3.17}$$

$$\frac{\begin{array}{c} st, h, E \vdash be \Rightarrow bv, h_1 \\ st, h_1, E \vdash a \Rightarrow (rn, o), h_2 \\ h' = h_2 + \{(rn, o) \mapsto bv\} \end{array}}{st, h, E \vdash be\ a \Rightarrow (rn, o), h'} \tag{3.18}$$

$$\frac{\vdash c \Rightarrow ubv}{st, h, E \vdash c \Rightarrow ubv, h} \tag{3.19}$$

$$\frac{\begin{array}{c} st, h, E \vdash e_1 \Rightarrow i_1, h_1 \\ st, h_1, E \vdash e_2 \Rightarrow i_2, h' \\ eval_{bop}(i_1, i_2) = i \end{array}}{st, h, E \vdash e_1\ bop\ e_2 \Rightarrow i, h'} \tag{3.20}$$

$$\frac{st, h, E \vdash e \Rightarrow (v_1, v_2), h'}{st, h, E \vdash \, ::e \Rightarrow \, ::(v_1, v_2), h'} \tag{3.21}$$

$$\frac{\begin{array}{c} st, h, E \vdash e \Rightarrow (rn, o), h' \\ h'(rn, o) = (v_0, \ldots, v_m) \quad 0 \leq n \leq m \end{array}}{st, h, E \vdash \texttt{\#}n(e) \Rightarrow v_n, h'} \tag{3.22}$$

$$\frac{\begin{array}{c} s, h, (VE, FE) \vdash be \Rightarrow bv, h_1 \\ s, h_1, (VE, FE) \vdash a \Rightarrow (rn, o), h_2 \\ h_3 = h_2 + \{(rn, o) \mapsto bv\} \\ VE' = VE + \{f \mapsto (rn, o)\} \\ st, h_3, (VE', FE) \vdash e \Rightarrow v', h' \end{array}}{st, h, (VE, FE) \vdash \texttt{letrec}\ f_{lab}\ \texttt{=}\ be\ a\ \texttt{in}\ e\ \texttt{end} \Rightarrow v', h'} \tag{3.23}$$

$$\frac{\begin{array}{c} st, h, E \vdash_{ap} e_{ck} \Rightarrow \langle cc, e' \rangle \\ st, h, (VE, FE) \vdash_{cc} (cc, \langle \vec{e} \rangle \ \langle\rangle \ \langle\rangle \ \langle e_{clos} \rangle \ \langle \vec{e_f} \rangle) \Rightarrow VE', h_1 \\ st, h_1, (VE', FE) \vdash e' \Rightarrow v', h' \end{array}}{st, h, (VE, FE) \vdash e_{ck} \ \langle \vec{e} \rangle \ \langle e_{clos} \rangle \ \langle \vec{e_f} \rangle \Rightarrow v', h'} \tag{3.24}$$

$$\frac{\begin{array}{c} st, h, E \vdash_{ap} lab_{ck} \Rightarrow \langle cc, e' \rangle \\ st, h, (VE, FE) \vdash_{cc} (cc, \langle \vec{e} \rangle \ \langle e_{reg} \rangle \ \langle \vec{e_\rho} \rangle \ \langle e_{clos} \rangle \ \langle \vec{e_f} \rangle) \Rightarrow VE', h_1 \\ st, h_1, (VE', FE) \vdash e' \Rightarrow v', h' \end{array}}{\begin{array}{c} st, h, (VE, FE) \vdash \\ lab_{ck} \ \langle \vec{e} \rangle \ \langle e_{reg} \rangle \ \langle \vec{e_\rho} \rangle \ \langle e_{clos} \rangle \ \langle \vec{e_f} \rangle \Rightarrow v', h' \end{array}} \tag{3.25}$$

$$\frac{\begin{array}{c} rn \notin \mathtt{dom}(h) \\ h_1 = h + \{rn \mapsto \{\}\} \\ st_1 = \mathtt{push}(st, rn) \\ st_1, h_1, (VE + \{\rho \mapsto (rn, m, \mathtt{attop})\}, FE) \vdash e \Rightarrow v', h_2 \\ (\_, st) = \mathtt{pop}(st_1) \\ h' = h_2 \setminus \{rn\} \end{array}}{st, h, (VE, FE) \vdash \mathtt{letregion}\ \rho : m \ \mathtt{in}\ e\ \mathtt{end} \Rightarrow v', h'} \tag{3.26}$$

$$\frac{\begin{array}{c} st, h, (VE, FE) \vdash e_1 \Rightarrow \langle v_1, \ldots, v_n \rangle, h_1 \\ st, h_1, (VE + \{x_i \mapsto v_i\}_{i=1,\ldots,n}, FE) \vdash e_2 \Rightarrow v', h' \end{array}}{st, h, (VE, FE) \vdash \mathtt{let\ val}\ \langle x_1, \ldots, x_n \rangle \ \mathtt{=}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \Rightarrow v', h'} \tag{3.27}$$

$$\frac{st, h_{i-1}, E \vdash e_i \Rightarrow v_i, h_i \quad i = 1, \ldots, n}{st, h_0, E \vdash \langle e_1, \ldots, e_n \rangle \Rightarrow \langle v_1, \ldots, v_n \rangle, h_n} \tag{3.28}$$

$$\frac{\begin{array}{c} st, h, (VE, FE) \vdash e_1 \Rightarrow \mathtt{::}(v), h_1 \\ st, h_1, (VE + \{x \mapsto v\}, FE) \vdash e_2 \Rightarrow v', h' \end{array}}{st, h, (VE, FE) \vdash \mathtt{case}\ e_1\ \mathtt{of}\ \mathtt{::}x\ \mathtt{=>}\ e_2\ \mathtt{|}\ \_\ \mathtt{=>}\ e_3 \Rightarrow v', h'} \tag{3.29}$$

$$\frac{\begin{array}{c} st, h, E \vdash e_1 \Rightarrow c, h_1 \\ st, h_1, E \vdash e_2 \Rightarrow v', h' \end{array}}{st, h, E \vdash \mathtt{case}\ e_1\ \mathtt{of}\ c\ \mathtt{=>}\ e_2\ \mathtt{|}\ \_\ \mathtt{=>}\ e_3 \Rightarrow v', h'} \tag{3.30}$$

$$\frac{\begin{array}{c} st, h, E \vdash e_1 \Rightarrow c', h_1 \quad c' \neq c \\ st, h_1, E \vdash e_3 \Rightarrow v', h' \end{array}}{st, h, E \vdash \mathtt{case}\ e_1\ \mathtt{of}\ c\ \mathtt{=>}\ e_2\ \mathtt{|}\ \_\ \mathtt{=>}\ e_3 \Rightarrow v', h'} \tag{3.31}$$

$$\frac{\begin{array}{c} st, h, E \vdash e_1 \Rightarrow \texttt{nil}, h_1 \\ st, h_1, E \vdash e_3 \Rightarrow v', h' \end{array}}{st, h, E \vdash \texttt{case}\ e_1\ \texttt{of}\ \texttt{::}x\ \texttt{=>}\ e_2\ \texttt{|}\ \_\ \texttt{=>}\ e_3 \Rightarrow v', h'} \tag{3.32}$$

A boxed expression (rule 3.18) is allocated in a region.

Binary operations (rule 3.20) work on integers only. As for RegExp, we use the function $eval_{bop} : Int \times Int \rightarrow Int$ to evaluate the function *bop* given the two integer arguments.

In the `letrec` rule (rule 3.23) the boxed expression (*be*) is always a shared closure.

Rule 3.24 covers the unknown function calls. The three sets of arguments are evaluated and bound in the environment VE' (rule 3.12) before the body of the called function is evaluated. The arguments and call convention must match.

The rule for known applications (rule 3.25) is similar to the rule for unknown applications except that we have five sets of arguments and not three. Rule 3.12 is used to match the call convention and arguments.

Rule 3.26 allocates a fresh region, sets the multiplicity to m and storage mode to `attop`. The storage mode is chosen arbitrarily because it is explicitly set at the allocation points and when building a region vector (rule 3.15).

3.4 Closure conversion

The algorithm that translates RegExp into ClosExp ($\mathcal{C}$) is fairly straightforward. However, there are a few minor points that we address before we present $\mathcal{C}$.

3.4.1 Preserve K-normal form

In order to keep the code generator simple we want all value creating expressions to be assigned a variable (i.e., preserve the K–normal form). If we keep ClosExp and LineExp (see Chapter 4) in K–normal form and all value creating expressions are atomic then we do not have to introduce more variables during code generation. We can generate code without introducing temporary variables as long as all value creating expressions are atomic. An *atomic* expression is an expression that may easily be implemented in machine code (maybe with the use of a few machine registers). All value creating expressions in ClosExp are atomic expressions. The simplification of function declaration and application in ClosExp is essential for this property to hold.

The ClosExp program in Figure 3.4 and 3.5 is not in K–normal form. For instance, in Figure 3.5 line 19, we have a call to *foldl*. The closure for *foldl* is stored as component number 2 in the closure variable c (i.e., #2(c)). To be in K–normal form we need the invariant that all uses in an expression are either a variable or a constant. If this is not true then we cannot in general generate code for the expression with no additional variables to hold the result of sub expressions.

The algorithm $\mathcal{C}$ must translate the application into

```
let
  val rv = [atbot r24, atbot r22] atbot r25
  val lv = #2(c)
in
  foldl_funcall ⟨lv⟩ ⟨rv⟩ ⟨⟩ ⟨foldl⟩ ⟨⟩
end
```

where lv is a fresh lambda variable.

3.4.2 Call Conventions

The algorithm $\mathcal{C}$ assigns call conventions to functions using the function *FE* computed in Section 3.2.3.

Consider a RegExp function

$$\lambda^{[f_1,\ldots,f_m]}\ \langle x_1,\ldots,x_n\rangle \texttt{ => } e$$

with free variables $f_1,\ldots,f_m$. If the function is closure implemented then we create a ClosExp closure:

$$\lambda^{lab}\ [f_1,\ldots,f_m]$$

and the following function at top level:

$$\lambda^{\texttt{fn}}_{lab}\ \{\text{clos} = \mathit{clos}, \text{args} = [x_1,\ldots,x_n]\} \texttt{ => } e'$$

where e' is e translated into ClosExp and *clos* is a fresh variable. The arguments $x_1,\ldots,x_n$ are inserted into the entry args, that is, we do not use fresh variables for the arguments. At an application to the function we insert

$$x_{ck}\ \langle e_1,\ldots,e_n\rangle\ \langle x\rangle\ \langle\rangle$$

where $e_1,\ldots,e_n$ are the arguments and x is bound to the closure.

In the case that a function is implemented without a closure we get the following ClosExp code:

$$lab$$

where the closure should have been, (i.e., we do not create a closure). At top level we get:

$$\lambda^{\texttt{fn}}_{lab}\ \{\text{free} = [fr_1,\ldots,fr_m], \text{args} = [x_1,\ldots,x_n]\} \texttt{ => } e'$$

where e' is e translated into ClosExp. At an application to the function we insert

$$x_{ck}\ \langle e_1, \ldots, e_n\rangle\ \langle\rangle\ \langle e_{f_1}, \ldots, e_{f_m}\rangle$$

where $e_1, \ldots, e_n$ are the arguments, $e_{f_1}, \ldots, e_{f_m}$ are the free variables and x is bound to the label *lab*. Note, that a label is a constant and we therefore do not store in the region where the closure should have been stored. It is possible to apply a drop regions phase after closure conversion that removes regions that are not used.

The `letrec` bound functions are handled similarly with an additional region vector. Regions holding shared closures of size 0 and region vectors of size 0 may also be removed iff they are write–once regions. A *write–once* region is a region that allows one value to be stored only.

3.4.3 Algorithm $\mathcal{C}$

In order to simplify the presentation, we consider all functions closure implemented. The extension to include non closure implemented functions is straightforward and concern only the translation rules for function declaration and application.

In the translation rules we write ClosExp keywords in `texttype` and algorithmic code in **boldface**.

Variable and Function Environments

The algorithm uses a variable environment *ve* and a function environment for `letrec` bound functions *funE*.

$$ve \in VE = Var \rightarrow Var \cup \{\texttt{\#}n(x) | n \in \mathbb{N} \wedge x \in Var\}$$

and

$$funE \in FunE = VE \rightarrow Label$$

At entry to a function a free variable *fv* may be bound to an element in a closure, say $\texttt{\#}n(clos)$. When translating the body of a function, *ve* maps *fv* into $\texttt{\#}n(clos)$. At every use of *fv* a function **lookup_ve** is used to figure out if a sub expression should be inserted, that is, if *fv* is not mapped into a variable. The lookup function is defined as follows:

lookup_ve : $Var \times VE \rightarrow Var \times SelectExp$
lookup_ve(v, ve) =
  if $v \notin$ **dom**(ve) **then**
    $(v$, `none`$)$
  else
    case $ve(v)$ **of**
      `#`$n(v')$ `=>` **let**

val lv = **fresh_lvar**("lv")
in
$(lv,(lv,\#n(v')))$
end
| v' => $(v',\texttt{none})$

where $se \in \mathit{SelectExp} = \{\texttt{none}\} \cup (\mathit{Var} \times \{\#n(x)|n \in \mathbb{N} \wedge x \in \mathit{Var}\})$. A select expression se is used by the function **insert_ses** to insert extra `let` expressions such that the result ClosExp program is in K–normal form. A fresh lambda variable is returned by the function **fresh_lvar**. The variables are always unique (numbered) even though we reuse the same textual names (e.g., "*lv*").

insert_ses : $\mathit{ClosExp} \times (\mathit{SelectExp\ list}) \rightarrow \mathit{ClosExp}$
insert_ses $(e, [\,]) = e$
insert_ses $(e, \texttt{none}::\mathit{rest})$ = **insert_ses**(e, rest)
insert_ses $(e, (v_1,\#n(v_2))::\mathit{rest})$ =
  let
    val v_1 = $\#n(v_2)$
  in
    insert_ses(e, rest)
  end

The function **insert_se** is similar to **insert_ses** except that it works on one select expression only. Consider the SML code

```
let
  val x = (e1,e2)
in
  e
end
```

where `e1` and `e2` are non atomic expressions. This is by algorithm $\mathcal{C}$ translated into

```
let
  val x =
    let
      val v1 = e1
      val v2 = e2
    in
      (v1,v2)
    end
in
  e
end
```

and not

```
let
  val x =
    (let val v1 = e1 in v1 end,
     let val v2 = e2 in v2 end)
in
  e
end
```

The record in the last translation is not an atomic expression.

Placement of Let Expressions

The placement of **insert_se** calls in the translation rules decides where the extra let bindings are inserted. Consider the function:

$\lambda^{\texttt{fn}}_{lab}$ {args=[x], clos=c} =>

```
  let
    val v1 = #1(c)+x
    val v2 = #2(c)+v1
        ⋮
    val v3 = #1(c)+v1
  in
    (v1,v2,v3)
  end
```

It is simple to just insert let expressions at the outer level binding the selections to new variables:

$\lambda^{\texttt{fn}}_{lab}$ {args=[x], clos=c} =>

```
  let
    val t1 = #1(c)
    val t2 = #2(c)
    val v1 = t1+x
    val v2 = t2+v1
        ⋮
    val v3 = t1+v1
  in
    (v1,v2,v3)
  end
```

but then the life range for t_1 is the entire function. It is better to shorten life ranges for variables such that the register allocator can reuse machine registers efficiently. Inserting local let bindings we get:

$\lambda^{\texttt{fn}}_{lab}$ {args=[x], clos=c} =>
```
let
  val v1 =
    let
      val t1 = #1(c)
    in
      t1+x
    end
  val v2 = ...
    ⋮
  val v3 =
    let
      val t1' = #1(c)
    in
      t1'+v1
    end
in
  (v1,v2,v3)
end
```

The drawback is that we evaluate `#1`(c) twice. We believe this is better than increasing life ranges and we never increase the number of `#`$n(c)$ expressions evaluated compared to the original program.

Region Environment

We use a region environment RE mapping a region variable into the set {`ff`,`fi`,`lf`,`li`} depending on how the region variable is bound and its multiplicity, see Section 3.3.3. The function **convert_sma** converts a storage mode `attop`, `atbot` and `sat` into a new storage mode using the region environment:

convert_sma: $RE \times$ {`atbot`,`attop`,`sat`} $\times$ $RegVar \rightarrow SMA$
convert_sma(re,`sat`,ρ) = `sat`_$re(\rho)$
convert_sma(re,`atbot`,ρ) = `atbot`_$re(\rho)$
convert_sma(re,`attop`,ρ) = `attop`_$re(\rho)$

We use the function **mult** to generate the extensions:

mult: {`f`,`l`} $\times$ $Mult \rightarrow$ {`ff`,`fi`,`lf`,`li`}
mult(`f`,n) = `ff`
mult(`f`,∞) = `fi`
mult(`l`,n) = `lf`
mult(`l`,∞) = `li`

where the second argument is the multiplicity as defined in Section 2.6.

Note, that a free region variable inherit the storage mode from the region environment in which it is defined at the function declaration.

Top Level Declarations

The algorithm $\mathcal{C}^{\mathcal{T}}$ translates the RegExp language into the ClosExp language using $\mathcal{C}$:

$$\mathcal{C} : \mathit{RegExp} \rightarrow \mathit{VE} \rightarrow \mathit{FunE} \rightarrow \mathit{RE} \rightarrow \mathit{Label} \rightarrow \mathit{ClosExp} \times \mathit{SelectExp}$$

We have

$$\mathcal{C}^{\mathcal{T}}(\mathit{re}) = \mathbf{add\_new\_fn}(\text{"}\mathbf{main}\text{"}, \{\}, \mathbf{insert\_se}(\mathcal{C}\ [\![\mathit{re}]\!]\ \{\}\ \{\}\ \{\}\ \text{"}\mathbf{main}\text{"}))$$

where **"main"** is the label used for the main function and *re* is the *RegExp* expression. As a side condition, all top level functions are introduced by the two functions **add_new_fn** and **add_new_fun** which collects them in a sequence as described by the grammar for top level declarations (see Figure 3.6):

$\mathbf{add\_new\_fn} : \mathit{Label} \times \mathit{CC} \times \mathit{ClosExp} \rightarrow \{\texttt{()}\}$
$\mathbf{add\_new\_fun} : \mathit{Label} \times \mathit{CC} \times \mathit{ClosExp} \rightarrow \{\texttt{()}\}$

We write `()` for unit and $\{\}$ for an empty call convention.

Variables

$\mathcal{C}\ [\![x]\!]\ \mathit{ve}\ \mathit{funE}\ \mathit{re}\ \mathit{lab} = \mathbf{lookup\_ve}(x, \mathit{ve})$

Allocation Points

$\mathcal{C}\ [\![\rho]\!]\ \mathit{ve}\ \mathit{funE}\ \mathit{re}\ \mathit{lab} = \mathbf{lookup\_ve}(\rho, \mathit{ve})$

Constants

$\mathcal{C}\ [\![c]\!]\ \mathit{ve}\ \mathit{funE}\ \mathit{re}\ \mathit{lab} = (c, \texttt{none})$

Boxed Expressions

$\mathcal{C}\ [\![(e_1, \ldots, e_n)\ \mathit{sma}\ \rho]\!]\ \ \mathit{ve}\ \mathit{funE}\ \mathit{re}\ \mathit{lab} =$
  let
    val $(e_i', \mathit{se}_i') = \mathcal{C}\ [\![e_i]\!]\ \ \mathit{ve}\ \mathit{funE}\ \mathit{re}\ \mathit{lab} \qquad i = 1, \ldots, n$
    val $(v, \mathit{se}_v) = \mathcal{C}\ [\![\rho]\!]\ \ \mathit{ve}\ \mathit{funE}\ \mathit{lab}$
    val $([e_1'', \ldots, e_n'', v'], \mathit{ses}) = \mathbf{unify\_ses}([(e_1', \mathit{se}_1'), \ldots, (e_n', \mathit{se}_n'), (v, \mathit{se}_v)])$
    val $\mathit{sma}' = \mathbf{convert\_sma}(\mathit{sma}, \mathit{re}, \rho)$
  in
    $(\mathbf{insert\_ses}((e_1'', \ldots, e_n'')\ \mathit{sma}'\ v', \mathit{ses}), \texttt{none})$
  end

We insert the select expressions around the record expression. The function **unify_ses** unifies select expressions such that all selections are different, that is, there is no need to select the same field from a record more than once. Consider the list of variables and select expressions:

$$([(v_1, (v_1, \#n(v))), (v_2, (v_2, \#n(v)))])$$

Both v_1 and v_2 are bound to the same selection $\#n(v)$. We only need one variable bound to $\#n(v)$ and the function **unify_ses** returns the following two lists:

$$([v_1, v_1], [(v_1, \#n(v))]),$$

where v_1 is used instead of v_2 and only one select expression is inserted around the record expression.

$\mathcal{C}\ [\![\lambda^{[f_1,\dots,f_m]}\ \langle x_1,\dots,x_n\rangle$ `=>` $e\ sma\ \rho]\!]\ \ ve\ funE\ re\ lab =$
let
  val $new\_lab = lab$ ^ ".anon"
  val $c =$ **fresh_lvar**("clos")
  val $cc = \{\text{clos} = c, \text{args} = [x_1,\dots,x_n]\}$
  val $ve' = \{f_i \mapsto \#i(c)\}_{i=1,\dots,m}$
  val $(lv_{f_i}, se_{f_i}) =$ **lookup_ve**$(f_i,\ ve) \qquad i = 1,\dots,m$
  val $(v, se_v) = \mathcal{C}\ [\![\rho]\!]\ \ ve\ funE\ lab$
  val $([lv'_{f_1},\dots,lv'_{f_m},v'], ses) =$ **unify_ses**$([(lv_{f_1}, se_{f_1}),\dots,(lv_{f_m}, se_{f_m}),(v, se_v)])$
  val $e' =$ **insert_se**$(\mathcal{C}\ [\![e]\!]\ \ ve'\ funE\ re\ new\_lab)$
  val _ = **add_new_fn**$(new\_lab,\ cc,\ e')$
  val $sma' =$ **convert_sma**(sma, re, ρ)
in
  (**insert_ses**$(\lambda^{new\_lab}\ [lv'_{f_1},\dots,lv'_{f_m}]\ sma'\ v', ses)$,`none`)
end

We assume the function is closure implemented. Note, that the argument variables $x_1,\dots,x_n$ are not inserted into ve' because we do not use fresh variables for arguments.

The above translation rules insert let expressions around a boxed expression, as shown below:

```
                                         let
                                           val x =
                                             let
  let                                          val lv1 = e1
    val x = (e1,e2) at r    ====>              val lv2 = e2
  in                                         in
    e                                          (lv1,lv2) at r
  end                                        end
```

```
in
  e'
end
```

The expression `e'` is `e` translated into ClosExp.

Binary Operators

Let expressions are inserted around a binary operation.

$\mathcal{C}$ $[\![e_1\ bop\ e_2]\!]$ $ve\ funE\ re\ lab$ =
  let
    val $(v_1, se_1) = \mathcal{C}\ [\![e_1]\!]\ ve\ funE\ re\ lab$
    val $(v_2, se_2) = \mathcal{C}\ [\![e_2]\!]\ ve\ funE\ re\ lab$
    val $([v_1', v_2'], ses)$ = **unify_ses**$([(v_1, se_1), (v_2, se_2)])$
  in
    (**insert_ses**$(v_1'\ bop\ v_2',\ ses)$,`none`)
  end

Constructors

Let expressions are inserted around a constructor.

$\mathcal{C}$ $[\![::e]\!]$ $ve\ funE\ re\ lab$ =
  let
    val $(v', se') = \mathcal{C}\ [\![e]\!]\ ve\ funE\ re\ lab$
  in
    (**insert_se**$(::v',\ se')$, `none`)
  end

Selection

The expression `#n(#m(v))` is translated into

```
let
  val v' = #m(v)
in
  #n(v')
end
```

$\mathcal{C}$ $[\![$`#`$n(e)]\!]$ $ve\ funE\ re\ lab$ =
  let
    val $(v', se') = \mathcal{C}\ [\![e]\!]\ ve\ funE\ re\ lab$
  in
    (**insert_se**(`#`$n(v')$, se'), `none`)
  end

letrec bound functions

$\mathcal{C}$ $[\![$`letrec` $f^{[f_0,\dots,f_m]}\ \langle x_1,\dots,x_n\rangle\ [\rho_0 : m_0,\dots,\rho_l : m_l]\ sma\ \rho$ = e_1 **in** e_2 **end**$]\!]$
$\quad ve\ funE\ re\ lab =$

let
- **val** new_lab = "f"
- **val** $sma' =$ **convert_sma**(sma, re, ρ)
- **val** $re' = re + \{\rho_0 \mapsto$ **mult**$(\mathtt{f}, m_0),\dots,\rho_l \mapsto$ **mult**$(\mathtt{f}, m_l)\}$
- **val** $(v',\ se') = \mathcal{C}\ [\![\rho]\!]\ \ ve\ funE\ re\ lab$
- **val** $c =$ **fresh_lvar**("clos")
- **val** $rv =$ **fresh_lvar**("regvec")
- **val** $cc = \{\text{clos} = c, \text{args} = [x_1,\dots,x_n], \text{regvec} = rv\}$
- **val** $ve' = \{f \mapsto c\} \cup \{f_i \mapsto \#i(c)\}_{i=0,\dots,m} \cup \{\rho_i \mapsto \#i(rv)\}_{i=0,\dots,l}$
- **val** $(lv_{f_i}, se_{f_i}) =$ **lookup_ve**$(f_i,\ ve) \qquad i = 0,\dots,m$
- **val** $([v'', lv'_{f_0},\dots,lv'_{f_m}], ses) =$ **unify_ses**$([(v', se'), (lv_{f_0}, se_{f_0}),\dots,(lv_{f_m}, se_{f_m})])$
- **val** $funE' = funE + \{\text{f} \mapsto new\_lab\}$
- **val** $e'_1 =$ **insert_se**$(\mathcal{C}\ [\![e_1]\!]\ \ ve'\ funE'\ re'\ new\_lab)$
- **val** _ = **add_new_fun**$(new\_lab,\ cc,\ e'_1)$

in

(**insert_ses**(`letrec` f_{new_lab} = $[lv'_{f_0},\dots,lv'_{f_m}]\ sma'\ v''$
`in`
insert_se$(\mathcal{C}\ [\![e_2]\!]\ \ ve\ funE'\ re\ lab)$
`end`, ses), `none`)

end

We assume the function to be closure implemented. The function f has scope in the body e_1 so we map f into the closure variable c in ve'. We store the function label in $funE$; then we know the label at the application points to f. We do not use fresh variables for the arguments $x_1,\dots,x_n$.

Applications

$\mathcal{C}\ [\![x_{ck}\ \langle e_1,\dots,e_n\rangle]\!]\ \ ve\ funE\ re\ lab =$

let
- **val** $(lv_i,\ se_i) = \mathcal{C}\ [\![e_i]\!]\ \ ve\ funE\ re\ lab \qquad i = 1,\dots,n$
- **val** $(lv, se) =$ **lookup_ve**$(x,\ ve)$
- **val** $([lv', lv'_1,\dots,lv'_n], ses) =$ **unify_ses**$([(lv, se), (lv_1, se_1),\dots,(lv_n, se_n)])$

in

(**insert_ses**$(lv'_{ck}\ \langle lv'_1,\dots,lv'_n\rangle\ \langle lv'\rangle\ \langle\rangle,\ ses)$, `none`)

end

$\mathcal{C}\ [\![f_{ck}\ \langle e_1,\dots,e_n\rangle\ [sma_0\ \rho_0,\dots,sma_l\ \rho_l]\ sma\ \rho]\!]\ \ ve\ re\ funE\ lab =$

let
- **val** $(lv_\rho, se_\rho) =$ **lookup_ve**$(\rho,\ ve)$

$$
\begin{array}{l}
\quad\quad \textbf{val } (lv_{\rho_i},\, se_{\rho_i}) = \mathcal{C}\ [\![\rho_i]\!]\ \ ve\ funE\ re\ lab \qquad i = 0, \ldots, l \\
\quad\quad \textbf{val } ([lv'_\rho, lv'_{\rho_0}, \ldots, lv'_{\rho_l}], ses_1) = \textbf{unify\_ses}([(lv_\rho, se_\rho), (lv_{\rho_0}, se_{\rho_0}), \ldots, (lv_{\rho_l}, se_{\rho_l})]) \\
\quad\quad \textbf{val } rv = \textbf{fresh\_lvar}(\text{"rv"}) \\
\quad\quad \textbf{val } f\_lab = funE(f) \\
\quad\quad \textbf{val } (lv_f, se_f) = \textbf{lookup\_ve}(f, ve) \\
\quad\quad \textbf{val } (lv_i,\, se_i) = \mathcal{C}\ [\![e_i]\!]\ \ ve\ funE\ re\ lab \qquad i = 1, \ldots, n \\
\quad\quad \textbf{val } ([lv'_f, lv'_1, \ldots, lv'_n], ses_2) = \textbf{unify\_ses}([(lv_f, se_f), (lv_1, se_1), \ldots, (lv_n, se_n)]) \\
\quad\quad \textbf{val } sma' = \textbf{convert\_sma}(sma, re, \rho) \\
\quad\quad \textbf{val } sma'_i = \textbf{convert\_sma}(sma_i, re, \rho_i) \qquad i = 0, \ldots, l \\
\quad \textbf{in} \\
\quad\quad (\textbf{insert\_ses}(\texttt{let} \\
\quad\quad\quad\quad\quad\quad \texttt{val } rv\ \texttt{=}\ [sma'_0\ lv'_{\rho_0}, \ldots, sma'_l\ lv'_{\rho_l}]_{\text{regvec}}\ sma'\ lv'_\rho \\
\quad\quad\quad\quad\quad \texttt{in} \\
\quad\quad\quad\quad\quad\quad \textbf{insert\_ses}(f\_lab_{ck}\ \langle lv'_1, \ldots, lv'_n\rangle\ \langle rv\rangle\ \langle\rangle\ \langle lv'_f\rangle\ \langle\rangle,\ ses_2) \\
\quad\quad\quad\quad\quad \texttt{end},\ ses_1),\ \texttt{none}) \\
\quad \textbf{end}
\end{array}
$$

We get the label for f by looking in *funE*. Note that we shorten life ranges by inserting let expressions at two different places.

Letregion

$$
\begin{array}{l}
\mathcal{C}\ [\![\texttt{letregion } \rho{:}m \texttt{ in } e \texttt{ end}]\!]\ \ ve\ funE\ re\ lab = \\
\quad\quad (\texttt{letregion } \rho{:}m \\
\quad\quad\ \texttt{in} \\
\quad\quad\quad \textbf{insert\_se}(\mathcal{C}\ [\![e]\!]\ \ ve\ funE\ (re + \{\rho \mapsto \textbf{mult}(\texttt{1}, m)\})\ lab) \\
\quad\quad\ \texttt{end},\ \texttt{none})
\end{array}
$$

Declaring lambda variables with `let`

$$
\begin{array}{l}
\mathcal{C}\ [\![\texttt{let val } \langle x_1, \ldots, x_n\rangle \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end}]\!]\ \ ve\ funE\ re\ lab = \\
\quad (\texttt{let} \\
\quad\quad \texttt{val } \langle x_1, \ldots, x_n\rangle \texttt{ = } \textbf{insert\_se}(\mathcal{C}\ [\![e_1]\!]\ \ ve\ funE\ re\ lab) \\
\quad\ \texttt{in} \\
\quad\quad \textbf{insert\_se}(\mathcal{C}\ [\![e_2]\!]\ \ ve\ funE\ re\ lab) \\
\quad\ \texttt{end},\ \texttt{none})
\end{array}
$$

Let expressions are inserted around unboxed records if necessary:

```
                  let
                    val v1 = e1
<e1,e2> =====>      val v2 = e2
                  in
                    <v1,v2>
                  end
```

$\mathcal{C}$ $[\![\langle e_1, \ldots, e_n \rangle]\!]$ *ve funE re lab* $=$
  let
    val $(lv_i, se_i) = \mathcal{C}\ [\![e_i]\!]$ *ve funE re lab* $i = 1, \ldots, n$
    val $([lv'_1, \ldots, lv'_n], ses) =$ **unify_ses**$([(lv_1, se_1), \ldots, (lv_n, se_n)])$
  in
    (**insert_ses**$(\langle lv'_1, \ldots, lv'_n \rangle,\ ses)$, `none`)
  end

Case

Sometimes it is necessary to insert a let around a `case` expression:

```
                           let
                             val v' = #5(v)
     case #5(v) of         in
       ::(x) => e2  ===>     case v' of
     | _     => e3             ::(x) => e2'
                             | _     => e3'
                           end
```

$\mathcal{C}$ $[\![$`case` e_1 `of` *pat* `=>` e_2 `| _ =>` $e_3]\!]$ *ve funE re lab* $=$
  let
    val $(e'_1, se'_1) = \mathcal{C}\ [\![e_1]\!]$ *ve funE re lab*
  in
    (**insert_se**(`case` e'_1 `of`
            pat `=>` **insert_se**($\mathcal{C}$ $[\![e_2]\!]$ *ve funE re lab*)
          `| _ =>` **insert_se**($\mathcal{C}$ $[\![e_3]\!]$ *ve funE re lab*), se'_1), `none`)
  end

3.5 Refinements of the Representation of Functions

It is hard to decide the right number of special cases to consider when deciding how a function should be represented. We have decided to either implement a function without a closure or with a closure holding all the free variables. However, it is possible to have functions where some of the free variables are accessible at the application point:

```
let                          f:fn<x,l1,clos>=>l1+(#1 clos)+x
  val l1 = 1                 let
in                             val l1 = 1
  let                        in
    val f =                    let
      let                        val clos =
```

```
    val l2 = 2
   in
    fn x => l1 + l2 + x
   end
 in
  f 2 (*1*)
 end
end
```

```
   let
    val l2 = 2
   in
    <l2>
   end
 in
  f <2, l1, clos>
 end
end
```

The free variables of `fn x => l1 + l2 + x` are `l1` and `l2`. At the application point `(*1*)` only `l1` is accessible. The function is known so it is only necessary to store the free variable `l2` in the closure and pass `l1` as an extra argument at the application point as in the above code at right. We have lifted the function to top level and explicitly created the closure `clos` for `f`.

It is possible to apply analyses that give more than two different call conventions (i.e., either with or without closures as in the above example). Wand and Steckler present a *selective* and *lightweight* closure conversion algorithm [56]. Only selected application points will use a closure (as in our approach) and some closures may be leightweight, that is, may only contain a subset of the free variables of the function; the rest are passed as arguments.

We use a simple approach to decide which functions should have the free variables passed as arguments and which should use a closure. A more aggressive but still simple algorithm is used in the SML/NJ compiler [5, Chapter 10] where a fixed point computation is done. It is based on the following assumptions:

1. escaping functions use a closure.

2. all mutual recursive functions (i.e., defined in the same `letrec`) use the same shared closure if they use a closure at all.

3. *known functions* (i.e., `letrec` bound functions) will not, to start with, require a closure. If function f calls the known function g then f will have its free variables extended with the free variables of g. This is necessary for f to pass them as arguments to g. This is more aggressive than our approach; we implement g with a closure if $\mathit{Free}(f) \subset \mathit{Free}(g)$, that is, we *never* extend the set of free variables of any function.

4. no function may have more that N arguments where N is the number of available argument registers.

5. if a known function f calls another known function g defined in the same `letrec` and g uses a closure then f will use the same closure (i.e., the shared closure). It is waste of time and registers to both pass the

> free variables as arguments and build a shared closure; why not use the shared closure that has to be build anyway. If the register pressure is limited then it is possible to assign closure elements to registers locally within each function so the number of fetches from the shared closure is limited.

A main difference from our approach is that we either implement *all* functions in the same `letrec` with a shared closure or none at all. In SML/NJ [5], some functions in the same `letrec` may use the shared closure and some may have the free variables passed as arguments.

Chapter 4

Linearization

After closure conversion the ClosExp language is linearized into a LineStmt language. The linearization phase makes sure that all value creating expressions are simple and that the value of such an expression is bound to a variable. The ClosExp expression

```
let
  val v = e1
in
  e2
end
```

does not bind `e2` to a variable. We want the linearization phase to translate it into a sequence of statements:

```
val v = e1;
val res = e2
```

where `res` is the variable holding the result value. If the above let expression is the body of a function then `res` is probably a machine register at runtime in which the result of the function is returned. The expressions `e1` and `e2` must be simple, that is, an expression that roughly corresponds to a KAM instruction (see Chapter 9).

4.1 Scope of Regions and Variables

It is important not to change the scope of regions. Consider the code:

```
let
  val x =
    letregion r in
      let
        val f = e1
```

```
          in
            f 2
          end
        end
    in
      e2
    end
```

The expression `letregion r` ... is not simple. The binding `val x =` ... has to be moved into the body of the inner `let` expression. If we keep the `letregion` and `let` construct as is, then we get:

```
letregion r in
  let
    val f = e1
    val x = f 2
  in
    e2
  end
end
```

However, this does not work because `e2` is still not bound to a variable and more important the scope of `r` has changed. We cannot keep the `let` construct because in linearized form we never have a value creating expression that is not bound to a variable, (i.e., we cannot have a body as in the `let` construct).

We want to keep the scoping of regions and if possible keep the `letregion` construct as is. We also want to sequentialise value bindings. If we let the result of `e2` be stored in `res` then we want something like this:

```
letregion r in
  val f = e1;
  val x = f 2
end;
val res = e2
```

The scope of region `r` is now preserved but we have lost the scope of variables. However, we must know the scope of variables in Chapter 7 where flush addresses on the stack are calculated. We make the scope visible with a new construct `scope` *lv* `in` *e* `end`. The `scope` construct introduces the variable *lv* but does not bind it to a value. An expression *e* is now a sequence of statements. Linearization of the above example then becomes:

```
scope res in
  scope x in
    letregion r in
```

```
        scope f in
          f := e1;
          x := f 2
        end
      end;
      res := e2
    end
  end
```

where the result `e2` is explicitly stored in a variable `res`.

We use *lv* `:=` *e* for assignment instead of `val` *lv* `=` *e* in order to emphasize that the variable *lv* is already created; we only initialize the variable. After linearization, we still have the invariant that a variable is initialized only once during program evaluation. This also holds for conditionals. Consider the program

```
let
  val x =
    case e1 of
      42 => e1
    | _  => e2
in
  e3
end
```

After linearization we get

```
scope res in
  scope x in
    case e1 of
      42 => x := e1
    | _  => x := e2;
    res := e3
  end
end
```

where `res` contains the result. The variable `x` is always initialized only one time after the `case` no matter what branch is taken.

4.2 Return Convention

In the closure conversion phase we made call conventions explicit. We also want return conventions to be explicit and this is done during linearization because thats where we bind the result expressions to a variable (i.e., remove the body of `let` expressions).

Consider the ClosExp function *foldl* from Figure 3.4.

$\lambda^{\texttt{fun}}_{\mathit{foldl}}$ {args=[f], rec_vec=rv, clos=c} `=>`
```
  let
    val fn_b = λ^fn_b [f, #1(rv), c] attop_ff #0(rv)
  in
    fn_b
  end
```

and an application

$\mathit{foldl}_{\texttt{funcall}}\ \langle lv_i\rangle\ \langle rv\rangle\ \langle\rangle\ \langle lv_j\rangle\ \langle\rangle$

The function returns the value *fn_b* which is bound to some variable at the application point. We make the return convention explicit at the function by extending the call convention *cc* to include a list of result variables:

$cc = \{$clos:	*LamVar*	*option*,
free:	*LamVar*	*list*,
args:	*LamVar*	*list*,
reg_vec:	*LamVar*	*option*,
reg_args:	*LamVar*	*list*,
res:	*LamVar*	*list* $\}$

Below we show the function after linearization (the extra let bindings are actually inserted by the closure conversion algorithm).

$\lambda^{\texttt{fun}}_{\mathit{foldl}}$ {args=[f], reg_vec=rv, clos=c, res=[res]} `=>`
```
  scope lv1, lv2, fn_b in
    lv1 := #1(rv);
    lv2 := #0(rv);
    fn_b := λ^fn_b [f, lv1, c] attop_ff lv2;
    res := fn_b
  end
```

At the application point we bind the result to a variable, *res* say:

$\langle res\rangle$ `:=` $\mathit{foldl}_{\texttt{funcall}}\ \langle lv_i\rangle\ \langle rv\rangle\ \langle\rangle\ \langle lv_j\rangle\ \langle\rangle$

4.3 LineStmt

The grammar for LineStmt is shown in Figure 4.1. The semantic objects are the same as for ClosExp, see Figure 3.7 on page 67. The only places where we have unboxed records are at application points. Unboxed records in `let` constructs are translated into a sequence of variable initialization constructs (i.e., x `:=` se).

$$
\begin{array}{lcl}
se & ::= & be\ a \\
 & | & x \\
 & | & c \\
 & | & \texttt{::}\ se \\
 & | & se_1\ bop\ se_2 \\
 & | & \texttt{\#}n(se)
\end{array}
$$

$$
\begin{array}{lcl}
ls & ::= & \texttt{scope}\ x\ \texttt{in}\ ls\ \texttt{end} \\
 & | & \texttt{letregion}\ b\ \texttt{in}\ ls\ \texttt{end} \\
 & | & x\ \texttt{:=}\ se \\
 & | & \texttt{case}\ se_1\ \texttt{of}\ pat\ \texttt{=>}\ ls_2\ \texttt{|}\ \_\ \texttt{=>}\ ls_3 \\
 & | & \langle x_1, \ldots, x_h \rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n \rangle\ \langle se_{clos} \rangle\ \langle se_{f_1}, \ldots, se_{f_m} \rangle \\
 & | & \langle x_1, \ldots, x_h \rangle\ \texttt{:=}\ lab_{ck}\ \langle se_1, \ldots, se_n \rangle\ \langle se_{reg} \rangle\ \langle se_{\rho_1}, \ldots, se_{\rho_l} \rangle \\
 & | & \qquad\qquad\qquad \langle se_{clos} \rangle\ \langle se_{f_1}, \ldots, se_{f_m} \rangle \\
 & | & ls_1\ \texttt{;}\ ls_2
\end{array}
$$

Figure 4.1: The grammar for LineStmt. Note, that the `letrec` and unboxed record constructs are removed from ls. We do not show the grammar for *top_decl*, *ck*, *sma*, *a*, *m*, *b*, *c*, *pat*, *bop* and *be*; they are the same as in ClosExp (see Figure 3.6 on page 66.)

4.4 Example

The linearized version of the function *foldl* from page 62 is shown in Section 4.2. The function *fn_xs* from page 63 translated into LineStmt is shown in Figure 4.2.

4.5 Algorithm $\mathcal{L}$

In this section we present algorithm $\mathcal{L}$ that translates a ClosExp expression into LineStmt. The top level declarations are translated by the function

$$\mathcal{L}^{\mathcal{T}} : TopDecl_{ce} \ \rightarrow \ TopDecl_{ls}$$

where $TopDecl_{ce}$ is the set of top level declarations in ClosExp and $TopDecl_{ls}$ is the set of top level declarations in LineStmt. The non top level expressions are translated by the function

$$\mathcal{L} : ClosExp \ \times \langle Var\ seq \rangle \rightarrow \ LineStmt$$

where *LineStmt* is the set of linearised statements ranged over by *ls*.

```
λ^fn_fn_xs {args=[xs], clos=c, res=[res]} =>
  scope v942 in
    (case xs of
       nil => res := #1(c)
     | :: (v942) =>
         scope x, xs' in
           x := #0(v942);
           xs' := #1(v942);
           letregion r22:4 in
             scope k80 in
               letregion r24:4 in
                 scope k77 in
                   letregion r25:2 in
                     scope rv, lv1, lv2 in
                       rv := [atbot_lf r24, atbot_lf r22] atbot_lf r25;
                       lv1 := #2(c);
                       lv2 := #3(c);
                       ⟨k77⟩ := foldl_funcall ⟨lv1⟩ ⟨rv⟩ ⟨⟩ ⟨lv2⟩ ⟨⟩
                     end
                   end; (*r25*)
                   scope k79 in
                     scope lv3, k78, lv4 in
                       lv3 := #2(c);
                       ⟨k78⟩ := (lv3)_fncall ⟨x⟩ ⟨lv3⟩ ⟨⟩;
                       lv4 := #1(c);
                       ⟨k79⟩ := k78_fncall ⟨lv4⟩ ⟨k78⟩ ⟨⟩
                     end;
                     ⟨k80⟩ := k77_fncall ⟨k79⟩ ⟨k77⟩ ⟨⟩
                   end
                 end
               end; (*r24*)
               ⟨res⟩ := k80_fncall ⟨xs'⟩ ⟨k80⟩ ⟨⟩
             end
           end (*r22*)
         end)
  end
```

Figure 4.2: The function *foldl* from page 63 translated into LineStmt. Note variable lv_3 in the call to lv_3 that is used for both the code pointer and function closure. The function **lookup_ve** (page 74) will assign #2(c) to two fresh lambda variables and not lv_3 only. However, the function **unify_se** then unifies the two select expressions.

4.5.1 Top level declarations

We assume the number of values returned by each function to be known and the function

$$RC \ : Label \ \rightarrow Int$$

returns the number of return values for each function represented with its label.

$\mathcal{L}^{\mathcal{T}}$ $[\![\lambda^{\texttt{fun}}_{lab}$ cc `=>` $e]\!]$ $=$
  let
    val lv_i $=$ **fresh_lvar**("lv_i") $i = 1, \ldots, RC(lab)$
  in
    $\lambda^{\texttt{fun}}_{lab}$ $(cc + \{\text{res}{=}[lv_1, \ldots, lv_{RC(lab)}]\})$ `=>`
      $\mathcal{L}$ $[\![\mathrm{e}]\!]$ $\langle lv_1, \ldots, lv_{RC(lab)} \rangle$
  end

The top level function $\lambda^{\texttt{fn}}_{lab}$ cc `=>` e is translated likewise. We write $cc + \{\text{res}{=}[lv_1, \ldots, lv_{RC(lab)}]\}$ as an abbreviation for constructing a new call convention extended with the list of result variables. A sequence of top level declarations is translated by:

$$\mathcal{L}^{\mathcal{T}} [\![top\_decl_1 ; top\_decl_2]\!] = \mathcal{L}^{\mathcal{T}} [\![top\_decl_1]\!] ; \mathcal{L}^{\mathcal{T}} [\![top\_decl_2]\!]$$

4.5.2 Simple Expressions

$\mathcal{L}$ $[\![c]\!]$ $\langle lv \rangle = lv$ `:=` c

$\mathcal{L}$ $[\![x]\!]$ $\langle lv \rangle = lv$ `:=` x

$\mathcal{L}$ $[\![be\ a]\!]$ $\langle lv \rangle = lv$ `:=` $be\ a$

Boxed expressions are not changed in the translation.

$\mathcal{L}$ $[\![e_1\ bop\ e_2]\!]$ $\langle lv \rangle = lv$ `:=` $e_1\ bop\ e_2$

$\mathcal{L}$ $[\![\texttt{\#}n(e)]\!]$ $\langle lv \rangle = lv$ `:=` $\texttt{\#}n(e)$

4.5.3 Expressions

$\mathcal{L}$ $[\![$`letrec` f_{lab} `=` $be\ a$ `in` e `end`$]\!]$ $\langle \vec{lv} \rangle =$
  `scope` f `in`
    f `:=` $be\ a$;
    $\mathcal{L}$ $[\![e]\!]$ $\langle \vec{lv} \rangle$
  `end`

$\mathcal{L}$ $[\![e_{ck}\ \langle e_1, \ldots, e_n\rangle\ \langle e_{clos}\rangle\ \langle e_{f_1}, \ldots, e_{f_m}\rangle]\!]$ $\langle \vec{lv}\rangle =$
  $\langle \vec{lv}\rangle$ `:=` $e_{ck}\ \langle e_1, \ldots, e_n\rangle\ \langle e_{clos}\rangle\ \langle e_{f_1}, \ldots, e_{f_m}\rangle$

$\mathcal{L}$ $[\![lab_{ck}\ \langle e_1, \ldots, e_n\rangle\ \langle e_{reg}\rangle\ \langle e_{\rho_1}, \ldots, e_{\rho_l}\rangle\ \langle e_{clos}\rangle\ \langle e_{f_1}, \ldots, e_{f_m}\rangle]\!]$ $\langle \vec{lv}\rangle =$
  $\langle \vec{lv}\rangle$ `:=` $lab_{ck}\ \langle e_1, \ldots, e_n\rangle\ \langle e_{reg}\rangle\ \langle e_{\rho_1}, \ldots, e_{\rho_l}\rangle\ \langle e_{clos}\rangle\ \langle e_{f_1}, \ldots, e_{f_m}\rangle$

The lambda variables $\langle \vec{lv}\rangle$ are now explicitly marked as the return convention.

$\mathcal{L}$ $[\![$`letregion` b `in` e `end`$]\!]$ $\langle \vec{lv}\rangle =$
  `letregion` b `in` $\mathcal{L}$ $[\![e]\!]$ $\langle \vec{lv}\rangle$ `end`

$\mathcal{L}$ $[\![$`let val` $\langle x_1, \ldots, x_n\rangle$ `=` e_1 `in` e_2 `end`$]\!]$ $\langle \vec{lv}\rangle =$
  `scope` $x_1, \ldots, x_n$ `in`
    $\mathcal{L}$ $[\![e_1]\!]$ $\langle x_1, \ldots, x_n\rangle$`;`
    $\mathcal{L}$ $[\![e_2]\!]$ $\langle \vec{lv}\rangle$
  `end`

$\mathcal{L}$ $[\![\langle e_1, \ldots, e_n\rangle]\!]$ $\langle lv_1, \ldots, lv_n\rangle =$
  lv_1 `:=` e_1`;`
    $\vdots$
  lv_{n-1} `:=` e_{n-1}`;`
  lv_n `:=` e_n

$\mathcal{L}$ $[\![$`case` e_1 `of` c `=>` e_2 `| _ =>` $e_3]\!]$ $\langle \vec{lv}\rangle =$
  `case` e_1 `of`
    c `=>` $\mathcal{L}$ $[\![e_2]\!]$ $\langle \vec{lv}\rangle$
  `| _ =>` $\mathcal{L}$ $[\![e_3]\!]$ $\langle \vec{lv}\rangle$

$\mathcal{L}$ $[\![$`case` e_1 `of ::`(v) `=>` e_2 `| _ =>` $e_3]\!]$ $\langle \vec{lv}\rangle =$
  `scope` v `in`
    `case` e_1 `of`
      `::`(v) `=>` $\mathcal{L}$ $[\![e_2]\!]$ $\langle \vec{lv}\rangle$
    `| _ =>` $\mathcal{L}$ $[\![e_3]\!]$ $\langle \vec{lv}\rangle$
  `end`

We have two rules for the `case` construct because the pattern `::`(v) implicitly declares v and we therefore need a `scope` for v. We insert the `scope` outside the `case` which is fine in our grammar because we at most can have one declaration. However, it does not work in the general case. Consider the code

```
case e1 of
   ::(v) => e2
 | ::'(⟨v1,v2⟩) => e3
 | _ => e4
```

where we have two constructors `::` and `::'`. If the variables v, v_1 and v_2 are spilled then variable v can share stack space with the variables v_1 and v_2, that is, v and v_1, v_2 are not live at the same time. Introducing the `scope` construct outside the `case` makes this sharing impossible with the offset calculation algorithm used in Chapter 7. However, it is not a problem in the ML Kit because the `case` construct can only match constants so no variable is ever declared in a pattern in a `case` construct.

Chapter 5

Register Allocation

The phases preceding register allocation assume an infinite number of machine registers (i.e., variables). The closure conversion phase, for instance, introduces a new variable whenever a value is selected from a closure with no concern to how the value is stored at runtime (i.e., in a machine register or in a memory cell). The target computer assumes all values to either reside in machine registers or in memory.

The purpose of register allocation is to map the variables into machine registers, memory locations or both. On most computers, and especially RISC based computers, accessing values in memory is much more expensive than accessing values in registers. It is essential for the running time of the target program to map as many variables to machine registers as possible. It is not possible to always map all variables to registers and some variables must be stored in memory. We say such variables are *spilled*. Neither, is it possible to always obtain an optimal mapping from variables to registers and most algorithms are driven by heuristics directing the way registers are chosen for variables (and likewise how variables are chosed to be spilled).

A popular register allocation method is *graph coloring* [13]. An *interference graph* is built with nodes representing values (i.e., variables) and edges representing interferences. Two nodes that interfere may not be assigned the same register. There can be several reasons for two variables to interfere but the usual reason is that they are live at the same time. Given the interference graph and a set of colors we compute a coloring of the graph, that is, we assign colors to the nodes and no two nodes connected with an edge may be assigned the same color. The set of colors represent the set of machine registers.

Register allocation can either be done locally on each function (or basic block) called *intra–procedural* register allocation or globally on entire modules (or programs) called *inter–procedural* register allocation.[1]

[1] We use the term global for modules containing one or more functions. In many books on optimizing compilers the term global refers to function wide optimizations and local

An intra–procedural register allocator does not have information about register usage outside the function currently considered. It is therefore necessary that all local values are at fixed places (often memory) when control is transferred from one function to another. An inter–procedural register allocator may use information about several functions in the program and may therefore obtain better register mappings. For instance, it may be the case that at every call to a function f then a large set of registers do not contain live data and may therefore be used freely inside the function f; or if f only uses a few registers then the caller can use the other registers freely even though f is called within the live range of the values in the registers. A disadvantage with inter–procedural register allocators is that they often depend on control flow information which is difficult to obtain accurately in a language with unknown functions [12, 38].

The graph color algorithm has two major problems. It may be computationally hard to color the interference graph and a variable is either assigned a register or memory location for its entire life time. It may be better to assign a variable a register in one part of a function and a memory location in another part of the function. This may be done by *range splitting* (i.e., to use two variables for the same value with no overlapping life range) but this is not included in the graph color algorithm.

It is widely believed that graph coloring is a wise choise for register allocation. The average computation necessary for coloring can be made acceptable and generating variables with short life ranges solves the range splitting problem. For instance, the variables introduced by function **insert_se** in the closure conversion algorithm (Chapter 3) have small live ranges.

Other register allocation algorithms not based on graph coloring exists. They are often motivated by the need to solve the register allocation problem faster than is possible with graph coloring. For instance, an inter–procedural register allocator based on graph coloring may build a huge interference graph that is too costly to solve. Also, dynamic optimizations done at load time require the optimizations to be almost linear which is not the case with graph coloring.

A fast algorithm called *linear scan* uses the same liveness information as graph coloring but instead of building an interference graph it is used to compute a *lifetime interval* for each variable. A lifetime interval includes the code that starts at the program point where the variable is first defined and ends at the program point where it is last used. The lifetime intervals can be sorted according to the order of the code. A finite number of lifetime intervals are active at each program point and they all compete for a register. If there are more active lifetime intervals than registers then one or more variables, represented by the active lifetime intervals, are spilled.

A recent study of a well written linear scan based algorithm and a well

to optimizations on basic blocks only [1, 37].

written graph color based algorithm has shown that in general the linear scan approach is close to the quality of graph coloring [54]. However, the compile time diverges as the programs get larger with more competing register candidates, (i.e., more variables); on a program with 245 register candidates graph coloring used 0.4 seconds and linear scan 1.5 seconds. On a program with 6697 register candidates graph coloring used 15.8 seconds and linear scan 4.5 seconds.

Many phases in the compiler introduce statements that move values from one variable v_1 to another variable v_2. If v_1 and v_2 are assigned the same register then the move statement can be eliminated. We can traverse the program after register allocation and remove unnecessary move statements. However, we can do better. If there is no interference between the two variables v_1 and v_2 then we can assign the same register to the variables and thereby eliminate the move statement. This is called *coalescing*.

Our register allocator is intra–procedural and we believe graph coloring is the best choice because the interference graphs are no larger than they can be solved in acceptable time. The study discussed above has shown that the result is in general slightly better than a linear scan based algorithm. We use a variant of graph coloring with *coalescing* [6].

We try to merge two nodes in the interference graph if they are source and destination of a move statement and does not interfere. We merge the nodes only if we can prove that it is not harder to color the resulting interference graph, that is, if we have a K color–able interference graph then the resulting graph, with the two nodes merged, must still be K color–able. The algorithm described in this chapter is based on the one given by Appel [6, Chapter 11].

We first discuss the additions necessary to the intermediate language to express register allocation information. We then discuss call conventions (with caller and callee save registers), liveness analysis and explain the general principles used in the graph color algorithm; especially how moves are eliminated and how registers are chosen for variables. We do not discuss the algorithm in detail because it is explained elsewere [6].

5.1 Revised Grammar

A few additions are necessary to the grammar for LineStmt in order to include register allocation information. Normally, a register allocator returns a rewritten program where all variables have been replaced by machine registers. However, we do not remove lambda and region variables because they are needed in the succeeding phases. Instead we insert register allocation information into the code such that the code generator knows which registers the variables are mapped to. We keep the variables because we need the scope information (`scope` and `letregion` constructs) in the phase that calculates stack offsets, Chapter 7.

One could annotate the register allocation information in two ways giving a different level of flexibility.

1. For each `scope` *lv* `in` *e* `end` construct we can annotate the machine register to which the *lv* is mapped as follows:

   $$\texttt{scope}\ lv : phreg\ \texttt{in}\ e\ \texttt{end}$$

   This says that *lv* has been mapped into the machine register *phreg* in *e*.

2. We can also keep the `scope` construct as is, and then use a new construct

   $$e ::= \texttt{regalloc}\ lv\ \mapsto phreg$$

   that tells the code generator that variable *lv* has been assigned machine register *phreg*. The assignment is valid until the next `regalloc` construct reassigns *lv* to anoter machine register or cancels the assignment:

   $$e ::= \texttt{regalloc}\ lv\ \mapsto \texttt{none}$$

   This implicitly says that from now on *lv* is spilled, (i.e., `none` is not a valid color).

The first method is adequate for a graph color based register allocator working on entire function bodies as one entity because then any variable will be mapped to the same register during the entire function. The second method gives more flexibility because a variable may be mapped to different machine registers during its life time. However, the extra flexibility either requires a more flexible register allocator than graph coloring or a graph color algorithm that considers the code between two function calls as one entity instead of the entire function.

It is hard to see which is the best solution. We let the graph color algorithm work on an entire function body as one entity because it seems simpler. We then gain flexibility by splitting registers in caller and callee save registers which makes it possible to have registers live across function calls. This is difficult to achieve with a graph color algorithm working only on the code between function calls. We annotate the register allocation information on the `scope` constructs because it seems simpler and works with our graph color method.

5.1.1 Store Type

When calculating stack offsets we assign offsets to spilled variables only. The register allocator therefore annotate a *store type* (*sty*) on the `scope` construct.

$$\begin{array}{rcl} sty & ::= & \texttt{stack} \\ & | & phreg \end{array}$$

where $phreg \in PhReg$ is a machine register. A *phreg* annotation says that the variable is never spilled and `stack` means that there exists a path through the function where the variable is spilled.[2]

$$e \quad ::= \quad \texttt{scope}\ x : sty\ \texttt{in}\ e\ \texttt{end}$$

The variables in a call convention are replaced by a *call convention store type*:

$$\begin{array}{rcl} cc\_sty & ::= & LamVar : \texttt{stack} \\ & | & phreg \end{array}$$

where lv:`stack` means that the parameter lv is passed in the call convention on the stack and *phreg* says that the parameter (that was denoted by a variable v before register allocation) is passed in the machine register *phreg*. The variable v is removed from the call convention for reasons discussed in the next section. A call convention is then as follows:

$$\begin{array}{rll} cc = \{ & \text{clos:} & \text{cc\_sty}\ \mathit{option}, \\ & \text{free:} & \text{cc\_sty}\ \mathit{list}, \\ & \text{args:} & \text{cc\_sty}\ \mathit{list}, \\ & \text{reg\_vec:} & \text{cc\_sty}\ \mathit{option}, \\ & \text{reg\_args:} & \text{cc\_sty}\ \mathit{list}, \\ & \text{res:} & \text{cc\_sty}\ \mathit{list}\ \} \end{array}$$

where for instance, *args*:*cc_sty list* is a list of call convention store types each denoting an argument parameter. At this point it is important to note that a variable denoting a parameter passed on the stack has been declared, (i.e., a stack slot has been reserved for the variable). A parameter passed in a machine register is not bound to a variable (i.e., the variable has been removed from the call convention). However, it may be the case that the machine register needs to be spilled and this requires a slot on the stack to be reserved. A stack slot is reserved by explicitly introducing the removed variable (with `scope`) in the body of the function and bind it to the machine register that it was replaced with. This is explained in the next section. We let CC_{sty} denote the set of call conventions with store types inserted ranged over by cc_{sty}.

It is not necessary to annotate store types on region variables at the `letregion` construct because they always denote a known slot in the activation record of the function in which they are declared. This holds for both finite and infinite regions.

[2] Because we are compiling Standard ML where variables are immutable and a variable keeps the same stack place while it is live then we store a variable only once in memory.

5.1.2 Call Conventions

Before building the interference graph we make dedicated machine registers explicit in the program (e.g., machine registers used for argument and result values at application points). This is necessary for the precolored variables, that is, variables that are dictated a specific register by the call conventions.

Parameters and results passed on the stack do not need explicit `scope` declarations in the body of the called function because they are already stored on the stack. Actually we simplify matters even further by not letting spilled parameters and results be part of the register allocator (i.e., if a parameter is spilled by a call convention then it remains spilled after register allocation). The simplification is justified by the fact that we reserve many machine registers for arguments and results and if the number of arguments or results exceeds the number of available machine registers then the register pressure is high and we believe the advantage of letting them be register allocated is limited.[3]

Parameters and results passed in machine registers may be spilled during register allocation. Say that a function takes variable v as argument and the argument is passed in a machine register ph. Then v is removed from the call convention and declared in the body of the function with a `scope` construct. At entry to the function we assign v the value of ph. It is then possible for the register allocator to spill the argument variable v and load ph with another value. It is also possible that v is assigned another machine register in which case the move statement becomes a copy between two registers. It may also be the case that v and ph are coalesced in which case the assignment is removed. It all depends on the register pressure in the body of the function.

There are two main reasons for moving v out of the call convention and into the body of the function: if v is spilled then the `scope` construct automatically reserves a slot in the activation record by the calculate offset phase (Chapter 7) and the variable v is only declared once. It would be confusing to have the same variable present in both the call convention and in a `scope` construct because where is the variable then declared? It is not declared in the call convention because the parameter is passed in a register!

Assuming we have two machine registers for passing arguments ($phreg_1$ and $phreg_2$) and one for the result ($phreg_1$) then the slightly modified function *foldl* from page 89 may be rewritten as the function below. We have added another result variable res_2 that is returned on the stack to illustrate passing both arguments and results in registers and on the stack.

$\lambda^{\mathtt{fun}}_{foldl}$ {args=[$phreg_1$], recvec=$phreg_2$, clos=c:`stack`,
res=[$phreg_1$,res_2:`stack`]} `=>`

[3] The number of available registers depends on the target architecture which may either be a RISC or CISC machine.

```
scope res₁ in
  scope f, rv in
    f := phreg₁;
    rv := phreg₂;
    scope lv₁, lv₂, fn_b in
      lv₁ := #1(rv);
      lv₂ := #0(rv);
      ⟨fn_b⟩ := λ^fn_b [f, lv₁, c] attop lv₂;
      res₁ := fn_b;
      res₂ := fn_b
    end
  end;
  phreg₁ := res₁
end
```

5.1.3 Resolving Call Conventions

Given a call convention as defined on page 57 with variables donoting argument and result parameters we must convert it into a call convention that specifies how the parameters are passed, that is, either on the stack on in registers.

The function

$$\mathbf{resolve\_cc} : CC \rightarrow \overbrace{(LamVar \ \times \ PhReg)\,list}^{precolored\ arguments} \times \overbrace{(LamVar \ \times \ PhReg)\,list}^{precolored\ results} \times CC_{sty}$$

makes this conversion and returns a list of precolored argument variables, precolored result variables and a call convention with store type annotations inserted. The compiler supports only one call convention, the standard call convention, but any number of call conventions are easily supported. In case more than one call convention is used, then an analysis deducing the call conventions must be implemented. All functions which can be called at the same application point must have the same call convention.

Assume, we have a list of machine registers for arguments *args_phreg* and a list of machine registers for results *res_phreg*. We then assign the variables in a call convention argument and result registers as follows. Result variables are assigned machine registers in the order left to right. If res=$[lvr_1, \ldots, lvr_n]$ and we have $m = |res\_phreg| \leq n$ then the first m variables $(lvr_1, \ldots, lvr_m)$ in res are assigned a register from *res_phreg*.

Argument parameters are resolved with the following order (priority): clos, reg_vec, args, free, reg_args. For each entry, the variables are assigned registers from *args_phreg*.

For example, with $args\_phreg = [q_1, \ldots, q_9]$, $res\_phreg = [p_1, p_2]$ and call convention

$$
\begin{aligned}
cc = \{\text{clos} = \quad & c, \\
\text{free} = \quad & [f_1, f_2], \\
\text{args} = \quad & [a_1, \ldots, a_9], \\
\text{reg\_vec} = \quad & \text{rv}, \\
\text{reg\_args} = \quad & [], \\
\text{res} = \quad & [r_1, \ldots, r_4]\}
\end{aligned}
$$

then **resolve_cc**(cc) returns

$$
\begin{aligned}
&([(c, q_1), (rv, q_2), (a_1, q_3), \ldots, (a_7, q_9)], \\
&[(r_1, p_1), (r_2, p_2)], \\
&\{\text{clos} = \quad q_1, \\
&\ \text{free} = \quad [f_1 : \texttt{stack}, f_2 : \texttt{stack}], \\
&\ \text{args} = \quad [q_3, \ldots, q_9, a_8 : \texttt{stack}, a_9 : \texttt{stack}], \\
&\ \text{reg\_vec} = \quad q_2{:}, \\
&\ \text{reg\_args} = \quad [], \\
&\ \text{res} = \quad [p_1, p_2, r_3 : \texttt{stack}, r_4 : \texttt{stack}]\})
\end{aligned}
$$

The arguments a_8, a_9, f_1, f_2 and results r_3, r_4 are passed on the stack and the other variables are removed from the call convention and returned as either precolored argument or precolored result variables. For instance, result variable r_1 is precolored as register p_1.

5.1.4 Rewriting LineStmt

As discussed above we must declare the precolored variables with `scope` constructs such that a stack slot can be reserved in the case they are spilled. We must also make sure that arguments and results, at application points, are passed in either registers or on the stack specified by the call convention. In this section we show how to rewrite the body of a function such that this is explicit. We believe it is better to have such information explicit in the term because it makes it easier to implement the register allocation algorithm and see what happens, (e.g., why is an argument variable coalesced with register ph_1 and not ph_2). The extra statements inserted does not effect the resulting program because if they were not inserted then we would have to incorporate the same information into the register allocation algorithm and the algorithm would be harder to comprehend. We hope, that most of the extra moves are removed by coalescing.

Given the function **resolve_cc** we can rewrite a LineStmt program to be on the same form as *foldl* on page 100.

We use the three functions:

resolve_args : $(LamVar \times PhReg)\ list \rightarrow LineStmt$
resolve_res : $(LamVar \times PhReg)\ list \rightarrow LineStmt$
resolve_app : $LineStmt \rightarrow LineStmt$

Declaring Argument and Result Variables

The function **resolve_args** translates all argument variables passed in machine registers into a sequence of assignments. If **resolve_cc** is called with precolored argument variables: $[(a_1, r_1), \ldots, (a_n, r_n)]$ then the sequence

$$\begin{array}{l} a_1 \texttt{ := } r_1\texttt{;} \\ \quad\vdots \\ a_n \texttt{ := } r_n \end{array}$$

is returned. The function **resolve_res** does the opposite. Given the precolored result variables $[(a_1, r_1), \ldots, (a_n, r_n)]$ as argument the sequence of assignments returned is

$$\begin{array}{l} r_1 \texttt{ := } a_1\texttt{;} \\ \quad\vdots \\ r_n \texttt{ := } a_n \end{array}$$

Application Points

The function **resolve_app** resolves an application using a variant of **resolve_cc** (called **resolve_app_cc**) working on call conventions containing simple expressions and not variables only. The argument brackets are converted into a temporary call convention and a list of argument and result expressions passed in machine registers are returned from **resolve_app_cc**.[4] A sequence of copies between expressions and machine registers are then inserted around the application. Consider the application

$$\langle x_1, \ldots, x_n \rangle \texttt{ := } se_1\ \langle se_2, \ldots, se_m \rangle\ \langle se_1 \rangle\ \langle\rangle$$

The application is converted into the term

$$\begin{array}{l} arg_1 \texttt{ := } se_1\texttt{;} \\ \quad\vdots \\ arg_k \texttt{ := } se_k\texttt{;} \\ \langle res_1, \ldots, res_h, x_{h+1}, \ldots, x_n \rangle \ \texttt{ :=} \\ \qquad\qquad\qquad arg_1\ \langle arg_2, \ldots, arg_k, se_{k+1}, \ldots, se_m \rangle\ \langle arg_1 \rangle\ \langle\rangle\texttt{;} \\ x_1 \texttt{ := } res_1\texttt{;} \\ \quad\vdots \\ x_{h-1} \texttt{ := } res_{h-1}\texttt{;} \\ x_h \texttt{ := } res_h \end{array}$$

[4]The function **resolve_app_cc** is identical to **resolve_cc** except that **resolve_app_cc** has simple expressions in each entry of the call convention and not variables.

where arg_i and res_j are machine registers. Argument and result values not assigned machine registers by the standard call convention are passed on the stack (e.g., $se_{k+1}, \ldots, se_m$ and $x_{h+1}, \ldots, x_n$). Notice that the translation preserves the property that machine registers are never live across an application. This property is used in the phase that inserts flush statements, Chapter 6. Note, that we never insert assignments after a tail call; the assignments are never executed.

Translation

The LineStmt program is translated with the function

$$\mathcal{CC}^{\mathcal{T}} : TopDecl_{ls} \ \rightarrow TopDecl_{ls}$$

working on top level declarations and the body of each function is translated by

$$\mathcal{CC} : \mathit{LineStmt} \ \rightarrow \mathit{LineStmt}$$

Below we show the interesting cases:

$\mathcal{CC}^{\mathcal{T}}$ $[\![\lambda^{\texttt{fun}}_{lab}\ cc$ `=>` $ls]\!]$ $=$
  let
    val $(args,\ res,\ cc_{sty})$ $=$ **resolve_cc**(cc)
  in
    $\lambda^{\texttt{fun}}_{lab}\ cc_{sty}$ `=>`
      `scope` **to_lvar_seq**(res) `in`
        `scope` **to_lvar_seq**$(args)$ `in`
          resolve_args$(args)$`;`
          $\mathcal{CC}$ $[\![ls]\!]$
        `end;`
        resolve_res(res)
      `end`
  end

The function **to_lvar_seq** transforms a list $[(a_1,r_1), \ldots, (a_n,r_n)]$ into the sequence $a_1, \ldots, a_n$ (i.e., we declare the lambda variables that are mapped into machine registers).

$$\mathcal{CC}\ [\![ls_1 ; ls_2]\!] \ = \mathcal{CC}\ [\![ls_1]\!]\ ;\ \mathcal{CC}\ [\![ls_2]\!]$$

We do a forward scan of the program.

$\mathcal{CC}$ $[\![\langle x_1, \ldots, x_n\rangle$ `:=` $se_1\ \langle se_2, \ldots, se_m\rangle\ \langle se_1\rangle\ \langle\rangle]\!]$ $=$
  resolve_app$(\langle x_1, \ldots, x_n\rangle$ `:=` $se_1\ \langle se_2, \ldots, se_m\rangle\ \langle se_1\rangle\ \langle\rangle)$

```
λ^fn_fn_xs {args=[ph2], clos=ph1, res=[ph1]} =>
scope res in
 scope xs, c in
  c := ph1;
  xs := ph2;
  scope v942 in
   (case xs of
     nil => res := #1(c)
    |:: (v942) =>
      scope x, xs' in
       x := #0(v942);
       xs' := #1(v942);
       letregion r22:4 in
        scope k80 in
         letregion r24:4 in
          scope k77 in
           letregion r25:2 in
            scope rv, lv1, lv2 in
             rv := [atbot_lf r24, atbot_lf r22] atbot_lf r25;
             lv1 := #2(c);
             lv2 := #3(c);
             ph1 := lv2;
             ph2 := rv;
             ⟨ph1⟩ := foldl_funcall ⟨lv1⟩ ⟨ph2⟩ ⟨⟩ ⟨ph1⟩ ⟨⟩;
             k77 := ph1
            end
           end; (*r25*) end
```

Figure 5.1: Part one of the function *fn_xs* with the precolored variables made explicit and application points resolved. We have *args_phreg*=$[ph_1, ph_2]$ and *res_phreg*=$[ph_1]$. No arguments or results are passed on the stack.

```
          scope k79 in
           scope lv3, k78, lv4 in
            lv3 := #2(c);
            ph1 := lv3;
            ph2 := x;
            ⟨ph1⟩ := (lv3)fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
            k78 := ph1;
            lv4 := #1(c);
            ph1 := k78;
            ph2 := lv4;
            ⟨ph1⟩ := k78fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
            k79 := ph1
           end;
           ph1 := k77;
           ph2 := k79;
           ⟨ph1⟩ := k77fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
           k80 := ph1
          end
         end
        end; (*r24*)
        ph1 := k80;
        ph2 := xs';
        ⟨ph1⟩ := k80fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
        res := ph1
       end
      end (*r22*)
     end)
  end
 end;
 ph1 := res
end
```

Figure 5.2: Part two of the function *fn_xs* with the precolored variables made explicit and application points resolved.

A `letrec` application is done similarly.

With the LineStmt program in this form we can build the interference graph. Figure 5.1 and 5.2 shows the example function *fn_xs* from Figure 4.2 on page 91 with precolored variables made explicit.

Precolored registers do not spill so their live range must be small. This is indeed the case in *foldl* where the argument registers are stored in argument variables at entry to the function and result registers are given their value at exit from the function. The live ranges are also minimized around applications.

5.2 Dummy Register Allocator

A nice effect of having call conventions explicit in the LineStmt code is that we can compile the program without using a register allocator or at least use a trivial register allocator only. Our trivial register allocator spills all variables, that is, annotate all `scope` introduced variables with `stack`. The code generator then inserts load and store code such that only temporary registers are used besides the registers used in the call convention.

We found it invaluable to use such a simple register allocator in the early stages of the compiler because we did not have a complicated register allocator algorithm to debug while debugging the mandatory phases of the backend. Also, the garbage collector was easier to debug without a complicated register allocator.

5.3 Liveness Analysis

To build the interference graph we need a liveness analysis that, for each program point, computes the set of live variables, that is, the variables that contain a value that is needed for the remainder of the computation.

For imperative languages the liveness analysis is normally based on a control flow graph that, for each statement, has an edge to the statements that may follow. If the program contains loops, then the control flow graph contains cycles. Liveness is found by a fixed point computation on the control flow graph. We compute liveness information locally for each function and because loops are implemented with function calls in Standard ML then we do not have loops locally in each function. We can then compute the liveness information by a single backward scan of the LineStmt program.

Consider the LineStmt code

```
scope x, y, z in
  x := 0        (1)
  y := x + 42   (2)
  z := x + y    (3)
```

```
x := 42      (4)
z := z + x   (5)
    ⋮
end
```

The liveness information is computed using *def* and *use* sets of each statement, that is, the set of variables that are defined and used respectively when the statement is evaluated. Note that the same variable can be used and defined in the same statement (e.g., z `:=` z `+` x). We let the functions **def** and **use** return the set of defined and used variables for the statement given as argument. For assignment and addition we have:

$$\begin{array}{l}\mathbf{def}(v_1 \texttt{ := } v_2 \texttt{ + } v_3) = \{v_1\} \\ \mathbf{use}(v_1 \texttt{ := } v_2 \texttt{ + } v_3) = \{v_2,\ v_3\}\end{array}$$

The **def** and **use** functions for *LineStmt* are defined in Section 5.3.1.

The *live range* of a variable is the code between the first definition and last use of the variable. The same variable may be defined at different program points in imperative languages but a variable is defined only once in Standard ML. LineStmt is not a functional language but because it is a linearized version of a functional language then we have a similar property, that is, a variable is initialized only once during program evaluation, see Section 4.1 on page 86. This is a unique property because then a variable needs to be stored in memory at most one time even though it is spilled.

The live ranges for x, y and z in the example program above are

$$\begin{array}{l} z = \{4 \mapsto 5,\ 3 \mapsto 4\} \\ x = \{4 \mapsto 5,\ 2 \mapsto 3,\ 1 \mapsto 2\} \\ y = \{2 \mapsto 3\} \end{array}$$

Note that the above program cannot be a linearized version of a ClosExp program because x is defined twice. The set of live variables at each statement is computed by looking at each statement in reverse order. For each statement we maintain a set *live* that is the set of live variables at entry to the statement. If we assume z to be live after statement 5 (i.e., $live(6)=\{z\}$), then we have

$$\begin{array}{lcl} live(5) & = & use(5) \cup (live(6) \setminus def(5)) = \{z,x\} \cup (\{z\} \setminus \{z\}) = \{z,x\} \\ live(4) & = & use(4) \cup (live(5) \setminus def(4)) = \{\} \cup (\{z,x\} \setminus \{x\}) = \{z\} \end{array}$$

and in general

$$live(i) \quad = \quad use(i) \cup (live(i+1) \setminus def(i))$$

5.3.1 Def and Use Sets for LineStmt

The functions **def** and **use** for LineStmt constructs are easily defined. The table below shows the def and use sets for various simple constructs:

Construct	def	use
$sma\ lv$	$\{\}$	$\{lv\}$
$sma\ \rho$	$\{\}$	$\{\}$
$\rho : m$	$\{\}$	$\{\}$
c	$\{\}$	$\{\}$
$::x$	$\{x\}$	$\{\}$

An allocation point $sma\ \rho$ does not define ρ because ρ is a constant at runtime (i.e., an offset from the stack pointer found at compile time). The pattern $::x$ defines the variable x.

Boxed expressions do not define variables. The use sets are calculated as follows:

$$\begin{array}{l}
\mathbf{use}((se_1,\ldots,se_n)) = \bigcup_{i=1,\ldots,n} \mathbf{use}(se_i) \\
\mathbf{use}(\lambda^{lab}\ [se_{f_1},\ldots,se_{f_n}]) = \bigcup_{i=1,\ldots,n} \mathbf{use}(se_{f_i}) \\
\mathbf{use}([a_1,\ldots,a_n]_{regvec}) = \bigcup_{i=1,\ldots,n} \mathbf{use}(a_i) \\
\mathbf{use}([se_{f_1},\ldots,se_{f_n}]_{sclos}) = \bigcup_{i=1,\ldots,n} \mathbf{use}(se_{f_i})
\end{array}$$

Def and use sets for simple expressions are found below:

Construct	def	use
$be\ a$	$\{\}$	$\mathbf{use}(be) \cup \mathbf{use}(a)$
x	$\{\}$	$\{x\}$
c	$\{\}$	$\{\}$
$::\ se$	$\{\}$	$\mathbf{use}(se)$
$se_1\ bop\ se_2$	$\{\}$	$\mathbf{use}(se_1) \cup \mathbf{use}(se_2)$
$\#n(se)$	$\{\}$	$\mathbf{use}(se)$

5.4 Interference Graph

With liveness information we can compute the interference graph ig where nodes denote variables and we have an edge between two nodes if they interfere. Two variables v_1 and v_2 may interfere for several reasons but the usual situation is that v_1 and v_2 are live at the same time (i.e., they cannot be assigned the same register). Other interferences may come from instruction sets where some instructions can access a limited number of the registers only. Then the variables in the instruction interfere with all the registers that the instruction does not access. We only consider interferences of the first kind so two variables *interfere* iff there exists a program point where they are both live.

We let IG be the set of interference graphs ranged over by ig.

5.4.1 Unnecessary Copy Related Interferences

We give the assignment construct special treatment. A copy x := y does not make x and y interfere because they contain the same value and may therefore be assigned the same register. If, however, one of the registers, say x, is redefined while y is still live then an interference edge will be inserted.[5] This is indeed necessary because at that time they do not necessarily contain the same value.

Edges are added to an interference graph ig using, for each statement ls, the two rules

1. if ls is a copy d := s, and the *live* set (live out) is $\{lv_1, \ldots, lv_n\}$ then we add edges (d, lv_i) to ig for all lv_i not equal to s.

2. if ls is not a copy, defines variable d and the *live* set (live out) is $\{lv_1, \ldots, lv_n\}$, then we add the edges $(d, lv_1), \ldots, (d, lv_n)$ to ig.

The graph ig is undirected.

5.4.2 Move Related Nodes

For every copy d := s we insert a move related edge in ig. A *move related edge* is not an interference edge. Move related edges makes it possible to distinguish move related nodes (i.e., d and s) from non move related nodes. In the examples to come we use dotted edges for move related edges and normal edges for interference edges. A node can be categorized as either:

1. a move related node (i.e., the node has at least one move related edge and maybe other interference edges).

2. a non move related node (i.e., the node does not have any move related edges).

3. a constrained move related node (i.e., a node with a move related edge to a node n and an interference edge also to node n). Then it is impossible to coalesce the two nodes. We remove all move related edges from constrained move related nodes.

5.4.3 Building IG

With the **def** and **use** functions defined in Section 5.3.1 we can state the algorithm that computes the interference graph.

$$\mathcal{IG}^{\mathcal{T}} : TopDecl_{ls} \ \rightarrow IG$$

[5] This never happens because we are compiling Standard ML and not an imperative language.

works on top level declarations. The body of a function is handled by

$$\mathcal{IG} : \mathit{LineStmt} \ \times \mathcal{P}(\mathit{LamVar}) \times \mathit{IG} \rightarrow \mathcal{P}(\mathit{LamVar}) \times \mathit{IG}$$

where $\mathcal{P}(\mathit{LamVar})$ is ranged over by *live* and is the set of live variables.

Top Level Functions

$$\begin{array}{l}
\mathcal{IG}^{\mathcal{T}} \ [\![\lambda^{\texttt{fun}}_{lab} \ cc \ \texttt{=>} \ ls]\!] \ = \\
\quad \textbf{let} \\
\quad\quad \textbf{val} \ (live, ig) = \mathcal{IG} \ [\![ls]\!] \ \ \{\} \ \{\} \\
\quad \textbf{in} \\
\quad\quad ig \\
\quad \textbf{end}
\end{array}$$

The empty graph is written $\{\}$.

Statements

We do a backward scan on the statements and collect liveness information at the same time that we insert either interference or move related edges into ig.

$$\begin{array}{l}
\mathcal{IG} \ [\![\texttt{scope} \ x \ \texttt{in} \ ls \ \texttt{end}]\!] \ \ live \ ig = \mathcal{IG} \ [\![ls]\!] \ \ live \ ig \\
\mathcal{IG} \ [\![\texttt{letregion} \ b \ \texttt{in} \ ls \ \texttt{end}]\!] \ \ live \ ig = \mathcal{IG} \ [\![ls]\!] \ \ live \ ig \\
\mathcal{IG} \ [\![lv_1 \ \texttt{:=} \ lv_2]\!] \ \ live \ ig = \\
\quad (\{lv_2\} \cup (live \setminus \{lv_1\}), \\
\quad\ \textbf{addMoveEdge}(\textbf{addEdge}(ig, \{lv_1\}, live \setminus \{lv_2\}), lv_1, lv_2)) \\
\mathcal{IG} \ [\![lv \ \texttt{:=} \ (se_1, \ldots, se_n) \ a]\!] \ \ live \ ig \\
\quad ((live \setminus \{lv\}) \cup \textbf{use}((se_1, \ldots, se_n) \ a), \\
\quad\ \textbf{addEdge}(ig, \{lv\}, live \cup \textbf{use}((se_1, \ldots, se_n) \ a))) \\
\mathcal{IG} \ [\![lv \ \texttt{:=} \ se]\!] \ \ live \ ig = \\
\quad ((live \setminus \{lv\}) \cup \textbf{use}(se), \ \textbf{addEdge}(ig, \{lv\}, live)) \\
\mathcal{IG} \ [\![\texttt{case} \ se_1 \ \texttt{of} \ pat \ \texttt{=>} \ ls_2 \ \texttt{|} \ _ \ \texttt{=>} \ ls_3]\!] \ \ live \ ig = \\
\quad \textbf{let} \\
\quad\quad \textbf{val} \ (live_2, ig_2) = \mathcal{IG} \ [\![ls_2]\!] \ \ live \ ig \\
\quad\quad \textbf{val} \ (live_3, ig_3) = \mathcal{IG} \ [\![ls_3]\!] \ \ live \ ig_2 \\
\quad\quad \textbf{val} \ live_1 = live_2 \cup live_3 \\
\quad \textbf{in} \\
\quad\quad (\textbf{use}(se_1) \cup (live_1 \setminus \textbf{def}(pat)), \ \textbf{addEdge}(ig_3, \textbf{def}(pat), live)) \\
\quad \textbf{end} \\
\mathcal{IG} \ [\![\langle x_1, \ldots, x_n \rangle \ \texttt{:=} \ se_{\{\texttt{fncall},\texttt{funcall}\}} \ \langle se_1, \ldots, se_n \rangle \ \langle se_{clos} \rangle \ \langle se_{f_1}, \ldots, se_{f_m} \rangle]\!] \ \ live \ ig = \\
\quad ((\bigcup_{i \in \{ck, 1, \ldots, n, clos, f_1, \ldots, f_m\}} \textbf{use}(se_i)) \cup (live \setminus \{x_1, \ldots, x_n\}), \\
\quad\ \textbf{addEdge}(ig, \{x_1, \ldots, x_n\}, live \cup \{x_1, \ldots, x_n\})) \\
\mathcal{IG} \ [\![\langle x_1, \ldots, x_n \rangle \ \texttt{:=} \ se_{\{\texttt{fnjmp},\texttt{funjmp}\}} \ \langle se_1, \ldots, se_n \rangle \ \langle se_{clos} \rangle \ \langle se_{f_1}, \ldots, se_{f_m} \rangle]\!] \ \ live \ ig =
\end{array}$$

$$
\begin{array}{l}
\quad\quad ((\bigcup_{i\in\{ck,1,\ldots,n,clos,f_1,\ldots,f_m\}} \mathbf{use}(se_i))), \\
\quad\quad\ \mathbf{addEdge}(ig, \{x_1,\ldots,x_n\}, \{x_1,\ldots,x_n\})) \\
\mathcal{IG}\ [\![ls_1\ ;\ ls_2]\!]\ \ live\ ig = \\
\quad \mathbf{let} \\
\quad\quad \mathbf{val}\ (live_2, ig_2) = \mathcal{IG}\ [\![ls_2]\!]\ \ live\ ig \\
\quad \mathbf{in} \\
\quad\quad \mathcal{IG}\ [\![ls_1]\!]\ \ live_2\ ig_2 \\
\quad \mathbf{end}
\end{array}
$$

The rule for `letrec` application is similar to the rule for ordinary application. Note that if more than one variable is defined as in an application then the defined variables interfere with themselves and the live variables.

The function

$$\mathbf{addEdge} : IG \times \mathcal{P}(LamVar) \times \mathcal{P}(LamVar) \to IG$$

adds interference edges between variables in the two variable sets. The function

$$\mathbf{addMoveEdge} : IG \times LamVar \times LamVar \to IG$$

adds a move related edge between the two variables.

5.5 Spilling

Spilling a variable normally requires rewritting the program where a new temporary will fetch the spilled variable and die right after the use. For instance, if variable `x` is spilled we have

```
y = x + z  ===x is spilled===> tmp = Fetch(x)
                                 y = tmp + z
```

and `tmp` has a tiny life range. However, the rewritten program has to go through the register allocation phase again (i.e., iterated register allocation). In the ML Kit version 2 (and version 3) [22, 49, 50] we simplified matters a bit because we never rewrite the program if a variable is spilled. Instead we reserve four registers that are not used by the register allocator.

The four registers are reserved for the purpose of compiling simple expressions and fetching spilled variables. For instance, the machine code to allocate a region requires a few extra registers. Given any statement with all variables spilled it is possible to compile it with the use of no more than four registers.

The simplification gives a less good register usage because the four reserved registers are never used by the register allocator even though they are not needed to compile all the statements. It may be solved by letting all registers be available to the register allocator. If the code generator requires additional registers to compile a statement then the requirements may be build into the interference graph, that is, let the statements interfere with all registers that are needed to compile the statements. This can be controlled by the **def** and **use** functions. We do not use this method and still reserve a number of registers for the code generator.[6]

It is not necessary to insert spill code explicitly and use iterated register allocation because we reserve temporaries for the code generator. This speeds up the register allocator and we believe the performance degradation to be moderate (i.e., the register allocator still has enough registers on a RISC machine).

5.5.1 Statements Are Not Like Three Address Instructions

Register allocation is normally performed on three address like instructions which are simpler than the statements in LineStmt. This means that we perform register allocation on a term with less nodes than if we used three address instructions which is good for the running time of the algorithm but bad for the live ranges of variables containing intermediate values.

Consider the following assignment of a record expression where all variables $x_1, \ldots, x_n$ are spilled.

$$x \texttt{:=(} x_1, \ldots, x_n \texttt{)}$$

If we have K machine registers and $n > K$ then it is impossible to store all variables in temporary registers. It is not possible to perform register allocation on

$$
\begin{array}{l}
\texttt{scope}\ tmp_1, \ldots, tmp_n\ \texttt{in} \\
\quad tmp_1\ \texttt{:=}\ x_1\texttt{;} \\
\quad\quad \vdots \\
\quad tmp_n\ \texttt{:=}\ x_n\texttt{;} \\
\quad \mathrm{x}\ \texttt{:= (}tmp_1, \ldots, tmp_n\texttt{)} \\
\texttt{end}
\end{array}
$$

where all temporaries have to be machine registers. We reserve n registers for a record with n elements which is an unreasonable amount. If we were using three address code then an n element record would probably only require two registers; one index register and one for the elements. This is not possible in LineStmt because we must put all elements in registers before allocating the record. However, this can be solved by using an alloc primitive that can allocate memory for the record and an update primitive to initialize each element:

[6] The discussion assumes that the targe machine is a RISC machine. For CISC machines with less registers it may not be necessary to reserve temporary registers because most instructions can probably fetch from memory, perform the operation and then store the result in memory again if necessary.

```
scope tmp_1 in
  x := alloc(n);
  tmp_1 := x_1;
  update(x, 0, tmp_1);
      ⋮
  tmp_1 := x_n;
  update(x, n − 1, tmp_{n−1});
end
```

We will probably use this technique in the future and thereby reduce the register pressure.

5.6 Graph Coloring With Coalescing

After the interference graph ig is built the register allocator cycles though the following phases that remove nodes from ig and pushes them on a stack. Let K be the number of available machine registers (i.e., colors). The *degree* of a node is the number of interference edges on the node. The move related edges are not part of the degree.

Simplify: All non move related nodes of degree less than K are removed from ig.

Coalesce: If simplify is not possible then do conservative coalescing (see below) and return to simplify again.

Freeze: If neither simplify nor coalescing is possible, then choose a move related node of low degree and remove the move related edges from that node. The node is not a move related node anymore and may be removed by the simplify phase which is applied next.

Spill: If we cannot simplify, coalesce or freeze a node then we have to spill a variable. We calculate the priority (see below) of the remaining nodes and remove the variable with the lowest priority. We then try simplify again.

When all nodes are removed from ig we start popping nodes from the stack. Each node is inserted in ig again and assigned a color that is not assigned to any of the neighbors of the inserted node. We assign colors from the caller save and callee save sets as described in Section 5.7 below. Spilled variables are also inserted in ig and we try to assign a color even though they were marked as *potential spills* on the stack. Sometimes it is possible to assign a color to a variable that was considered spilled during the simplify phase. For instance, if some of its neighbors share the same color or other neighbors are spilled too. If it is not possible to color a potential spilled variable we call it an *actual spilled* variable.

5.6.1 Conservative Coalescing

If an interference graph ig can be colored with M registers, then we call it *conservative coalescing* when we can union two nodes in ig and still be guaranteed that the resulting interference graph is M color–able. Appel gives two strategies and proves that they are conservative [6, page 232]:

Briggs: Two nodes n_1 and n_2 can be coalesced if the resulting node has fewer that K neighbors.

George: Two nodes n_1 and n_2 can be coalesced if for every neighbor m to n_1, either m has degree less than K or m is also a neighbor to n_2.

5.6.2 Spill Priority

To find an adequate node to spill we calculate a priority based on how many times the variable is used in the program and the degree. Variables with a large number of uses and low degree get a higher priority. Variables with low priority are spilled.

$$\mathbf{priority}(lv) = \frac{\mathbf{uses}(lv) + \mathbf{defs}(lv)}{\mathbf{degree}(lv)}$$

The functions **uses** and **defs** return the number of times the variable is used and defined in the function.

5.6.3 Implementation

Appel gives an efficient algorithm which can be adjusted to the specifics of LineStmt and the ML Kit [6]. As noted in the introduction Elsman has started the implementation in the ML Kit using the algorithm by Appel.

5.7 Caller and Callee Save Registers

If we have caller save registers only then it is necessary to save all live registers before an application. If we have callee save registers only then the called function must save all registers that it uses. Both methods may lead to an excessive number of stores. Consider the code

```
f(...) =
        ⋮
        x := 5;                    g(x) = y := x + 1;
        y := 6;                           a := y * 43;
        z := y + x;                       b := a + y;
        q := g(y);                        res := b
```

$$a := q + x;$$
$$b := q + z;$$
$$\vdots$$

With caller save registers only, we have to flush the registers to which z and x are bound at the application to g and fetch them afterwards. With callee save registers only we have to flush the registers to which y, a and b are bound in g and fetch them before we return the result res. With a combination of callee and caller save registers we can allocate x and z to callee save registers and avoid flushing them in f. In g we can bind y, a and b to caller save registers because their live range do not cross an application. Then we avoid flushing any variables at all.

Flushes across applications may be totally avoided by having the application expression define all caller save registers when the interference graph is created. If a variable is not live across a function call then it will likely be allocated to a caller save register. If a variable is live across the application then it interferes with all caller save registers and will either be assigned a callee save register or spilled. With caller save registers r_1, r_2 and callee save register r_3 a possible register assignment is: $z : r_3, x : \mathit{spill}, y : r_1, q : r_1, a : r_2$ and $b : r_1$. The variable x is spilled because we only have one callee save register and it is given to z which is also live across the call to g. Both z and x interfere with all caller save registers and hence one of them must be spilled.

We can do better because we may bind x to a caller save register and then store it across the function call. This is better than spilling where the variable is held in memory all the time. This is achieved by not having the call statement define all caller save registers. Instead we mark variables with a live range that crosses an application.[7] We then assign colors depending on the marks. Variables not crossing applications are assigned caller save registers if possible and variables crossing applications are assigned callee save registers if possible. With this method a possible coloring is: $y : r_1, z : r_3, x : r_2, q : r_1, a : r_2$ and $b : r_1$. Now x is assigned a caller save register.

As an example of how registers can be divided into caller and callee save registers we take a closer look at the HP PA-RISC architecture. The HP PA-RISC machine has 32 general purpose registers [41, 40, 39].[8] The Procedure Calling Convention manual [40] specifies the callee save registers to be $gr_3, \ldots, gr_{18}$ and caller save registers to be $gr_1, gr_{19}, \ldots, gr_{26}, gr_{28}$ and gr_{29}. The convention is used by compilers supported by HP. We do not have to use the same convention and it seems better to increase the number

[7] This is done similarly to algorithm $\mathcal{F}$ in Chapter 6.

[8] There are also floating point registers and they can also be divided into caller and callee save registers but we limit the discussion to general purpose registers only.

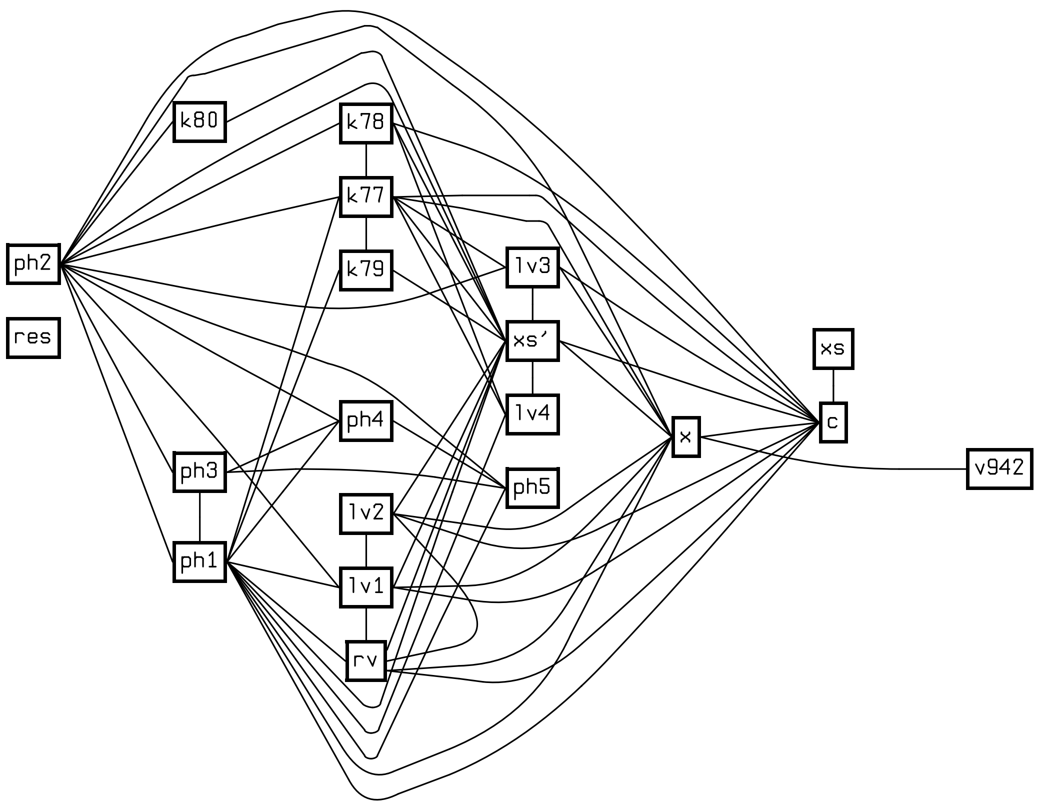

Figure 5.3: All interferences in function *fn_xs*.

of caller save registers. The reason is that all arguments and results must be passed in caller save registers and we expect to get a high demand on registers to be used in the call conventions.

We note that it may not be possible to be this generous with caller save registers on other architectures like X86 based machines.

5.8 Example

Figure 5.3 shows all interferences in the example function *fn_xs* from Figure 5.1 and 5.2. Figure 5.4 shows all non constrained move related nodes. We assume $K = 5$ with $ph1, ph2$ and $ph3$ as caller save registers and $ph4, ph5$ as callee save registers.

We can simplify node *v942* with only one interference edge and then coalesce the nodes *ph1*,*k80*,*lv2*,*res*,*lv3* and *k78* using coalescing strategy George. For example, node *ph1* interfere with the nodes *ph2*, *ph3*, *k77*, *k79*, *ph4*, *lv1*, *rv*, *xs'*, *lv4*, *ph5*, *x* and *c*. Node *k80* interfere with the nodes

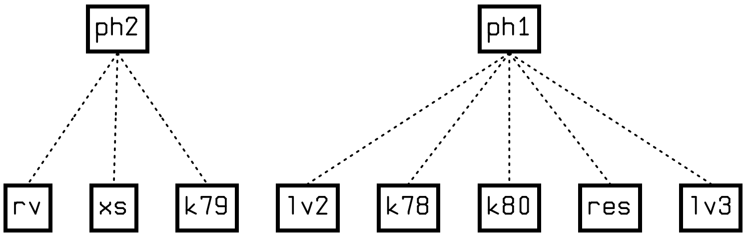

Figure 5.4: All move related nodes in function *fn_xs*. We have removed all constrained move related nodes.

ph2 and *xs'*. Because both nodes that interfere with *k80* (i.e., *ph2* and *xs'*) interfere with *ph1* then *k80* and *ph1* can be coalesced.

We can also coalesce the nodes *ph2*,*k79*,*xs* and *rv* using coalescing strategy George. At first it seems that we cannot coalesce *rv* with *ph2* because *lv2* does not interfere with *ph2*. But after *ph1* and *lv2* are coalesced then we can. The resulting graph is shown in Figure 5.5. There are no more move related nodes.

We can then simplify *lv4* with degree 3, see Figure 5.6. We do not have more low degree moves and no more move related nodes. We therefore spill a variable.

We calculate priorities for the 5 candidates

Variable	**Defs+Uses**	**Degree**	**Priority=**$\frac{Defs+Uses}{Degree}$
lv_1	2	5	0.40
$k77$	3	5	0.60
x	2	6	0.33
xs'	2	6	0.33
c	6	6	1.00

The variable xs', and x has lowest priority so we arbitrarily make xs' a potential spill. Then we simplify *lv1* with degree 4. The remaining variables *k77*, *x* and *c* can also be simplified. We have spilled and simplified in the following order:

$$stack = [\underline{c}, \underline{x}, \underline{k77}, lv1, \underline{xs'}, lv4, v942]$$

with c on top of the stack. We have underlined the variables with a live range crossing at least one application. We want to assign the underlined variables callee save registers (i.e., ph_4, ph_5) and the non underlined variables caller save registers (i.e., $ph_1, ph2$ and $ph3$) whenever possible. We pop and assign

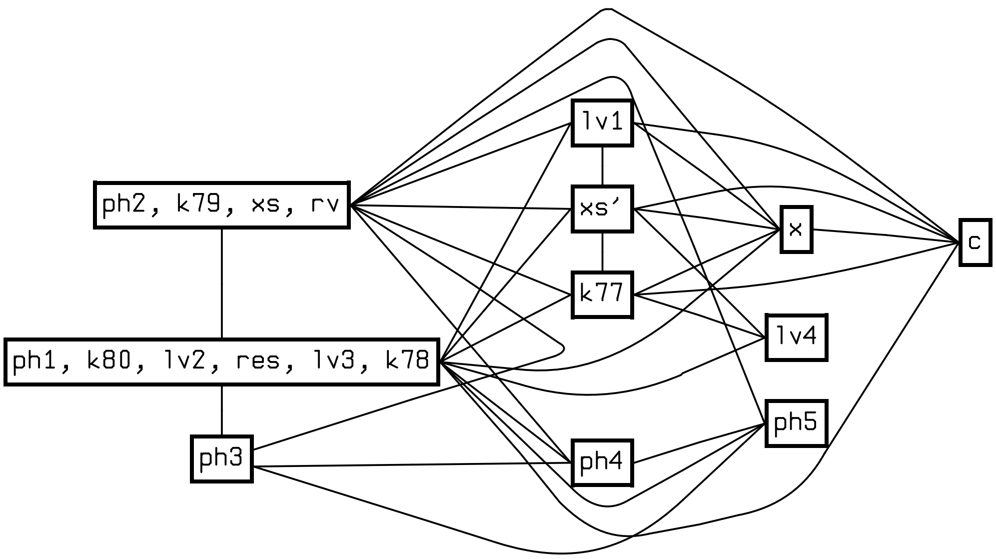

Figure 5.5: We have simplified *v942* and coalesced the nodes *ph1*,*k80*,*lv2*,*res* and *lv3*. We have also coalesced the nodes *ph2*,*k79* and *xs*.

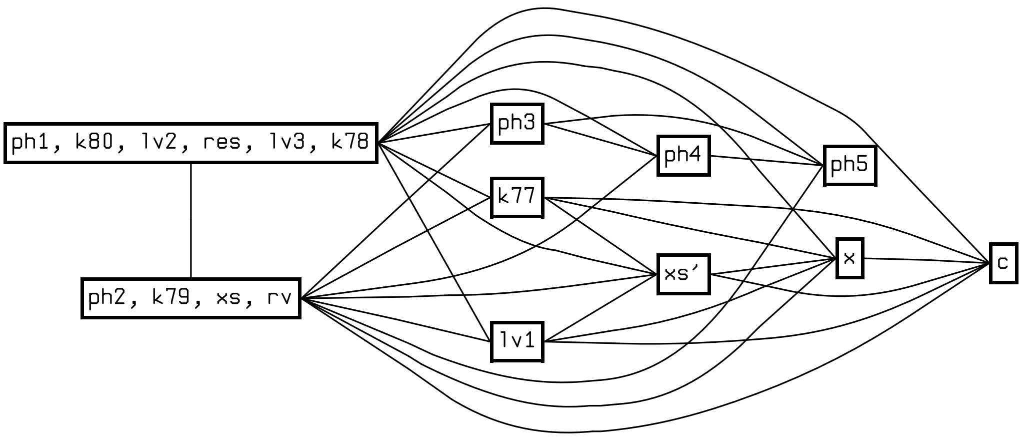

Figure 5.6: We have simplified *lv4*.

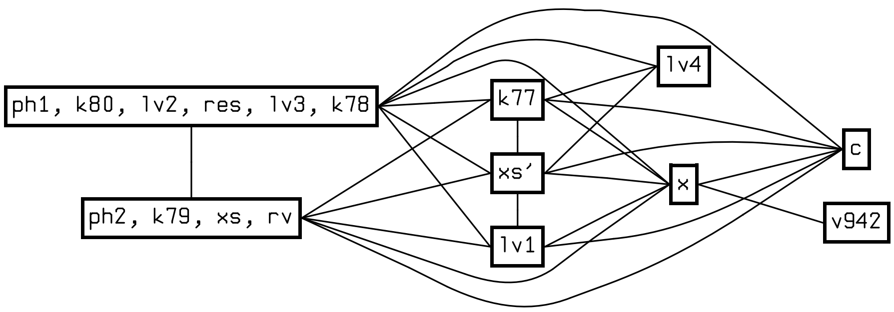

Figure 5.7: All nodes are now popped from the stack and inserted. We have also removed the nodes ph_3, ph_4 and ph_5 as they are only used by the algorithm.

the following colors to the variables:

Variable	**Interfere With**	**Assigned Color**
$\underline{c}$	ph_1, ph_2	ph_4
$\underline{x}$	ph_1, ph_2, ph_4	ph_5
$\underline{k77}$	ph_1, ph_2, ph_4, ph_5	ph_3
$lv1$	ph_1, ph_2, ph_4, ph_5	ph_3
$\underline{xs'}$	$ph_2, ph_2, ph_3, ph_4, ph_5$	*actual spill*
$lv4$	ph_1, ph_3	ph_2
$v942$	ph_5	ph_1

We do not have any callee save registers left for *k77* so *k77* is assigned the caller save register ph_3. The only variable that has to be flushed across function calls is *k77* and the two callee save registers at entry and exit of the function. The resulting interference graph after inserting variables is shown in Figure 5.7.

With the above register assignments and the coalesced nodes we have rewritten the function *fn_xs* to include register allocation information, see Figure 5.8 and 5.9. It is important not to substitute registers for variables yet because we need to compute the live range of each variable when we insert fetch and flush instructions, see Chapter 6.

```
λ^fn_fn_xs {args=[ph2], clos=ph1, res=[ph1]} =>
scope res:ph1 in
 scope xs:ph2, c:ph4 in
  c := ph1;
  xs := ph2;
  scope v942:ph1 in
   (case xs of
     nil => res := #1(c)
    | :: (v942) =>
       scope x:ph5, xs':stack in
        x := #0(v942);
        xs' := #1(v942);
        letregion r22:4 in
         scope k80:ph1 in
          letregion r24:4 in
           scope k77:ph3 in
            letregion r25:2 in
             scope rv:ph2, lv1:ph3, lv2:ph1 in
              rv := [atbot_lf r24, atbot_lf r22] atbot_lf r25;
              lv1 := #2(c);
              lv2 := #3(c);
              ph1 := lv2;
              ph2 := rv;
              ⟨ph1⟩ := foldl_funcall ⟨lv1⟩ ⟨ph2⟩ ⟨⟩ ⟨ph1⟩ ⟨⟩;
              k77 := ph1
             end
            end; (*r25*) end
```

Figure 5.8: Part 1 of *fn_xs* where we have inserted register allocation information but not removed any assignments or simplified it otherwise. This is done by Substitution and Simplify, Chapter 8.

```
          scope k79:ph2 in
           scope lv3:ph1, k78:ph1, lv4:ph2 in
            lv3 := #2(c);
            ph1 := lv3;
            ph2 := x;
            ⟨ph1⟩ := (lv3)fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
            k78 := ph1;
            lv4 := #1(c);
            ph1 := k78;
            ph2 := lv4;
            ⟨ph1⟩ := k78fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
            k79 := ph1
           end;
           ph1 := k77;
           ph2 := k79;
           ⟨ph1⟩ := k77fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
           k80 := ph1
          end
         end
        end; (*r24*)
        ph1 := k80;
        ph2 := xs';
        ⟨ph1⟩ := k80fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
        res := ph1
       end
      end (*r22*)
     end)
  end
 end;
 ph1 := res
end
```

Figure 5.9: Part 2 of *fn_xs* where we have inserted register allocation information but not removed any assignments or simplified it otherwise. This is done by Substition and Simplify, Chapter 8.

Chapter 6

Fetch And Flush

After register allocation (Chapter 5) we have mapped variables to either machine registers or stack locations. If a variable is mapped to a caller save register and the variable is live across a function call then it is necessary to store its value before calling the function and retrieve its value after the call; the caller save register may be overwritten in the called function. We do not have to store variables mapped to callee save registers because they are always preserved by the called function. In this chapter we present a few phases that insert fetch and flush statements such that caller save registers containing values live across an application are preserved.

The set of variables to flush are the variables that fulfills all of the requirements:

- the variable is *locally bound* (i.e., bound with a `scope` construct). In contrast, argument and return values passed on the stack (i.e., variables with stack annotations in the call convention for the function) are not said to be locally bound. Stack slots have been reserved for such variables in the call convention before the function is entered.
- the variable has a live range that crosses an application, that is, the variable is defined before the application and used after the application.
- the variable has not already been flushed (i.e., we only flush variables once in a function).
- the variable is assigned a caller save register. Variables assigned callee save registers are flushed at entry to the function.

We flush callee save registers that are used within the function at entry to the function (i.e., before the registers are assigned another value). We fetch callee save registers at program points where control leaves the function (i.e., at tail calls, raise and returns).

A backward scan on the LineStmt program is used to compute the live variables and then a forward transform inserts flush statements at selected program points.

It is not obvious what program points should be chosen to flush and fetch variables. Højfeld et. al. discuss this in detail [38, Section 6.9]. We simplify the discussion and consider two placement strategies only for flush statements. We can flush a variable before the first application where the variable is live or immediately after the statement that defines the variable. Conditionals complicates the insertion of flush statements before applications because if one branch contains an application and another branch does not, then flushing the variable in only one branch is not safe if the variable is live across another application after the conditional. We choose the simple solution and flush variables right after they are defined. If a variable is defined in a conditional then all branches will contain a flush statement because either a variable is not defined in a branch or it is defined in all branches. This property is guaranteed by the linearization phase, see Section 4.1. Only one flush statement is evaluated at runtime.

We insert a `fetch` v statement after a non tail call if variable v is bound to a caller save register and used before the next call. We could also wait and fetch v just before the use of v but then again conditionals may complicate matters. Also, the register allocater has mapped v to a register for the entire function so fetching earlier does not influence the register pressure. Actually fetching ealier may be good for the performance of the memory subsystem but it is difficult to verify such statements.

6.1 Revised Grammar

We extend LineStmt with the statements

$$\begin{array}{rcl} ls & ::= & \texttt{flush}\ (\mathit{LamVar} \cup \mathit{PhReg}) \\ & | & \texttt{fetch}\ (\mathit{LamVar} \cup \mathit{PhReg}) \end{array}$$

to flush and fetch a variable or a machine register. The register allocator annotates a store type *sty* (Section 5.1.1) on each variable when it is declared. We extend the storage type to include the case when a variable is assigned a register that needs to be flushed.

$$\begin{array}{rcl} \mathit{sty} & ::= & \texttt{stack} \\ & | & \mathit{phreg} \\ & | & \texttt{flushed}\ \mathit{phreg} \end{array}$$

We flush a callee save register if the register is defined in the body of the function. The question is then: where should we flush the register? Consider the code fragment:

```
scope v1:phreg1 in
    ⋮
  scope v2:phreg2 in
    flush phreg2;
      ⋮
    fetch phreg2;
  end;
  scope v3:phreg2 in
    flush phreg2;
      ⋮
    fetch phreg2;
  end;
    ⋮
end
```

where *phreg*$_2$ is a callee save register and v_2 is assigned that register. The variable v_2 does not make use of its slot on the stack and we could therefore flush *phreg*$_2$ in that slot (as shown in the code fragment). However, because the flushed value is lost when the scope for v_2 ends the program fetches the value before the scope ends. This approach leads to an excessive number of flushes and fetches and the advantage of having callee save registers disappears. Instead we wrap a special scope construct around the function body that will set space aside for callee save registers used by the function. The above program then becomes:

```
scope_reg phreg2:flushed phreg2 in
  flush phreg2;
  scope v1:phreg1 in
      ⋮
    scope v2:phreg2 in
        ⋮
    end;
    scope v3:phreg2 in
        ⋮
    end;
      ⋮
  end;
  fetch phreg2
end
```

We then flush the registers at entry to the function and fetch them at exit (i.e., callee save registers are flushed and fetched one time only in each function). The scope construct is added to the language:

$$ls ::= \texttt{scope\_reg}\ phreg : sty\ \texttt{in}\ e\ \texttt{end}$$

We have not computed the actual stack addresses yet (Chapter 7) so the `flush` and `fetch` statements do not contain a source and destination address.

6.2 Set Of Variables To Flush

A backward scan computes the set $F \in \mathcal{P}(LamVar)$ of variables that are live across at least one function call. It also computes the set $R \in \mathcal{P}(PhReg)$ of callee save registers that are used in the function.

The function

$$\mathcal{F}^{\mathcal{T}} : TopDecl_{ls} \rightarrow \mathcal{P}(LamVar) \times \mathcal{P}(PhReg)$$

computes F and R for a top level function. It uses the function

$$\begin{aligned}\mathcal{F} : LineStmt \times \mathcal{P}(LamVar) \times \mathcal{P}(LamVar) \times \mathcal{P}(PhReg) \rightarrow \\ \mathcal{P}(LamVar) \times \mathcal{P}(LamVar) \times \mathcal{P}(PhReg)\end{aligned}$$

on the body of a function.

6.2.1 Top Level Functions

$$\begin{array}{l}
\mathcal{F}^{\mathcal{T}}\ [\![\lambda^{\texttt{fun}}_{lab}\ cc\ \texttt{=>}\ ls]\!]\ = \\
\quad \textbf{let} \\
\quad\quad \textbf{val}\ (_, F, R) = \mathcal{F}\ [\![ls]\!]\ \ \{\}\ \{\}\ \{\} \\
\quad \textbf{in} \\
\quad\quad (\textbf{remove_spilled_args}(cc, F), R \cap callee_save) \\
\quad \textbf{end}
\end{array}$$

The set R contains all machine registers defined in the function so we intersect with the set of callee save registers.

The set F, containing variables live across a function call, returned from $\mathcal{F}$ may contain argument variables passed on the stack. They are removed from F by the function **remove_spilled_args**. They are spilled and therefore always preserved across a function call.

6.2.2 Statements

The set L contains all live variables (and not machine registers). The program returned by the register allocator does not have machine registers live across applications.

$\mathcal{F}$ $[\![$`scope` x: sty `in` ls `end`$]\!]$ $L\ F\ R =$
  let
    val $(L', F', R') = \mathcal{F}\ [\![ls]\!]\ \ L\ F\ R$
    val $R'' =$ **if** $sty = phreg_i$ **then** $R' \cup \{phreg_i\}$ **else** R'
  in
    if $sty =$ `stack` **or** $(sty = phreg_i$ **and** $phreg_i \in callee\_save)$ **then**
      $(L', F' \setminus \{x\}, R'')$
    else
      (L', F', R'')
  end
$\mathcal{F}$ $[\![$`letregion` b `in` ls `end`$]\!]$ $L\ F\ R = \mathcal{F}\ [\![ls]\!]\ \ L\ F\ R$
$\mathcal{F}$ $[\![v$ `:=` $se]\!]$ $L\ F\ R =$
  $((L \setminus \mathbf{def\_var}(v)) \cup \mathbf{use\_var}(se),\ F,$
  $R \cup (\mathbf{get\_reg}(v) \cup \mathbf{get\_reg}(se)))$
$\mathcal{F}$ $[\![$`case` se_1 `of` pat `=>` ls_2 `|` `_` `=>` $ls_3]\!]$ $L\ F\ R =$
  let
    val $(L_2, F_2, R_2) = \mathcal{F}\ [\![ls_2]\!]\ \ L\ F\ R$
    val $(L_3, F_3, R_3) = \mathcal{F}\ [\![ls_3]\!]\ \ L\ F_2\ R_2$
    val $L_1 = L_2 \cup L_3$
  in
    $((L_1 \setminus \mathbf{def\_var}(pat)) \cup \mathbf{use\_var}(se_1),\ F,$
    $R_3 \cup (\mathbf{get\_reg}(se_1) \cup \mathbf{get\_reg}(pat)))$
  end
$\mathcal{F}$ $[\![\langle v_1, \ldots, v_n\rangle$ `:=` $se_{ck}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle]\!]$ $L\ F\ R =$
  let
    val $vars\_to\_flush = L \setminus \{v_1, \ldots, v_n\}$
    val $use = \bigcup_{i \in \{ck, 1, \ldots, n, clos, f_1, \ldots, f_m\}} \mathbf{use\_var}(se_i)$
    val $L' = vars\_to\_flush \cup use$
    val $R' = R \cup (\bigcup_{i \in \{ck, 1, \ldots, n, clos, f_1, \ldots, f_m\}} \mathbf{get\_reg}(se_i)) \cup \mathbf{get\_reg}(\langle v_1, \ldots, v_n\rangle)$
  in
    if $ck \in \{$`funcall`, `fncall`$\}$ **then**
      $(L',\ F \cup vars\_to\_flush,\ R')$
    else
      $(use,\ F,\ R')$
  end
$\mathcal{F}$ $[\![ls_1$ `;` $ls_2]\!]$ $L\ F\ R =$
  let
    val $(L_2, F_2, R_2) = \mathcal{F}\ [\![ls_2]\!]\ \ L\ F\ R$
  in
    $\mathcal{F}\ [\![ls_1]\!]\ \ L_2\ F_2\ R_2$
  end

The functions **def_var** and **use_var** are similar to the **def** and **use** functions from Chapter 5 except that all machine registers are removed. The function

get_reg returns all machine registers in the argument construct. We do not have to flush variables live over a tail call. Before we flush a variable from F we have to check that it is assigned a caller save register. This is done implicitly by removing variables from F at the `scope` construct.

We note that it should not be necessary to update R in the application, case and assignment constructs because if a callee save register is defined in the function then it is mentioned in at least one scope construct. However, we consider the constructs anyway because this holds only as long as argument and result values are passed in caller save registers only.

6.3 Insert Flushes and `scope_reg`

We now insert fetch statements for callee save registers used in the body of a function at program points where the function is left, that is, at return and at tail calls. We also insert the `scope_reg` construct that makes sure that stack locations are reserved for the callee save registers used in the function. Flush statements are inserted for variables assigned caller save registers and live across an application.

We use the function

$$\mathcal{IFF}^{\mathcal{T}} : TopDecl_{ls} \times \mathcal{P}(LamVar) \times \mathcal{P}(PhReg) \rightarrow TopDecl_{ls}$$

on top level functions and

$$\mathcal{IFF} : LineStmt \times \mathcal{P}(LamVar) \times \mathcal{P}(PhReg) \rightarrow LineStmt$$

on function bodies.

6.3.1 Top Level Functions

$$\begin{array}{l}
\mathcal{IFF}^{\mathcal{T}}\ [\![\lambda^{\mathtt{fun}}_{lab}\ cc\ \texttt{=>}\ ls]\!]\ \ F\ R = \\
\quad \lambda^{\mathtt{fun}}_{lab}\ cc\ \texttt{=>} \\
\quad\quad \texttt{scope_reg}\ \textbf{to_reg_seq}(R)\ \texttt{in} \\
\quad\quad\quad \textbf{flush_regs}(R)\,\texttt{;} \\
\quad\quad\quad \mathcal{IFF}\ [\![ls]\!]\ \ F\ R\,\texttt{;} \\
\quad\quad\quad \textbf{fetch_regs}(R) \\
\quad\quad \texttt{end}
\end{array}$$

The set F, that contains all variables that are live across at least one application and assigned caller save registers, does not change during the translation. The set R is passed to $\mathcal{IFF}$ such that callee save registers can be fetched before tail calls. The set R never changes. The function **to_reg_seq** converts the set of machine registers into a sequence (i.e., $\{r_1, r_2\} = r_1 : \texttt{flushed}\ r_1, r_2 : \texttt{flushed}\ r_2$). The function **flush_regs** converts a set of machine registers $\{r_1, r_2\}$ into

$\texttt{flush}\ r_1\texttt{;}$
$\texttt{flush}\ r_2$

and **fetch_regs** converts the set into

$\texttt{fetch}\ r_1\texttt{;}$
$\texttt{fetch}\ r_2$

6.3.2 Statements

The set F contains all locally declared variables assigned caller save registers that are live across at least one function call. The set R contains all callee save registers used in the function.

$\mathcal{IFF}\ [\![\texttt{scope}\ x : sty\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F\ R =$
  if $x \in F$ **then**
    $\texttt{scope}\ x : \texttt{flushed}\ phreg_i\ \texttt{in}\ \mathcal{IFF}\ [\![ls]\!]\ \ F\ R\ \texttt{end}$
  else
    $\texttt{scope}\ x : sty\ \texttt{in}\ \mathcal{IFF}\ [\![ls]\!]\ \ F\ R\ \texttt{end}$
$\mathcal{IFF}\ [\![\texttt{letregion}\ b\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F\ R = \texttt{letregion}\ b\ \texttt{in}\ \mathcal{IFF}\ [\![ls]\!]\ \ F\ R\ \texttt{end}$
$\mathcal{IFF}\ [\![v\ \texttt{:=}\ se]\!]\ \ F\ R =$
  if $v \in F$ **then**
    $v\ \texttt{:=}\ se\texttt{;}$
    $\texttt{flush}\ v$
  else
    $v\ \texttt{:=}\ se$
$\mathcal{IFF}\ [\![\texttt{case}\ se_1\ \texttt{of}\ pat\ \texttt{=>}\ ls_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ ls_3]\!]\ \ F\ R =$
  if $\mathbf{def}(pat) \in F$ **then**
    $\texttt{case}\ se_1\ \texttt{of}$
      $pat\ \texttt{=>}\ \texttt{flush}\ \mathbf{def}(pat)\texttt{;}\ \mathcal{IFF}\ [\![ls_2]\!]\ \ F\ R$
    $\texttt{|}\ \texttt{\_}\ \texttt{=>}\ \texttt{flush}\ \mathbf{def}(pat)\texttt{;}\ \mathcal{IFF}\ [\![ls_3]\!]\ \ F\ R$
  else
    $\texttt{case}\ se_1\ \texttt{of}\ pat\ \texttt{=>}\ \mathcal{IFF}\ [\![ls_2]\!]\ \ F\ R\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ \mathcal{IFF}\ [\![ls_3]\!]\ \ F\ R$
$\mathcal{IFF}\ [\![\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle]\!]\ \ F\ R =$
  if $ck \in \{\texttt{funjmp,fnjmp}\}$ **then**
    let
      val $ck' = \mathbf{mk\_ck}(ck,R)$
    in
      $\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{ck'}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle$
    end
  else
    let
      val $\{v'_1, \ldots, v'_k\} = \{v_i | v_i \in F, i = 1, \ldots, n\}$
    in
      $\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle\texttt{;}$

$$
\begin{array}{l}
\quad\quad \texttt{flush}\ v'_1\,\texttt{;} \\
\quad\quad\quad \vdots \\
\quad\quad \texttt{flush}\ v'_{k-1}\,\texttt{;} \\
\quad\quad \texttt{flush}\ v'_k \\
\quad \textbf{end} \\
\mathcal{IFF}\ [\![ls_1\ \texttt{;}\ ls_2]\!]\ \ F\ R = \mathcal{IFF}\ [\![ls_1]\!]\ \ F\ R\ \texttt{;}\ \mathcal{IFF}\ [\![ls_2]\!]\ \ F\ R
\end{array}
$$

If the pattern is a cons (i.e., $::x$) in the `case` construct then we flush x (if necessary) at entry to the branches.

Consider the rule for application. A result passed on the stack will have a variable in the result bracket and not a machine register. We flush the variable if necessary.

We never insert flushes after a tail call but we insert fetches of callee save registers before the call because a tail call is an exit from the function. However, we must be careful. Is it safe to insert fetch constructs of callee save registers before the application construct? Is it possible that the callee save registers may, before they are fetched, contain values necessary to compile the application? The problem is that code to pass values on the stack has not been generated but is postponed to code generation and the callee save registers may contain values necessary to compile the arguments. However, in Chapter 9 we, of reasons not to mention here, decide that tail calls may not pass arguments on the stack. The remaining code that is generated by the code generator and executed before the call includes fetching the code pointer from a closure in the case of an ordinary function call. If the closure is assigned a callee save register then it is unsafe to redefine the register before fetching the code pointer. We therefore annotate the fetch statements on the call kind and the code generator can then insert the fetch statements after the code pointer has been fetched but before jumping to the called function. The call kind is then defined as

$$
\begin{array}{rcl}
ck & ::= & \texttt{funjmp}\ \mathit{LineStmt} \\
 & | & \texttt{funcall} \\
 & | & \texttt{fnjmp}\ \mathit{LineStmt} \\
 & | & \texttt{fncall}
\end{array}
$$

and the function **mk_ck** as

$$\textbf{mk\_ck}(\texttt{funjmp}, R) = \texttt{funjmp}\ (\textbf{fetch\_regs}\ R)$$
$$\textbf{mk\_ck}(\texttt{fnjmp}, R) = \texttt{fnjmp}\ (\textbf{fetch\_regs}\ R)$$

where **fetch_regs** is defined in Section 6.3.1. Notice, that we, at tail calls, may fetch more callee save registers than actually necessary because a callee save register may not have been defined at the tail call. It is a simple optimization to make sure that, at each tail call, we only fetch callee save registers that have been defined.

6.4 Insert Fetches

Given the set F computed by $\mathcal{F}$ we know what variables are flushed. With a backward scan of the program we compute the set of variables used between function calls and insert fetch statements.

6.4.1 Top Level Functions

The function

$$\mathcal{IF}^{\mathcal{T}} : TopDecl_{ls} \times \mathcal{P}(LamVar) \rightarrow TopDecl_{ls}$$

inserts fetch statements in top level functions.

$\mathcal{IF}^{\mathcal{T}}$ $[\![\lambda^{\texttt{fun}}_{lab}\ cc$ `=>` $ls]\!]$ $F =$
  let
    val $(ls', \_) = \mathcal{IF}\ [\![ls]\!]\ \{\ \}\ F$
  in
    $\lambda^{\texttt{fun}}_{lab}\ cc$ `=>` ls'
  `end`

6.4.2 Statements

The function

$$\mathcal{IF} : LineStmt\ \times \mathcal{P}(LamVar) \times \mathcal{P}(LamVar) \rightarrow LineStmt\ \times \mathcal{P}(LamVar)$$

inserts fetch statements in the body of top level functions. The set U is the set of variables that are used after the current statement and before the next application. Notice, that U is not the same as the set of live variables because we start with the empty set at each function call. The set F is the set of variables live across at least one application and assigned a caller save register.

$\mathcal{IF}$ $[\![$`scope` $x : sty$ `in` ls `end`$]\!]$ $U\ F =$
  let
    val $(ls', U') = \mathcal{IF}\ [\![ls]\!]\ U\ F$
  in
    (`scope` $x : sty$ `in` ls' `end`, U')
  end
$\mathcal{IF}$ $[\![$`letregion` b `in` ls `end`$]\!]$ $U\ F =$
  let
    val $(ls', U') = \mathcal{IF}\ [\![ls]\!]\ U\ F$
  in
    (`letregion` b `in` ls' `end`, U')

$$
\begin{array}{l}
\quad \textbf{end} \\
\mathcal{IF}\ [\![v\ \texttt{:=}\ se]\!]\ \ U\ F = (v\ \texttt{:=}\ se,\ \textbf{use}(se) \cup (U \setminus \{v\})) \\
\mathcal{IF}\ [\![\texttt{case}\ se_1\ \texttt{of}\ pat\ \texttt{=>}\ ls_2\ \texttt{|}\ \_\ \texttt{=>}\ ls_3]\!]\ \ U\ F = \\
\quad \textbf{let} \\
\qquad \textbf{val}\ (ls'_2, U'_2) = \mathcal{IF}\ [\![ls_2]\!]\ \ U\ F \\
\qquad \textbf{val}\ (ls'_3, U'_3) = \mathcal{IF}\ [\![ls_3]\!]\ \ U\ F \\
\quad \textbf{in} \\
\qquad (\texttt{case}\ se_1\ \texttt{of}\ pat\ \texttt{=>}\ ls'_2\ \texttt{|}\ \_\ \texttt{=>}\ ls'_3, \\
\qquad\ ((U'_2 \cup U'_3) \setminus \textbf{def}(pat)) \cup \textbf{use}(se_1)) \\
\quad \textbf{end} \\
\mathcal{IF}\ [\![\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{\{\texttt{fncall},\texttt{funcall}\}}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle]\!]\ \ U\ F = \\
\quad \textbf{let} \\
\qquad \textbf{val}\ \{v'_1, \ldots, v'_k\} = F \cap (U \setminus \{v_1, \ldots, v_n\}) \\
\quad \textbf{in} \\
\qquad (\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle\texttt{;} \\
\qquad\ \texttt{fetch}\ v'_1\texttt{;} \\
\qquad\qquad \vdots \\
\qquad\ \texttt{fetch}\ v'_{k-1}\texttt{;} \\
\qquad\ \texttt{fetch}\ v'_k,\ \bigcup_{i \in \{ck,1,\ldots,n,clos,f_1,\ldots,f_m\}} \textbf{use}(se_i)) \\
\quad \textbf{end} \\
\mathcal{IF}\ [\![ls_1\ \texttt{;}\ ls_2]\!]\ \ U\ F = \\
\quad \textbf{let} \\
\qquad \textbf{val}\ (ls'_2, U'_2) = \mathcal{IF}\ [\![ls_2]\!]\ \ U\ F \\
\qquad \textbf{val}\ (ls'_1, U'_1) = \mathcal{IF}\ [\![ls_1]\!]\ \ U'_2\ F \\
\quad \textbf{in} \\
\qquad (ls'_1\ \texttt{;}\ ls'_2,\ U'_1) \\
\quad \textbf{end}
\end{array}
$$

Notice, that we never have the case that a variable is fetched before it is flushed. For instance, consider the code fragment:

$$
\begin{array}{l}
\quad \vdots \\
\langle v_0\rangle\ \texttt{:=}\ f\ \langle x\rangle\ \langle c\rangle\ \langle\rangle\texttt{;} \\
\texttt{fetch}\ v_0\texttt{;} \\
\texttt{flush}\ v_0\texttt{;} \\
\quad \vdots
\end{array}
$$

Assume the function $\mathcal{IFF}$ inserts the flush statement because v_0 is defined at the call site and $v_0 \in F$. However, the function $\mathcal{IF}$ does not insert the fetch because v_0 is not live at entrance to the function call.

6.5 Example

The function *fn_xs* from Figure 5.8 and 5.9 on page 121 and 122 is shown in Figure 6.1 and 6.2 where we have inserted `flush`, `fetch` and `scope_reg` statements.

We have used the following caller and callee save sets: $caller\_save = \{ph_1, ph_2, ph_3\}$ and $callee\_save = \{ph_4, ph_5\}$.

The variables $\{c, x, k77, xs'\}$ are live across function calls but only $k77$ is assigned a caller save register: ph_3. The variables c and x are assigned the callee save registers ph_4 and ph_5 respectively. The variable xs' is spilled. The function $\mathcal{F}$ then returns the set $F = \{k77\}$. The set R of used callee save registers returned by $\mathcal{F}$ is $\{ph_4, ph_5\}$.

```
λ^fn_fn_xs {args=[ph2], clos=ph1, res=[ph1]} =>
scope_reg ph4 : flushed ph4, ph5 : flushed ph5 in
flush ph4;
flush ph5;
scope res:ph1 in
 scope xs:ph2, c:ph4 in
  c := ph1;
  xs := ph2;
  scope v942:ph1 in
   (case xs of
     nil => res := #1(c)
    |:: (v942) =>
      scope x:ph5, xs':stack in
       x := #0(v942);
       xs' := #1(v942);
       letregion r22:4 in
        scope k80:ph1 in
         letregion r24:4 in
          scope k77:flushed ph3 in
           letregion r25:2 in
            scope rv:ph2, lv1:ph3, lv2:ph1 in
             rv := [atbot_lf r24, atbot_lf r22] atbot_lf r25;
             lv1 := #2(c);
             lv2 := #3(c);
             ph1 := lv2;
             ph2 := rv;
             ⟨ph1⟩ := foldl_funcall ⟨lv1⟩ ⟨ph2⟩ ⟨⟩ ⟨ph1⟩ ⟨⟩;
             k77 := ph1;
             flush k77
            end
           end; (*r25*)
```

Figure 6.1: Part 1 of function *fn_xs* after we have inserted explicit `scope_reg`, `fetch` and `flush` statements. We have used *caller_save* = $\{ph_1, ph_2, ph_3\}$ and *callee_save* = $\{ph_4, ph_5\}$.

```
            scope k79:ph₂ in
             scope lv₃:ph₁, k78:ph₁, lv₄:ph₂ in
              lv₃ := #2(c);
              ph₁ := lv₃;
              ph₂ := x;
              ⟨ph₁⟩ := (lv₃)fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
              k78 := ph₁;
              lv₄ := #1(c);
              ph₁ := k78;
              ph₂ := lv₄;
              ⟨ph₁⟩ := k78fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
              fetch k77;
              k79 := ph₁
             end;
             ph₁ := k77;
             ph₂ := k79;
             ⟨ph1⟩ := k77fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
             k80 := ph₁
            end
           end
          end; (*r24*)
          ph₁ := k80;
          ph₂ := xs′;
          ⟨ph1⟩ := k80fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
          res := ph₁
         end
        end (*r22*)
       end)
   end
  end;
  ph₁ := res
end;
fetch ph₄;
fetch ph₅
end
```

Figure 6.2: Part 2 of function *fn_xs* after we have inserted explicit `scope_reg`, `fetch` and `flush` statements. We have used *caller_save* = $\{ph_1, ph_2, ph_3\}$ and *callee_save* = $\{ph_4, ph_5\}$.

Chapter 7

Calculate Offsets

All variables and machine registers in the LineStmt program are now annotated a storage type that uniquely specifies how it should be stored, either on the stack, in a machine register or both. In the calculate offsets phase we assign a slot in the activation record for the function (i.e., on the stack) to every variable or machine register that needs it.

We also assign one or more slots for regions. Finite regions are allocated directly in the activation record. Infinite regions are implemented with an infinite region descriptor (Chapter 2) that is allocated in the activation record. More specifically we

- add offsets to the storage type *sty* annotated on `scope` and `scope_reg` statements. The storage types `stack` and `flushed` require a slot in the activation record.
- add a storage type *sty* to the `letregion` construct such that region variables can be assigned one or more slots in the activation record.
- add offsets to the `fetch` and `flush` statements.

7.1 Revised Storage Type

The storage type is extended with an offset annotation for `stack` and `flushed`:

$$\begin{array}{rcl} sty & ::= & \texttt{stack}(o) \\ & | & phreg \\ & | & \texttt{flushed}(phreg, o) \end{array}$$

where $o \in \mathit{Offset}$. The `fetch` and `flush` statements also include an offset:

$$\begin{array}{rcl} e & ::= & \texttt{flush}(xv, o) \\ & | & \texttt{fetch}(xv, o) \\ & | & \texttt{letregion}\ b : sty\ \texttt{in}\ e\ \texttt{end} \end{array}$$

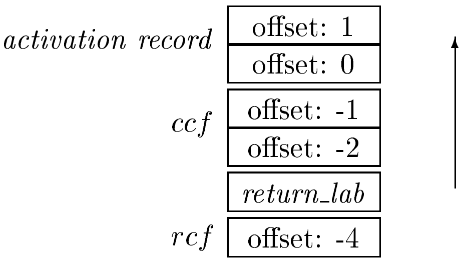

Figure 7.1: An example stack containing a call convention frame (ccf), return convention frame (rcf), return label and an activation record. A call convention contains ccf, rcf and the return label. The first slot in the activation record has offset 0 and the offsets in the call convention are negative. The stack grows upwards.

where xv is a variable or a machine register. The storage type on `letregion` constructs are always on the form $\texttt{stack}(o)$, for some offset o.

7.2 Algorithm $\mathcal{CO}$

We do a single forward scan on the LineStmt program. We use the environment

$$F : (\mathit{LamVar} \cup \mathit{PhReg}) \rightarrow \mathit{Offset}$$

to map flushed variables or machine registers into their offset address in the activation record.

7.2.1 Top Level Functions

The function

$$\mathcal{CO}^{\mathcal{T}} : \mathit{TopDecl}_{ls} \rightarrow \mathit{TopDecl}_{ls}$$

is used on top level functions

$$\begin{array}{l} \mathcal{CO}^{\mathcal{T}} \ [\![\lambda_{lab}^{\texttt{fun}}\ cc \ \texttt{=>}\ ls]\!] \ = \\ \quad \lambda_{lab}^{\texttt{fun}(f\_size)}\ cc \ \texttt{=>}\ \mathcal{CO}\ [\![\mathrm{ls}]\!] \ \ \textbf{init\_F}(cc)\ 0 \end{array}$$

Ordinary functions are done similarly. The f_size annotation on top level functions denote the size of the function frame. It is calculated as the maximum offset used in the body of the function and calculated as a side effect in $\mathcal{CO}$ (though, not shown in the algorithm).

The function **init_F** updates F with offsets to parameters passed on the stack. Consider Figure 7.1 where we have a call convention and an activation record allocated on a stack. Offset 0 is the first slot in the activation record (i.e., the slot at the bottom of the activation record). Offsets to parameters in the call convention are negative and offsets to locally defined variables are positive. Given a call convention *cc* the function **init_F** can assign offsets to the stack allocated parameters because the layout of the call convention on the stack is uniquely defined in the same way that the function **resolve_cc** in Section 5.1.3 uniquely assigns machine registers to parameters.

7.2.2 Statements

The function

$$\mathcal{CO} : \mathit{LineStmt} \times ((\mathit{LamVar} \cup \mathit{PhReg}) \to \mathit{Offset}) \times \mathit{Offset} \to \mathit{LineStmt}$$

is used on function bodies.

$\mathcal{CO}\ [\![\texttt{scope}\ x : \texttt{stack in}\ ls\ \texttt{end}]\!]\ \ F\ o =$
  $\texttt{scope}\ x : \texttt{stack}(o)\ \texttt{in}\ \mathcal{CO}\ [\![ls]\!]\ \ (F + \{x \mapsto o\})\ (o+1)\ \texttt{end}$
$\mathcal{CO}\ [\![\texttt{scope}\ x : \mathit{phreg}_i\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F\ o =$
  $\texttt{scope}\ x : \mathit{phreg}_i\ \texttt{in}\ \mathcal{CO}\ [\![ls]\!]\ \ F\ o\ \texttt{end}$
$\mathcal{CO}\ [\![\texttt{scope}\ x : \texttt{flushed}\ \mathit{phreg}_i\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F\ o =$
  $\texttt{scope}\ x : \texttt{flushed}(\mathit{phreg}_i, o)\ \texttt{in}\ \mathcal{CO}\ [\![ls]\!]\ \ (F + \{x \mapsto o\})\ (o+1)\ \texttt{end}$
$\mathcal{CO}\ [\![\texttt{scope\_reg}\ \mathit{phreg}_i : \texttt{flushed}\ \mathit{phreg}_i\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F\ o =$
  $\texttt{scope\_reg}\ \mathit{phreg}_i : \texttt{flushed}(\mathit{phreg}_i, o)\ \texttt{in}\ \mathcal{CO}\ [\![ls]\!]\ \ (F + \{\mathit{phreg}_i \mapsto o\})\ (o+1)\ \texttt{end}$
$\mathcal{CO}\ [\![\texttt{letregion}\ \rho : \infty\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F\ o =$
  $\texttt{letregion}\ \rho : \infty : \texttt{stack}(o)\ \texttt{in}\ \mathcal{CO}\ [\![ls]\!]\ \ F\ (o + \mathit{size\_reg\_desc})\ \texttt{end}$
$\mathcal{CO}\ [\![\texttt{letregion}\ \rho : n\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F\ o =$
  $\texttt{letregion}\ \rho : n : \texttt{stack}(o)\ \texttt{in}\ \mathcal{CO}\ [\![ls]\!]\ \ F\ (o+n)\ \texttt{end}$
$\mathcal{CO}\ [\![v\ \texttt{:=}\ se]\!]\ \ F\ o = v\ \texttt{:=}\ se$
$\mathcal{CO}\ [\![\texttt{case}\ se_1\ \texttt{of}\ \mathit{pat}\ \texttt{=>}\ ls_2\ \texttt{|}\ \_\ \texttt{=>}\ ls_3]\!]\ \ F\ o =$
  $\texttt{case}\ se_1\ \texttt{of}\ \mathit{pat}\ \texttt{=>}\ \mathcal{CO}\ [\![ls_2]\!]\ \ F\ o\ \texttt{|}\ \_\ \texttt{=>}\ \mathcal{CO}\ [\![ls_3]\!]\ \ F\ o$
$\mathcal{CO}\ [\![\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle]\!]\ \ F\ o =$
  $\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle$
$\mathcal{CO}\ [\![ls_1\ \texttt{;}\ ls_2]\!]\ \ F\ o = \mathcal{CO}\ [\![ls_1]\!]\ \ F\ o\ \texttt{;}\ \mathcal{CO}\ [\![ls_2]\!]\ \ F\ o$
$\mathcal{CO}\ [\![\texttt{flush}\ xv]\!]\ \ F\ o = \texttt{flush}(xv,\ F(xv))$
$\mathcal{CO}\ [\![\texttt{fetch}\ xv]\!]\ \ F\ o = \texttt{fetch}(xv,\ F(xv))$

The constant *size_reg_desc* denotes the size of an infinite region descriptor.

7.3 Example

The function *fn_xs* from Figure 6.1 and 6.2 on page 134 and 135 is shown in Figure 7.2 and 7.3 with offsets inserted.

$\lambda^{\texttt{fn}(14)}_{\textit{fn\_xs}}$ {args=[ph_2], clos=ph_1, res=[ph_1]} `=>`
`scope_reg` ph_4 : `flushed`$(ph_4, 0)$, ph_5 : `flushed`$(ph_5, 1)$ `in`
`flush` (ph_4,0);
`flush` (ph_5,1);
`scope` res:ph_1 `in`
 `scope` xs:ph_2, c:ph_4 `in`
  c `:=` ph_1;
  xs `:=` ph_2;
  `scope` $v942$:ph_1 `in`
   (`case` xs `of`
     `nil =>` res `:= #1`(c)
    `|::` ($v942$) `=>`
      `scope` x:ph_5, xs':`stack`(2) `in`
       x `:= #0`($v942$);
       xs' `:= #1`($v942$);
       `letregion` $r22$:4:`stack`(3) `in`
        `scope` $k80$:ph_1 `in`
         `letregion` $r24$:4:`stack`(7) `in`
          `scope` $k77$:`flushed`$(ph_3, 11)$ `in`
           `letregion` $r25$:2:`stack`(12) `in`
            `scope` rv:ph_2, lv_1:ph_3, lv_2:ph_1 `in`
             rv `:=` [`atbot_lf` $r24$, `atbot_lf` $r22$] `atbot_lf` $r25$;
             lv_1 `:= #2`(c);
             lv_2 `:= #3`(c);
             ph_1 `:=` lv_2;
             ph_2 `:=` rv;
             $\langle ph_1\rangle$ `:=` $foldl_{\texttt{funcall}}$ $\langle lv_1\rangle$ $\langle ph_2\rangle$ $\langle\rangle$ $\langle ph_1\rangle$ $\langle\rangle$;
             $k77$ `:=` ph_1;
             `flush` ($k77$,11)
            `end`
           `end; (*r25*)`

Figure 7.2: Part 1 of the function *fn_xs* where we have added offset annotations on the storage type, `fetch` and `flush` statements. The size of the activation record is 14; a finite region ($r25$) of size two is stored at offset 12 and we start at offset 0.

```
          scope k79:ph₂ in
           scope lv₃:ph₁, k78:ph₁, lv₄:ph₂ in
            lv₃ := #2(c);
            ph₁ := lv₃;
            ph₂ := x;
            ⟨ph₁⟩ := (lv₃)fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
            k78 := ph₁;
            lv₄ := #1(c);
            ph₁ := k78;
            ph₂ := lv₄;
            ⟨ph₁⟩ := k78fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
            fetch (k77,11);
            k79 := ph₁
           end;
           ph₁ := k77;
           ph₂ := k79;
           ⟨ph1⟩ := k77fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
           k80 := ph₁
          end
         end
        end; (*r24*)
        ph₁ := k80;
        ph₂ := xs';
        ⟨ph1⟩ := k80fncall ⟨ph₂⟩ ⟨ph₁⟩ ⟨⟩;
        res := ph₁
       end
      end (*r22*)
     end)
  end
 end;
 ph₁ := res
end;
fetch (ph₄,0);
fetch (ph₅,1)
end
```

Figure 7.3: Part 2 of the function *fn_xs* where we have added offset annotations on the storage type, `fetch` and `flush` statements.

Chapter 8

Substitution and Simplify

The register allocator inserts register mapping information at `scope` constructs but does not replace variables with registers throughout the code. Substitution and simplify is the last phase before code generation and replaces variables with register allocation information. Unnecessary copy statements are removed; coalescing two nodes may produce an unnecessary copy statement.

Substitution and simplify produces a term where at every use or definition, we either have a label, a constant, a machine register or one of the following annotations:

- spilled variables are annotated with `stack`(o), where o is the offset in the activation record where the variable is stored.
- locally declared infinite regions are annotated with `reg_i`(o) where o is the offset in the activation record where the infinite region descriptor is stored.
- locally declared finite regions are annotated with `reg_f`(o), where o is the offset in the activation record where the content of the region is stored.

After substitution and simplify we have a simplified term on which we believe code generation (Chapter 9) gets as simple as possible. For instance, it is possible to perform code generation with no use of environments. The code generator performs a single pass over the term and unfolds each LineStmt construct to machine code.

It is not necessary to bind a local declared region to a variable or register because the region is boxed (i.e., denote a memory area) and the address of the memory area is known at compile time. However, passing a region to a region polymorphic function does happen in either a machine register or in a spilled variable on the stack.

8.1 Revised Grammar

Now that we remove statements we may end up with all statements removed (likely not to happen, though). We add the statement `none` to denote that.

The full grammar for LineStmt after substitution and simplify is shown in Figure 8.1. The semantic objects are shown in Figure 8.2. We have added an *access type* that denotes how a value is obtained (e.g., from a machine register or from the stack).

The constructs `scope`, `scope_reg` and `letregion` on finite regions are not needed after substitution and simplify because variables are replaced with an access type. We can either keep the constructs anyway for the purpose of information (i.e., how variables are mapped to stack offsets and machine registers) or remove them from the language. We remove them in the presentation but the implementation keeps them. However, the pretty printer in the implementation may choose not to print the constructs. We may not remove the `letregion` construct for infinite regions because the dynamic semantics for an infinite region is to store an infinite region descriptor in the activation record. We therefore introduce the construct `letregioni` o `in` ls `end` in the language, where o is the offset in the activation record where the infinite region descriptor is stored. We can also remove the storage type from the language now that the `scope`, `scope_reg` and `letregion` constructs are removed.

8.2 Algorithm $\mathcal{SS}$

We do a single forward scan on the LineStmt program and records the access type for each variable in an environment F.

$$F : Var \rightarrow Aty$$

The environment maps variables and region variables into an access type denoted by the set Aty ($aty \in Aty$). Unnecessary copy statements are removed by comparing the access type of the variables in the copy statement and if they are equal then the copy statement is removed.

8.2.1 Top Level Functions

The function

$$\mathcal{SS}^{\mathcal{T}} : TopDecl_{ls} \rightarrow TopDecl_{ls}$$

is used on top level functions

$$\mathcal{SS}^{\mathcal{T}}\ [\![\lambda_{lab}^{\texttt{fun}(f\_size)}\ cc \texttt{ => } ls]\!]\ = \lambda_{lab}^{\texttt{fun}(f\_size)}\ cc \texttt{ => } \mathcal{SS}\ [\![\mathrm{ls}]\!]\ \{\ \}$$

Ordinary functions are done similarly.

$$
\begin{array}{rcl}
top\_decl & ::= & \lambda_{lab}^{\texttt{fun}(f\_size)}\ cc\ \texttt{=>}\ ls \\
& | & \lambda_{lab}^{\texttt{fn}(f\_size)}\ cc\ \texttt{=>}\ ls \\
& | & top\_decl_1\ \texttt{;}\ top\_decl_2 \\
\\
aty & ::= & \texttt{stack}(o) \\
& | & phreg \\
& | & \texttt{reg\_i}(o) \\
& | & \texttt{reg\_f}(o) \\
\\
ck & ::= & \texttt{funjmp}\ LineStmt \mid \texttt{funcall} \mid \texttt{fnjmp}\ LineStmt \mid \texttt{fncall} \\
sma & ::= & \texttt{attop} \mid \texttt{atbot} \mid \texttt{sat} \\
a & ::= & sma\ aty \\
m & ::= & n \mid \infty \\
b & ::= & \rho : m \\
\\
c & ::= & i \mid \texttt{nil} \mid lab \\
pat & ::= & c \mid \texttt{::}\ aty \\
bop & ::= & \texttt{+},\ \texttt{-},\ \texttt{<},\ \ldots \\
\\
be & ::= & (se_1, \ldots, se_n) \\
& | & \lambda^{lab}\ [se_{f_1}, \ldots, se_{f_n}] \\
& | & [a_1, \ldots, a_n]_{regvec} \\
& | & [se_{f_1}, \ldots, se_{f_n}]_{sclos} \\
\\
se & ::= & be\ a \\
& | & aty \\
& | & c \\
& | & \texttt{::}\ aty \\
& | & se_1\ bop\ se_2 \\
& | & \texttt{\#}n(aty) \\
\\
ls & ::= & \texttt{letregioni}\ o\ \texttt{in}\ ls\ \texttt{end} \\
& | & aty\ \texttt{:=}\ se \\
& | & \texttt{case}\ se_1\ \texttt{of}\ pat\ \texttt{=>}\ ls_2\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ ls_3 \\
& | & \langle aty_1, \ldots, aty_n \rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n \rangle\ \langle se_{clos} \rangle\ \langle se_{f_1}, \ldots, se_{f_m} \rangle \\
& | & \langle aty_1, \ldots, aty_n \rangle\ \texttt{:=}\ lab_{ck}\ \langle se_1, \ldots, se_n \rangle\ \langle se_{reg} \rangle\ \langle a_1, \ldots, a_l \rangle \\
& | & \qquad\qquad\qquad\qquad \langle se_{clos} \rangle\ \langle se_{f_1}, \ldots, se_{f_m} \rangle \\
& | & ls_1\ \texttt{;}\ ls_2 \\
& | & \texttt{flush}(aty, o) \\
& | & \texttt{fetch}(aty, o) \\
& | & \texttt{none}
\end{array}
$$

Figure 8.1: The grammar for LineStmt after substitution and simplify. We have removed the constructs: `letregion`, `scope`, `scope_reg` and the storage type. We have introduced the `letregioni` construct where o is the offset in the activation record where the infinite region descriptor is stored.

$$\begin{array}{rcl} lab & \in & Label \\ i & \in & Int \\ phreg & \in & PhReg \\ o & \in & Offset \\ cc & \in & CC \end{array}$$

Figure 8.2: The semantic objects used in LineStmt. We let *CC* denote the set of call conventions. The set *Offset* denote offsets in an activation record. The set *PhReg* contains the machine registers.

8.2.2 Statements

The function

$$\mathcal{SS} : LineStmt \times (Var \to Aty) \to LineStmt$$

is used on function bodies.

$$\begin{array}{l}
\mathcal{SS}\ [\![\texttt{scope}\ x : \texttt{stack}(o)\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F = \mathcal{SS}\ [\![ls]\!]\ \ (F + \{x \mapsto \texttt{stack}(o)\}) \\
\mathcal{SS}\ [\![\texttt{scope}\ x : \texttt{flushed}(phreg)\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F = \mathcal{SS}\ [\![ls]\!]\ \ (F + \{x \mapsto phreg\}) \\
\mathcal{SS}\ [\![\texttt{scope}\ x : phreg\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F = \mathcal{SS}\ [\![ls]\!]\ \ (F + \{x \mapsto phreg\}) \\
\mathcal{SS}\ [\![\texttt{scope_reg}\ phreg : \texttt{flushed}(phreg, o)\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F = \mathcal{SS}\ [\![ls]\!]\ \ F \\
\mathcal{SS}\ [\![\texttt{letregion}\ \rho : \infty : \texttt{stack}(o)\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F = \\
\quad \texttt{letregioni}\ o\ \texttt{in}\ \mathcal{SS}\ [\![ls]\!]\ \ (F + \{\rho \mapsto \texttt{reg_i}(o)\})\ \texttt{end} \\
\mathcal{SS}\ [\![\texttt{letregion}\ \rho : n : \texttt{stack}(o)\ \texttt{in}\ ls\ \texttt{end}]\!]\ \ F = \mathcal{SS}\ [\![ls]\!]\ \ (F + \{\rho \mapsto \texttt{reg_f}\}(o)) \\
\mathcal{SS}\ [\![v\ \texttt{:=}\ se]\!]\ \ F = \\
\quad \textbf{if}\ F(v) = (\mathcal{SS}^{se}\ [\![se]\!]\ \ F)\ \textbf{then} \\
\quad\quad \texttt{none} \\
\quad \textbf{else} \\
\quad\quad F(v)\ \texttt{:=}\ \mathcal{SS}^{se}\ [\![se]\!]\ \ F \\
\mathcal{SS}\ [\![\texttt{case}\ se_1\ \texttt{of}\ pat\ \texttt{=>}\ ls_2\ \texttt{|}\ _\ \texttt{=>}\ ls_3]\!]\ \ F = \\
\quad \texttt{case}\ \mathcal{SS}^{se}\ [\![se_1]\!]\ \ F\ \texttt{of}\ \mathcal{SS}^{pat}\ [\![pat]\!]\ \ F\ \texttt{=>}\ \mathcal{SS}\ [\![ls_2]\!]\ \ F\ \texttt{|}\ _\ \texttt{=>}\ \mathcal{SS}\ [\![ls_3]\!]\ \ F \\
\mathcal{SS}\ [\![\langle v_1, \ldots, v_n\rangle\ \texttt{:=}\ se_{ck}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle]\!]\ \ F = \\
\quad \langle F(v_1), \ldots, F(v_n)\rangle\ \texttt{:=}\ \mathcal{SS}^{se}\ [\![se_{(\mathcal{SS}^{ck}\ [\![ck]\!]\ F)}]\!]\ \ F \\
\quad\quad\quad\quad\quad\quad\quad\quad \langle \mathcal{SS}^{se}\ [\![se_1]\!]\ F, \ldots, \mathcal{SS}^{se}\ [\![se_n]\!]\ F\rangle \\
\quad\quad\quad\quad\quad\quad\quad\quad \langle \mathcal{SS}^{se}\ [\![se_{clos}]\!]\ F\rangle \\
\quad\quad\quad\quad\quad\quad\quad\quad \langle \mathcal{SS}^{se}\ [\![se_{f_1}]\!]\ F, \ldots, \mathcal{SS}^{se}\ [\![se_{f_m}]\!]\ F\rangle \\
\mathcal{SS}\ [\![ls_1\ \texttt{;}\ ls_2]\!]\ \ F = \\
\quad \textbf{case}\ (\mathcal{SS}\ [\![ls_1]\!]\ \ F, \mathcal{SS}\ [\![ls_2]\!]\ \ F)\ \textbf{of} \\
\quad\quad (\textbf{none},\textbf{none}) => \textbf{none} \\
\quad |\ \ (ls'_1,\textbf{none}) => ls'_1 \\
\quad |\ \ (\textbf{none},ls'_2) => ls'_2 \\
\quad |\ \ (ls'_1,ls'_2) => ls'_1\ \texttt{;}\ ls'_2 \\
\mathcal{SS}\ [\![\texttt{flush}(xv,o)]\!]\ \ F = \texttt{flush}(F(xv), o) \\
\mathcal{SS}\ [\![\texttt{fetch}(xv,o)]\!]\ \ F = \texttt{fetch}(F(xv), o)
\end{array}$$

The algorithm $\mathcal{SS}$ uses some derived functions for simple expressions, boxed expressions, allocation directives, call kinds and patterns.

$$
\begin{aligned}
&\mathcal{SS}^{ck} : \mathit{CallKind} \rightarrow (\mathit{Var} \rightarrow \mathit{Aty}) \rightarrow \mathit{CallKind}\\
&\mathcal{SS}^{ck}\ [\![\texttt{funcall}]\!]\ \ F = \texttt{funcall}\\
&\mathcal{SS}^{ck}\ [\![\texttt{fncall}]\!]\ \ F = \texttt{fncall}\\
&\mathcal{SS}^{ck}\ [\![\texttt{funjmp}\ ls]\!]\ \ F = \texttt{funjmp}\ (\mathcal{SS}\ [\![ls]\!]\ \ F)\\
&\mathcal{SS}^{ck}\ [\![\texttt{fnjmp}\ ls]\!]\ \ F = \texttt{fnjmp}\ (\mathcal{SS}\ [\![ls]\!]\ \ F)
\end{aligned}
$$

where $\mathit{CallKind}$ is the set of call kinds ranged over by ck. Tail calls may contain fetch statements for callee save registers in the call kind, see Chapter 6.

$$
\begin{aligned}
&\mathcal{SS}^{pat} : \mathit{Pat} \rightarrow (\mathit{Var} \rightarrow \mathit{Aty}) \rightarrow \mathit{Pat}\\
&\mathcal{SS}^{pat}\ [\![c]\!]\ \ F = c\\
&\mathcal{SS}^{pat}\ [\![::\ x]\!]\ \ F = F(x)
\end{aligned}
$$

where Pat is the set of patterns ranged over by pat.

$$
\begin{aligned}
&\mathcal{SS}^{be} : \mathit{BoxedExp} \rightarrow (\mathit{Var} \rightarrow \mathit{Aty}) \rightarrow \mathit{BoxedExp}\\
&\mathcal{SS}^{be}\ [\![(se_1, \ldots, se_n)]\!]\ \ F = (\mathcal{SS}^{se}\ [\![se_1]\!]\ F, \ldots, \mathcal{SS}^{se}\ [\![se_1]\!]\ F)\\
&\mathcal{SS}^{be}\ [\![\lambda^{lab}\ [se_{f_1}, \ldots, se_{f_n}]]\!]\ \ F = \lambda^{lab}\ [\mathcal{SS}^{se}\ [\![se_{f_1}]\!]\ F, \ldots, \mathcal{SS}^{se}\ [\![se_{f_n}]\!]\ F]\\
&\mathcal{SS}^{be}\ [\![[a_1, \ldots, a_n]_{regvec}]\!]\ \ F = [\mathcal{SS}^{sma}\ [\![a_1]\!]\ F, \ldots, \mathcal{SS}^{sma}\ [\![a_n]\!]\ ]_{regvec}\\
&\mathcal{SS}^{be}\ [\![[se_{f_1}, \ldots, se_{f_n}]_{sclos}]\!]\ \ F = [\mathcal{SS}^{se}\ [\![se_{f_1}]\!]\ , \ldots, \mathcal{SS}^{se}\ [\![se_{f_n}]\!]\ ]_{sclos}
\end{aligned}
$$

where $\mathit{BoxedExp}$ is the set of boxed expressions ranged over by be.

$$
\begin{aligned}
&\mathcal{SS}^{sma} : \mathit{Alloc} \rightarrow (\mathit{Var} \rightarrow \mathit{Aty}) \rightarrow \mathit{Alloc}\\
&\mathcal{SS}^{sma}\ [\![sma\ xv]\!]\ \ F = sma\ F(xv)
\end{aligned}
$$

where Alloc is the set of allocation directives ranged over by a.

$$
\begin{aligned}
&\mathcal{SS}^{se} : \mathit{SimpleExp} \rightarrow (\mathit{Var} \rightarrow \mathit{Aty}) \rightarrow \mathit{SimpleExp}\\
&\mathcal{SS}^{se}\ [\![be\ a]\!]\ \ F = (\mathcal{SS}^{be}\ [\![be]\!]\ F)\ (\mathcal{SS}^{sma}\ [\![a]\!]\ F)\\
&\mathcal{SS}^{se}\ [\![x]\!]\ \ F = F(x)\\
&\mathcal{SS}^{se}\ [\![::\ se]\!]\ \ F = ::\ (\mathcal{SS}^{se}\ [\![se]\!]\ F)\\
&\mathcal{SS}^{se}\ [\![se_1\ bop\ se_2]\!]\ \ F = (\mathcal{SS}^{se}\ [\![se_1]\!]\ F)\ bop\ (\mathcal{SS}^{se}\ [\![se_2]\!]\ F)\\
&\mathcal{SS}^{se}\ [\![\texttt{\#}n(se)]\!]\ \ F = \texttt{\#}n(\mathcal{SS}^{se}\ [\![se]\!]\ F)
\end{aligned}
$$

where $\mathit{SimpleExp}$ is the set of simple expressions ranged over by se. The environment F has lambda variables and region variables as domain. In the case that we lookup a machine register (i.e., $F(ph)$) then we return ph.

8.3 Example

The function *fn_xs* from Figure 7.2 and 7.3 on page 139 and 140 is shown in Figure 8.3 after substitution and simplification.

```
λ^{fn(14)}_{fn_xs} {args=[ph2], clos=ph1, res=[ph1]} =>
flush (ph4,0);
flush (ph5,1);
ph4 := ph1;
(case ph2 of
   nil => ph1 := #1(ph4)
 | :: (ph1) =>
   ph5 := #0(ph1);
   stack(2) := #1(ph1);
   ph2 := [atbot_lf reg_f(7), atbot_lf reg_f(3)] atbot_lf reg_f(12);
   ph3 := #2(ph4);
   ph1 := #3(ph4);
   ⟨ph1⟩ := foldl_funcall ⟨ph3⟩ ⟨ph2⟩ ⟨⟩ ⟨ph1⟩ ⟨⟩;
   ph3 := ph1;
   flush (ph3,11);
   ph1 := #2(ph4);
   ph2 := ph5;
   ⟨ph1⟩ := (ph1)_fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
   ph2 := #1(ph4);
   ⟨ph1⟩ := (ph1)_fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
   fetch (ph3,11);
   ph2 := ph1;
   ph1 := ph3;
   ⟨ph1⟩ := (ph3)_fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩;
   ph2 := stack(2);
   ⟨ph1⟩ := (ph1)_fncall ⟨ph2⟩ ⟨ph1⟩ ⟨⟩);
fetch (ph4,0);
fetch (ph5,1)
```

Figure 8.3: We have simplified the function *fn_xs*. The code generator works on this code.

Chapter 9

Code generation

The simplified LineStmt program from Chapter 8 is in a form that makes code generation easy; every construct in the program can be compiled by looking at the construct only, that is, no environments are needed and the code generator can be viewed as a sophisticated macro expander.

We use the Kit Abstract Machine (KAM) as the target machine, which is a low level three address code machine extended with region allocation primitives. The KAM machine is invented for the purpose of presentation only and the code generator in the ML Kit compiles into the target machine directly (e.g., HP PA-RISC). We believe it is better to compile directly to the target machine (e.g., HP PA-RISC) than using another intermediate language before the tharget machine because it is difficult to define an intermediate language that is more low level than LineStmt and still incorporates the differences in the architectures. For instance, a X86 based machine does not require all operands of an instruction to reside in machine registers which is normally the case for a RISC based machine. The code generator can then compile the access type `stack` differenctly on the two architectures. A load from store into a machine register is necessary for a RISC machine but probably not for the X86 machine.

The grammar for KAM is shown in Section 9.1. The algorithm $\mathcal{CG}$ that compiles the LineStmt program into KAM is shown in Section 9.2. We discuss how allocation points with multiplicities and storage modes are compiled, Section 9.2.2 and 9.2.3. Functions and applications are discussed in Section 9.2.5 and 9.2.6. The running example program compiled into KAM is shown in Section 9.3.

9.1 Kit Abstract Machine

The Kit Abstract Machine (KAM) resembles a von Neumann Machine with mostly three address statements. The machine has a number of general 32 bit machine registers *PhReg* ranged over by *ph*.

We have a runtime stack with a stack pointer $sp \in PhReg$. We may push, pop and update the stack as an array (i.e., $sp[o]$ denotes the cell on the stack with offset $o \in Offset$ from sp). The stack grows with increasing addresses and sp always points at the first free cell on the stack. The code generator uses the convention that the stack pointer always points at the first address after the function frame for the current function, and the stack pointer is updated at application and return points only.

The KAM does not have a regular heap but a *region heap* with an infinite number of fixed sized region pages. Finite regions are stored on the stack and infinite regions have an infinite region descriptor stored on the stack with a pointer to a list of region pages. Region pages in the region heap are either in a *free list* or allocated to a region. Allocating data in infinite regions will fetch region pages from the free list. Contrary deallocating (or resetting) a region puts the region pages back into the free list.

9.1.1 Grammar for KAM

The grammar we use for KAM (Figure 9.1) is almost identical to the grammar found in "A Brief Introduction to Regions" [48].

We assume addresses to be word aligned, that is, with the two least significant bits zero. The two bits have a special meaning in pointers to regions. We use the first bit to hold the multiplicity (finite or infinite) and the second to hold the storage mode (atbot or attop). Setting the infinite bit is done by the primitive `set_inf_bit`. Setting the atbot bit is done by the primitive `set_atbot_bit`. Testing the bits are done by the boolean expressions `infinite`(ea) or `atbot`(ea). The infinite bit is cleared with `clear_atbot_bit`(ea). Infinite regions are allocated with `alloc`, reset with `reset_region` and deallocated with `dealloc_region`.

A constructor tag (`::`) is set with `cons_tag` and cleared with `decons`. The boolean expression `is_cons` tests wheter or not an ea is the constructor `::`; it is used to compile case statements only. The semantic objects used in the grammar are shown in Figure 9.2.

The code generator uses two macros to push and pop the stack.

$$
\begin{aligned}
\texttt{push}(\mathrm{ea}) \equiv\ & sp \texttt{ := } sp + 1\texttt{;}\\
& sp[\text{-}1] \texttt{ := } ea\\
\texttt{pop}(\mathrm{ea}) \equiv\ & ea \texttt{ := } sp[\text{-}1]\texttt{;}\\
& sp \texttt{ := } sp \text{ - } 1
\end{aligned}
$$

The code generator uses two temporary registers $t_1, t_2 \in PhReg$ that are not used by the register allocator.

$$
\begin{array}{rcl}
c & ::= & i \mid \texttt{nil} \mid lab \\
ea & ::= & c \mid ph \\
boolexp & ::= & ea \texttt{ = } ea \mid ea \texttt{ > } ea \mid ea \texttt{ < } ea \mid \ldots \\
 & \mid & \texttt{infinite}(ea) \mid \texttt{atbot}(ea) \mid \texttt{is\_cons}(ea) \\
bop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \ldots \\
 & & \\
stmt & ::= & ph \texttt{ := } ea \\
 & \mid & ph \texttt{ := } ph[o] \\
 & \mid & ph[o] \texttt{ := } ea \\
 & \mid & ph \texttt{ := } ea \; bop \; ea \\
 & \mid & \texttt{if } boolexp \texttt{ then } stmt \\
 & \mid & \texttt{if } boolexp \texttt{ then } stmt \texttt{ else } stmt \\
 & \mid & \texttt{nop} \\
 & \mid & \texttt{jmp}(ea) \\
 & \mid & \texttt{alloc}(ea, i) \\
 & \mid & \texttt{reset\_region}(ea) \\
 & \mid & \texttt{alloc\_region}(ea) \\
 & \mid & \texttt{dealloc\_region}(ea) \\
 & \mid & \texttt{cons\_tag}(ea) \\
 & \mid & \texttt{decons}(ea) \\
 & \mid & lab \texttt{ : } stmt \\
 & \mid & \texttt{clear\_atbot\_bit}(ea) \\
 & \mid & \texttt{set\_atbot\_bit} \\
 & \mid & \texttt{set\_inf\_bit} \\
 & \mid & stmt \texttt{ ; } stmt \\
 & & \\
fun & ::= & \texttt{fun } lab \texttt{ is } stmt \\
 & \mid & \texttt{fn } lab \texttt{ is } stmt \\
 & & \\
prg & ::= & fun \texttt{ ; } fun
\end{array}
$$

Figure 9.1: The grammar for KAM.

$$
\begin{array}{rcl}
lab & \in & Label \\
i & \in & Int \\
phreg & \in & PhReg \\
o & \in & Offset
\end{array}
$$

Figure 9.2: The semantic objects used in KAM. The set *Offset* denote offsets, either positive or negative. The set *PhReg* contains the machine registers.

9.2 Code Generation

The LineStmt code contains all information in the syntax necessary to generate KAM code without the use of any information about bound variables. The code generator $\mathcal{CG}$ simply does code expansion and takes only three integer arguments besides the LineStmt code. Top level declarations are translaled with

$$\mathcal{CG}^{\mathcal{T}} : TopDecl_{ls} \rightarrow Fun$$

using the function

$$\mathcal{CG} : \mathit{LineStmt} \times \mathit{Int}\ \times \mathit{Int}\ \rightarrow \mathit{Stmt}$$

on function bodies, where *Fun* is the set of functions ranged over by *fun* and *Stmt* is the set of KAM statements ranged over by *stmt*. The first constant is the size of the activation record ($size_{ff}$). The second constant is the size of the call convention frame ($size_{ccf}$), see Section 9.2.4.

The next few sections define auxiliary functions used by $\mathcal{CG}$.

9.2.1 Constants and Access Types

We use four functions to translate constants and access types. The functions take an extra argument being a temporary register that may be used to generate code (e.g., if an access type has to be fetched from or stored on the stack); the KAM resembles a RISC machine where an effective address is a constant or a machine register. The functions return a pair with the first component being code that needs to be evaluated before or after the value or register returned as the second component is used. The function **resolve_c_use** simply returns the constant:

resolve_c_use$(i) = i$
resolve_c_use$(\mathtt{nil}) = \mathtt{nil}$
resolve_c_use$(lab) = lab$

The function **resolve_aty_use** returns code to access a variable or region:

resolve_aty_use$(\mathtt{stack}(o),\ t,\ size_{ff}) = (t\ \mathtt{:=}\ sp[-size_{ff} + o], t)$
resolve_aty_use$(ph,\ t,\ size_{ff}) = (\mathtt{nop},\ ph)$
resolve_aty_use$(\mathtt{reg\_i}(o),\ t,\ size_{ff}) =$
$\quad(t\ \mathtt{:=}\ sp - size_{ff} + o\mathtt{;}\, \mathtt{set\_inf\_bit}(t), t)$
resolve_aty_use$(\mathtt{reg\_f}(o),\ t,\ size_{ff}) = (t\ \mathtt{:=}\ sp - size_{ff} + o, t)$

Figure 9.3 shows the convention used that sp always points at the first address after the function frame. Offset 0 is the offset at the bottom of the function frame. The call convention is below the function frame and the access types for parameters passed on the stack have negative offsets, see Chapter 7. The address of a slot at offset o in the activation record (or call

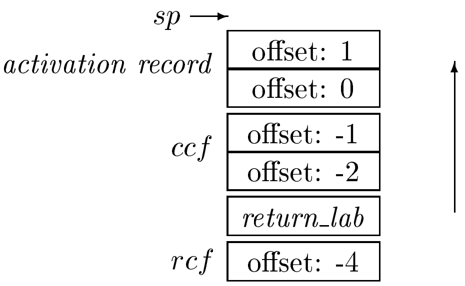

Figure 9.3: An example stack containing a call convention frame (ccf), return convention frame (rcf), return label and an activation record. A call convention contains ccf, rcf and the return label. The first slot in the activation record has offset 0 and the offsets in the call convention are negative. The stack grows upwards.

convention) relative to sp is then $sp - size_{ff} + o$. Addresses and offsets are in words.

The function **resolve_aty_def** is defined only for machine registers and spilled variables because it is not possible to store into (or define) a `letregion` bound region variable.

$\mathbf{resolve\_aty\_def}(\texttt{stack}(o),\ t,\ size_{ff}) = (sp[-size_{ff} + o]\ \texttt{:=}\ t, t)$
$\mathbf{resolve\_aty\_def}(ph,\ t,\ size_{ff}) = (\texttt{nop},\ ph)$

The function **resolve_se_use** is defined for either constants or access types.

$\mathbf{resolve\_se\_use}(c,\ t,\ size_{ff}) = (\texttt{nop},\mathbf{resolve\_c\_use}(c))$
$\mathbf{resolve\_se\_use}(aty,\ t,\ size_{ff}) = \mathbf{resolve\_aty\_use}(aty,\ t,\ size_{ff})$

9.2.2 Allocation Points

We have to consider both the multiplicity and storage mode when allocating into regions. For instance, allocating atbot in an infinite region is different from allocating atbot in a finite region; the region pages used in the infinite region must be freed before allocating but no resetting is done when allocating in the finite region. The storage modes were introduced in Section 3.3.3 on page 64. The multiplicities (`reg_f` and `reg_i`) were introduced in Chapter 8.

Storage modes and regions are categorized as follows in the LineStmt program ($phreg$ can be replaced by $\texttt{stack}(o)$):

`attop_li reg_i(`o`)`: the region is `letregion` bound and infinite. We allocate attop with the KAM instruction `alloc` because we know the region is infinite.

`attop_lf reg_f(`o`)`: the region is `letregion` bound and finite so storage has already been set aside on the stack.

`attop_lf` *phreg*: the region is `letregion` bound, finite and free to the function. Storage has already been allocated on the stack.

`attop_li` *phreg*: the region is `letregion` bound, infinite and free to the function. We use `alloc` to allocate storage.

`attop_ff` *phreg*: the region is an actual region argument to the current function or another function and free in the current function. The region can either be finite or infintie and we must check the multiplicity before allocating, that is, in the case *phreg* represents an infinite region we use `alloc`. If *phreg* represents a finite region then storage is already allocated on the stack.

`attop_fi` *phreg*: the region is infinite and we use `alloc` to allocate storage. A formal region parameter to a region polymorph function may either be finite or infinite but in the case that the function stores more than one time in the region (denoted by the last `i` in the annotation) then we know the region is infinite.

`sat_fi` *phreg*: the region is infinite. We check the storage bit to see wheter or not we shall reset the region before allocating.

`sat_ff` *phreg*: the region is either finite or infinite. We check the multiplicity and if infinite then we test the storage mode and if atbot then the region is reset. The test on multiplicity and storage mode may be done at the same time.

`atbot_li reg_i(`o`)`: the region is `letregion` bound and infinite. We reset the region before allocating.

`atbot_lf reg_f(`o`)`: the region is `letregion` bound and finite. Space has already been allocated on the stack and no resetting is necessary.

Combinations of storage modes and access types not shown above will never happen in the LineStmt program, except that *phreg* may be substituted with `stack(`o`)`.

Resolve Allocation Points

We use the function **resolve_ap** to generate the statement that will allocate memory according to an allocation point. The register ph_{res} is always used to hold the pointer to the allocated memory. If a finite region is represented by a register, ph say, then we cannot use ph for the result because ph may not be changed (e.g., clearing storage and infinite bits) if ph is used after the allocation.

resolve_ap(`attop_li reg_i`(o),ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` $sp - size_{ff} + o$`;`
  ph_{res} `:= alloc`(ph_{res}, n)
resolve_ap(`attop_lf reg_f`(o),ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` $sp - size_{ff} + o$
resolve_ap(`attop_lf` aty,ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` **resolve_aty_ap**(aty,$size_{ff}$)
resolve_ap(`attop_li` aty,ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` **resolve_aty_ap**(aty,$size_{ff}$)`;`
  ph_{res} `:= alloc`(ph_{res}, n)
resolve_ap(`attop_ff` aty,ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` **resolve_aty_ap**(aty,$size_{ff}$)`;`
  `if infinite`(ph_{res}) `then` ph_{res} `:= alloc`(ph_{res},n)
resolve_ap(`attop_fi` aty,ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` **resolve_aty_ap**(aty,$size_{ff}$)`;`
  ph_{res} `:= alloc`(ph_{res},n)
resolve_ap(`sat_fi` aty,ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` **resolve_aty_ap**(aty,$size_{ff}$)`;`
  `if atbot`(ph_{res}) `then reset_region`(ph_{res})`;`
  ph_{res} `:= alloc`(ph_{res},n)
resolve_ap(`sat_ff` aty,ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` **resolve_aty_ap**(aty,$size_{ff}$)`;`
  `if infinite`(ph_{res}) `then`
    `if atbot`(ph_{res}) `then`
      `reset_region`(ph_{res})`;`
    ph_{res} `:= alloc`(ph_{res},n)
resolve_ap(`atbot_li reg_i`(o),ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` $sp - size_{ff} + o$`;`
  `reset_region`(ph_{res})`;`
  ph_{res} `:= alloc`(ph_{res}, n)
resolve_ap(`atbot_lf reg_f`(o),ph_{res},n,$size_{ff}$) =
  ph_{res} `:=` $sp - size_{ff} + o$

The access type aty in the above function is always either a machine register or a spilled variable (i.e., `stack`(o)). The auxiliary function **resolve_aty_ap** is defined as follows:

resolve_aty_ap(ph,$size_{ff}$) $= ph$
resolve_aty_ap(`stack`(o),$size_{ff}$) $= sp[-size_{ff} + o]$

9.2.3 Set Storage Modes

Storage modes are set when passing regions as arguments to `letrec` bound functions, (e.g., when a region vector is constructed). The following specifies how the storage bits are set (*phreg* can be replaced by `stack`(o)).

`attop_li reg_i`(o): the atbot bit is not set so no change is necessary.

`attop_lf reg_f`(o): the region is finite and the atbot bit is therefore insignificant.

`attop_lf` *phreg*: the region is finite and the atbot bit is therefore insignificant.

`attop_li` *phreg*: even though the region is free to the function then the atbot bit is not set because it is `letregion` bound. It is not necessary to clear the atbot bit.

`attop_ff` *phreg*: the region may or may not be infinite so we have to check the multiplicity and then clear the atbot bit if the multiplicity is infinite. Actually, we may blindly clear the atbot bit no matter the multiplicity.

`attop_fi` *phreg*: the region is infinite and we clear the atbot bit.

`sat_fi` *phreg*: the atbot bit is already set appropriately.

`sat_ff` *phreg*: the atbot bit is already set appropriately.

`atbot_li reg_i`(o): we set the atbot bit.

`atbot_lf reg_f`(o): the atbot bit is insignificant.

`atbot_li` *phreg*: we set the atbot bit.

`atbot_lf` *phreg*: the atbot bit is insignificant.

The atbot bit is never set on `letregion` bound regions when they are created so we do not have to explicitly clear the atbot bit for those regions.

Resolve Storage Modes

The function **resolve_sm_regvec** sets or clears the atbot bit in regions passed to region polymorphic functions as specified in the previous section. The function is given a temporary register t that is used if necessary.

resolve_sm_regvec(`attop_li reg_i`(o),t,$size_{ff}$) =
  (t := $sp - size_{ff} + o$; `set_inf_bit`(t),t)
resolve_sm_regvec(`attop_lf reg_f`(o),t,$size_{ff}$) = (t := $sp - size_{ff} + o, t$)
resolve_sm_regvec(`attop_lf` ph,t,$size_{ff}$) = (`nop`, ph)
resolve_sm_regvec(`attop_lf stack`(o),t,$size_{ff}$) = (t := $sp[-size_{ff} + o], t$)
resolve_sm_regvec(`attop_li` ph,t,$size_{ff}$) = (`nop`, ph)
resolve_sm_regvec(`attop_li stack`(o),t,$size_{ff}$) = (t := $sp[-size_{ff} + o], t$)
resolve_sm_regvec(`attop_ff` ph,t,$size_{ff}$) =
  (t := ph; `clear_atbot_bit`(t), t)
resolve_sm_regvec(`attop_ff stack`(o),t,$size_{ff}$) =
  (t := $sp[-size_{ff} + o]$; `clear_atbot_bit`(t), t)
resolve_sm_regvec(`attop_fi` ph,t,$size_{ff}$) =
  (t := ph; `clear_atbot_bit`(t), t)
resolve_sm_regvec(`attop_fi stack`(o),t,$size_{ff}$) =
  (t := $sp[-f\_szie + o]$; `clear_atbot_bit`(t), t)
resolve_sm_regvec(`sat_fi` ph,t,$size_{ff}$) = (`nop`, ph)
resolve_sm_regvec(`sat_fi stack`(o),t,$size_{ff}$) = (t := $sp[-size_{ff} + o]$, t)
resolve_sm_regvec(`sat_ff` ph,t,$size_{ff}$) = (`nop`, ph)
resolve_sm_regvec(`sat_ff stack`(o), t) = (t := $sp[-size_{ff} + o]$, t)
resolve_sm_regvec(`atbot_li reg_i`(o),t,$size_{ff}$) =
  (t := $sp - size_{ff} + o$; `set_inf_bit`(t); `set_atbot_bit`(t),t)
resolve_sm_regvec(`atbot_lf reg_f`(o),t,$size_{ff}$) = (t := $sp - size_{ff} + o, t$)
resolve_sm_regvec(`atbot_li` ph,t,$size_{ff}$) =
  (t := ph; `set_atbot_bit`(t),t)
resolve_sm_regvec(`atbot_lf` ph,t,$size_{ff}$) = (`nop`,ph)

We note, that in many cases (e.g., when the access type is `stack`(o)) it is possible to add the constants *atbot_bit* and *inf_bit* when the address is calculated instead of using the `set_atbot_bit` and `set_inf_bit` primitives. The constants must be defined such that they set the appropriate bit when added.

9.2.4 Call Convention

The register allocation phase (Chapter 5) inserts statements around the function calls such that arguments and results passed in registers are actually put in the registers before the call and retrieved from the registers after the call. However, it is the responsibility of the code generator to put arguments, not passed in registers, on the stack and retrieve results, not passed in registers, from the stack on return from the call. This section specifies how this can be done.

We need a convention for storing a call convention on the stack. A call convention is split in three parts: a call convention frame (ccf), a return

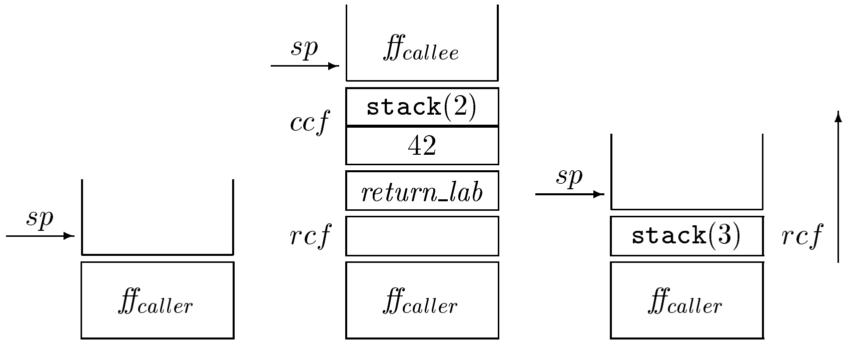

Figure 9.4: At left we have the stack before the application. At center we have the stack at entry to the called function and at right the stack at return from callee. The stack grows upwards.

label[1] and a return convention frame (rcf). The call convention is allocated before callee is called. On return from callee we have rcf allocated. Figure 9.4 shows the content of an example stack before an application, at entry to a called function and at return.

A call convention uniquely specifies which arguments and results are passed on the stack and in what order. The method we use is to first convert the application into a temporary call convention called the *actual call convention*. We then use a function **resolve_act_cc** that, given an actual call convention as argument, returns a statement that stores arguments into the call convention frame (ccf) and a statement fetching results from the return convention frame (rcf). The function also returns the size of the call and return convention frames.

The actual call convention for an ordinary application

$$\langle aty_1, \ldots, aty_h \rangle \texttt{ := } se_{ck}\ \langle se_{a1}, \ldots, se_{an} \rangle\ \langle se_c \rangle\ \langle se_{f_1}, \ldots, se_{f_m} \rangle$$

or a region polymorphic application

$$\langle aty_1, \ldots, aty_h \rangle \texttt{ := } lab_{ck}\ \langle se_{a1}, \ldots, se_{an} \rangle\ \langle se_r \rangle\ \langle a_1, \ldots, a_l \rangle\ \langle se_c \rangle\ \langle se_{f_1}, \ldots, se_{f_m} \rangle$$

is

$acc = \{$clos= se_c,
free= $[se_{f_1}, \ldots, se_{f_m}]$,
args= $[se_{a_1}, \ldots, se_{a_n}]$,
reg_vec= se_r,

[1]We always put the return label on the stack. However, we believe it is better to let the return label be an ordinary argument such that it can be passed in a register if the register pressure is low.

reg_args= $[a_1, \ldots, a_l]$,
res= $[aty_1, \ldots, aty_h]$ }

The function **resolve_act_cc** uses the same priorities as **resolve_cc** on page 101 (and **init_F** on page 137) to find the stack offsets. Consider the application

$$\langle ph_9, \texttt{stack}(3)\rangle \texttt{ := } lab_{ck}\ \langle ph_3, 42, \texttt{stack}(2)\rangle\ \langle ph_2\rangle\ \langle\rangle\ \langle ph_1\rangle\ \langle\rangle$$

with *args_phreg* $= [ph_1, ph_2, ph_3]$ and *res_phreg* $= [ph_9]$. The corresponding actual call convention is

acc = {clos= ph_1,
free= [ ],
args= $[ph_3, 42, \texttt{stack}(2)]$,
reg_vec= ph_2,
reg_args= [ ],
res= $[ph_9, \texttt{stack}(3)]$ }

Two arguments (42,`stack`(2)) and the result to go in `stack`(3) are passed on the stack, see Figure 9.4. The function **resolve_act_cc**(*acc*) returns the statement

$\text{pop}(t)$;
$sp[-size_{ff} + 3]$ `:=` t

that fetches the result from the return convention frame and the statement

`push`(42);
t `:=` $sp[-size_{ff} - 1 + 2]$;
`push`(t)

that stores the arguments in the call convention frame. We note that the stack pointer is pointing at the first address after the current function frame when the code to store arguments is executed. The stack pointer points at the first address after the return convention frame when the code to fetch results is executed. The offsets used when accessing slots in the function frame depends on the number of `pop` and `push` statements executed.

9.2.5 Functions

At entry to a function (either `letrec` or ordinary) we allocate the function frame on the stack. The frame size ($size_{ff}$) is annotated on top level declarations (computed in Chapter 7). The call and return convention is explicit in the body of a function by either refering to machine registers or stack positions directly. Arguments and results passed on the stack are referenced directly with an offset from the stack pointer. The stack annotations are calculated in Chapter 7 and Figure 9.4 shows an example stack when calling a function.

The code for the body of a function is organized as follows:

- code to allocate the function frame.
- code for the body of the function.
- code to return to the calling function by loading the return address from the stack.
- code to set the stack pointer at the first address after the return convention frame (i.e., rcf). The function frame for callee, call convention frame (ccf) and return label is thus deallocated. Caller needs rcf to fetch results passed on the stack.

Tail calls makes it necessary to deallocate both function and call convention frames before leaving the function; and not at the application point.

$$
\begin{array}{l}
\mathcal{CG}^{\mathcal{T}}\ [\![\lambda_{lab}^{\mathtt{fun}(size_{ff})}\ cc\ \texttt{=>}\ ls]\!]\ =\\
\quad \mathtt{fun}\ lab\ \mathtt{is}\\
\qquad sp\ \texttt{:=}\ sp + size_{ff};\\
\qquad \mathcal{CG}\ [\![ls]\!]\ \ size_{ff}\ \mathbf{size\_ccf}(cc)\ ;\\
\qquad sp\ \texttt{:=}\ sp - size_{ff} - \mathbf{size\_ccf}(cc);\\
\qquad pop(t);\\
\qquad \mathtt{jmp}\ t
\end{array}
$$

Figure 9.5 shows the stack pointer at entry and exit of a function. The function **size_ccf** returns the size of the call convention frame (i.e., the number of arguments pushed on the stack). An ordinary function (`fn`) is translated likewise. Top level declarations are translated sequentially:

$$
\begin{array}{l}
\mathcal{CG}^{\mathcal{T}}\ [\![top\_decl_1\,;top\_decl_2]\!]\ =\\
\quad \mathcal{CG}^{\mathcal{T}}\ [\![top\_decl_1]\!]\ ;\\
\quad \mathcal{CG}^{\mathcal{T}}\ [\![top\_decl_2]\!]
\end{array}
$$

9.2.6 Applications

An ordinary application involves the following phases:

- store registers, that must be saved across the function call. This is already done, see Chapter 6.
- allocate the return convention frame.
- push the return address on the stack. The code generator needs to create a new return label.

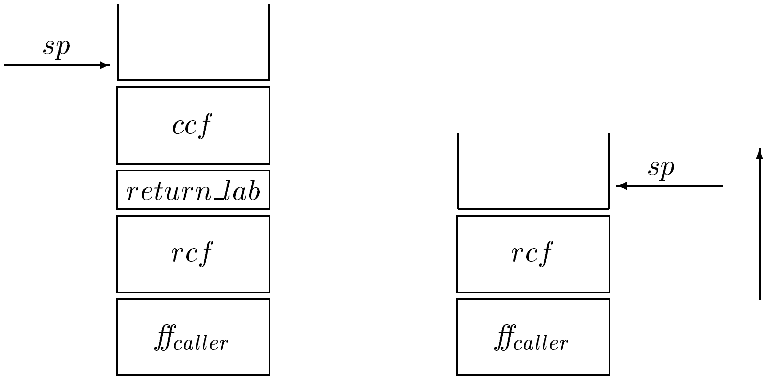

Figure 9.5: At entry to the function (at left), the stack pointer points at the first free address after the call convention frame (ccf). At exit from the function (at right), the stack pointer points at the first free address after the return convention frame (rcf). The stack grows upwards.

- allocate the call convention frame.
- put arguments in machine registers or into the call convention frame as specified by the call convention. We only have to consider arguments to go in the call convention frame. Arguments to go in machine registers are resolved by the register allocator, see Chapter 5.
- jump to the function.

On return from the call we have to

- move the result values from the return convention frame into the places (either machine registers or stack positions) where they are supposed to be.
- the stack pointer points at the first address after rcf so we must deallocate rcf.
- fetch values that were stored across the function call. This is already done, see Chapter 6.

The translation function for a function call to a region polymorphic function is thus:

$\mathcal{CG}$ $[\![\langle aty_1, \ldots, aty_h\rangle$ `:=`
    $se_{\texttt{funcall}}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{reg}\rangle\ \langle a_1, \ldots, a_l\rangle$
        $\langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle]\!]\ \ size_{\mathit{ff}}\ \_ =$
  let
    val $(st_{args}, size_{ccf}, st_{res}, size_{rcf}) =$ **resolve_act_cc**({clos=se_{clos}, ...})
    val $return\_lab =$ **fresh_lab**("app")

$$
\begin{array}{l}
\quad \textbf{val}\ st_{jmp} = \textbf{resolve\_jmp}(se, size_{rcf} + size_{ccf} + 1 + size_{ff}) \\
\textbf{in} \\
\qquad\qquad\qquad sp\ \texttt{:=}\ sp + size_{rcf}\texttt{;} \\
\qquad\qquad\qquad \texttt{push}(return\_lab)\texttt{;} \\
\qquad\qquad\qquad st_{args}\texttt{;} \\
\qquad\qquad\qquad st_{jmp}\texttt{;} \\
\quad return\_lab\texttt{:}\ st_{res} \\
\textbf{end}
\end{array}
$$

An application to an ordinary function is translated likewise. We use the function **resolve_jmp** to generate jump code depending on where the target label is; either as a label constant or in a closure.

$$
\begin{array}{l}
\textbf{resolve\_jmp}(lab, sp_{adjust}) = \texttt{jmp}\ lab \\
\textbf{resolve\_jmp}(\texttt{stack}(o),\ sp_{adjust}) = \\
\quad t\ \texttt{:=}\ sp[-sp_{adjust} + o]\texttt{;} \\
\quad t\ \texttt{:=}\ t[0]\texttt{;} \\
\quad \texttt{jmp}\ t \\
\textbf{resolve\_jmp}(ph, sp_{adjust}) = \\
\quad t\ \texttt{:=}\ ph[0]\texttt{;} \\
\quad \texttt{jmp}\ t
\end{array}
$$

The constant sp_{adjust} is the distance from top of ccf to the first address in ff_{caller}, see Figure 9.5 at left.

Tail Calls

A tail call is simpler than an ordinary call because we do not return to caller again; there are no more code to execute. We do not have to setup a return convention frame and we shall not push a return address on the stack.

Because a tail call is an implicit return from the current function we must deallocate the current function frame together with the current call convention frame before jumping to the called function. We must also fetch some callee save registers as explained in Chapter 6:

- fetch callee save registers.
- deallocate the current function frame.
- deallocate the current call convention frame.
- allocate a new call convention frame.
- put arguments into the call convention frame.
- jump to the function.

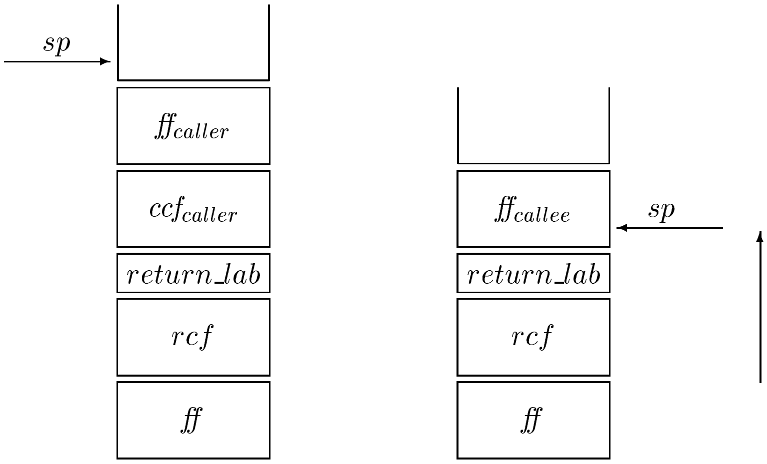

Figure 9.6: At left we have the stack before a tail call. In the general case we must deallocate ff_{caller} and ccf at the same time that we build a new ccf for the called function. This is, in general, not possible to do efficiently so we make the assumption that all functions called in a tail call may not have a ccf. At right, we have the stack at entry to the called function. The stack grows upwards.

Figure 9.6 shows the stack before a tail call. If callee expects arguments on the stack then we must overwrite ccf_{caller} and ff_{caller} with ccf_{callee}. However, this is difficult because both ccf_{caller} and ff_{caller} may be needed when computing the arguments (i.e., we cannot overwrite ccf_{caller} and ff_{caller} while building ccf_{callee}). This may be solved by copying ccf_{caller} into ff_{caller} at entry to the caller and then reserve as much space between *return_lab* and ff_{caller} as is needed for all possible calls inside caller. The amount of space needed is a local property of caller. We do not find this worth the trouble because a tail call is not a fast call anymore. We decide to convert all tail calls calling functions taking arguments on the stack into ordinary calls and thereby keep tail calls fast. We do not believe this restriction to be a problem because we have many argument registers. We make sure that algorithm $\mathcal{F}$ (see Chapter 3) does not make functions non closure implemented if they get more arguments than there are argument registers. If other analyses (like unboxing records) also confer to this restriction then we will never convert a tail call into a non tail call.

There is no restriction on the number of return values, that is, rcf may always be used. We may weaken the restriction and use ccf_{callee} in a tail call in the case that ccf_{callee} only contains actual region arguments being a subset of the region arguments already in ccf_{caller}, (see [49, Chapter 14]).

$$\begin{aligned}
&\mathcal{CG}\ [\![\langle aty_1, \ldots, aty_h \rangle \texttt{ := } \\
&\qquad lab_{\texttt{funjmp}(ls)}\ \langle se_1, \ldots, se_n \rangle\ \langle se_{reg} \rangle\ \langle a_1, \ldots, a_l \rangle \\
&\qquad\qquad\qquad \langle se_{clos} \rangle\ \langle se_{f_1}, \ldots, se_{f_m} \rangle]\!]\ \ size_{\mathit{ff}}\ size_{ccf} =
\end{aligned}$$

$$
\begin{array}{l}
\quad \textbf{if size\_act\_ccf}(\{\text{clos}=se_{clos}, \ldots\}) > 0 \textbf{ then} \\
\quad\quad \mathcal{CG}\ [\![\langle aty_1, \ldots, aty_h\rangle \texttt{ := } \\
\quad\quad\quad\quad lab_{\texttt{funcall}}\ \langle se_1, \ldots, se_n\rangle\ \langle se_{reg}\rangle\ \langle a_1, \ldots, a_l\rangle \\
\quad\quad\quad\quad\quad\quad \langle se_{clos}\rangle\ \langle se_{f_1}, \ldots, se_{f_m}\rangle]\!]\ \ size_{ff}\ size_{ccf} \\
\quad \textbf{else} \\
\quad\quad \mathcal{CG}\ [\![ls]\!]\ \ size_{ff}\ size_{ccf}\texttt{;} \\
\quad\quad sp \texttt{ := } sp - size_{ff} - size_{ccf}\texttt{;} \\
\quad\quad \texttt{jmp}\ lab
\end{array}
$$

The statements ls fetches callee save registers, see Chapter 6. The callee save registers are not fetched if the tail call is turned into an ordinary call. Tail calls to ordinary functions are handled likewise.

9.2.7 Statements

In this section we show $\mathcal{CG}$ for some of the remaining statements. The translation of statements not shown are similar to the translations shown. They are all straight–forward given the previously defined functions.

$$
\begin{array}{l}
\mathcal{CG}\ [\![aty \texttt{ := } (se_1, \ldots, se_n)]\!]\ \ size_{ff}\ \_ = \\
\quad \textbf{let} \\
\quad\quad \textbf{val}\ (st_r, r) = \textbf{resolve\_aty\_def}(aty,\ t_1,\ size_{ff}) \\
\quad\quad \textbf{val}\ st_{alloc} = \textbf{resolve\_ap}(a,\ r,\ n,\ size_{ff}) \\
\quad\quad \textbf{val}\ (t_i, st_i) = \textbf{resolve\_se}(se_i, t_2, size_{ff}) \\
\quad \textbf{in} \\
\quad\quad st_{alloc}\texttt{;} \\
\quad\quad st_1\texttt{;} \\
\quad\quad r[0] \texttt{ := } t_1\texttt{;} \\
\quad\quad\quad \vdots \\
\quad\quad st_n\texttt{;} \\
\quad\quad r[n-1] \texttt{ := } t_n\texttt{;} \\
\quad\quad st_r \\
\quad \textbf{end} \\
\mathcal{CG}\ [\![aty \texttt{ := } [a_1, \ldots, a_n]_{regvec}]\!]\ \ size_{ff}\ \_ = \\
\quad \textbf{let} \\
\quad\quad \textbf{val}\ (st_r, r) = \textbf{resolve\_aty\_def}(aty,\ t_1,\ size_{ff}) \\
\quad\quad \textbf{val}\ st_{alloc} = \textbf{resolve\_ap}(a,\ r,\ n,\ size_{ff}) \\
\quad\quad \textbf{val}\ (t_i, st_i) = \textbf{resolve\_sm\_regvec}(a_i, t_2,\ size_{ff}) \\
\quad \textbf{in} \\
\quad\quad st_{alloc}\texttt{;} \\
\quad\quad st_1\texttt{;} \\
\quad\quad r[0] \texttt{ := } t_1\texttt{;} \\
\quad\quad\quad \vdots \\
\quad\quad st_n\texttt{;}
\end{array}
$$

$\quad\quad r[n-1]$:= t_n;
$\quad\quad st_r$
$\quad$ **end**
$\mathcal{CG}$ $[\![ls_1$; $ls_2]\!]$ $size_{ff}\ size_{ccf} =$
$\quad$ $\mathcal{CG}$ $[\![ls_1]\!]$ $size_{ff}\ size_{ccf}$; $\mathcal{CG}$ $[\![ls_2]\!]$ $size_{ff}\ size_{ccf}$
$\mathcal{CG}$ $[\![$`flush`$(aty,o)]\!]$ $size_{ff}$ _ $=$
$\quad$ **let**
$\quad\quad$ **val** $(st, t) =$ **resolve_aty_use**$(aty,\ t_1,\ size_{ff})$
$\quad$ **in**
$\quad\quad st$;
$\quad\quad sp[-size_{ff} + o]$:= t
$\quad$ **end**
$\mathcal{CG}$ $[\![$`fetch`$(aty,o)]\!]$ $size_{ff}$ _ $=$
$\quad$ **let**
$\quad\quad$ **val** $(st, t) =$ **resolve_aty_def**$(aty,\ t_1,\ size_{ff})$
$\quad$ **in**
$\quad\quad t$:= $sp[-size_{ff} + o]$;
$\quad\quad st$
$\quad$ **end**
$\mathcal{CG}$ $[\![aty_1$:= #$n(aty_2)]\!]$ $size_{ff}$ _ $=$
$\quad$ **let**
$\quad\quad$ **val** $(st_r, t_r) =$ **resolve_aty_def**$(aty_1,\ t_1,\ size_{ff})$
$\quad\quad$ **val** $(st_s, t_s) =$ **resolve_aty_use**$(aty_2,\ t_1,\ size_{ff})$
$\quad$ **in**
$\quad\quad st_s$;
$\quad\quad t_r$:= $t_s[n-1]$;
$\quad\quad st_r$
$\quad$ **end**
$\mathcal{CG}$ $[\![$`letregioni` o `in` ls `end`$]\!]$ $size_{ff}\ size_{ccf} =$
$\quad t_1$:= $sp - size_{ff} + o$;
$\quad$ `alloc_region`(t_1);
$\quad$ $\mathcal{CG}$ $[\![ls]\!]$ $size_{ff}\ size_{ccf}$;
$\quad t_1$:= $sp - size_{ff} + o$;
$\quad$ `dealloc_region`(t_1)
$\mathcal{CG}$ $[\![aty_1$:= ::$aty_2]\!]$ $size_{ff}$ _ $=$
$\quad$ **let**
$\quad\quad$ **val** $(st_r, t_r) =$ **resolve_aty_def**$(aty_1,\ t_1,\ size_{ff})$
$\quad\quad$ **val** $(st_s, t_s) =$ **resolve_aty_use**$(aty_2,\ t_1,\ size_{ff})$
$\quad$ **in**
$\quad\quad st_s$;
$\quad\quad t_r$:= `cons_tag`(t_s);
$\quad\quad st_r$
$\quad$ **end**
$\mathcal{CG}$ $[\![aty$:= $se_1\ bop\ se_2]\!]$ $size_{ff}$ _ $=$

```
  let
    val (st_r, t_r) = resolve_aty_def(aty, t_1, size_ff)
    val (st_s1, t_s1) = resolve_se(se_1, t_1, size_ff)
    val (st_s2, t_s2) = resolve_se(se_2, t_2, size_ff)
  in
    st_s1;
    st_s2;
    t_r := t_s1 bop t_s2;
    st_r
  end
CG [[case se_1 of c => ls_2 | _ => ls_3]] size_ff size_ccf =
  let
    val (st, t) = resolve_se(se_1, t_1, size_ff)
  in
    st;
    if (t = c) then
      CG [[ls_2]] size_ff size_ccf
    else
      CG [[ls_3]] size_ff size_ccf
  end
CG [[case se_1 of ::aty => ls_2 | _ => ls_3]] size_ff size_ccf =
  let
    val (st_s, t_s) = resolve_se(se_1, t_1, size_ff)
    val (st_r, t_r) = resolve_aty_def(aty, t_1, size_ff)
  in
    st_s;
    if is_cons(t_s) then
      t_r := decons(t_s);
      st_r;
      CG [[ls_2]] size_ff size_ccf
    else
      CG [[ls_3]] size_ff size_ccf
  end
```

9.3 Example

The example function (*fn_xs*) from page 147 is shown in Figure 9.7 translated into KAM code.

```
fn fn_xs is
    sp := sp + 14;
    sp[-14 + 0] := ph4;
    sp[-14 + 1] := ph5;
    ph4 := ph5;
    if (ph2 = nil) then
        ph1 := ph4[1]
    else
        if is_cons(ph2) then
            ph1 := decons(ph2);
            ph5 := ph1[0];
            t1 := ph1[1];
            sp[-14 + 2] := t;
            ph2 := sp - 14 + 12;
            ph2[0] := sp - 14 + 7;
            ph2[1] := sp - 14 + 3;
            ph3 := ph4[2];
            ph1 := ph4[3];
            push(fold_ret);
            jmp foldl;
fold_ret:   ph3 := ph1;
            sp[-14 + 11] := ph3;
            ph1 := ph4[2];
            ph2 := ph5;
            push(fn_ret1);
            t1 := ph1[0];
            jmp t1;
fn_ret1:    ph2 := ph4[1];
            push(fn_ret2);
            t1 := ph1[0];
            jmp t1;
fn_ret2:    ph3 := sp[-14 + 11];
            ph2 := ph1;
            ph1 := ph3;
            push(fn_ret3);
            t1 := ph3[0];
            jmp t1;
fn_ret3:    t1 := sp[-14 + 2];
            ph2 := t1;
            push(fn_ret4);
            t1 := ph1[0];
            jmp t1;
fn_ret4:
        ph4 := sp[-14 + 0];
        ph5 := sp[-14 + 1];
        sp := sp - 14;
        pop(t1);
        jmp t1
```

Figure 9.7: The function *fn_xs* compiled into KAM code.

Part III

The Garbage Collector

Chapter 10

Basic Garbage Collection Algorithms

In this chapter, we discuss basic garbage collection algorithms and introduce a one phase and a two phase garbage collection abstraction. The immediate differences between the algorithms presented are emphasized.

The chapter motivates the choice of garbage collector to use with region inference. Region inference introduces several problems that are not found in systems based solely on garbage collection and the choice of garbage collector is therefore not obvious. A main difference is the number of heaps. A system based on region inference does not have one or at most a small bounded number of heaps but several maybe thousands, (i.e., each region has its own *region heap*) and each region heap does not constitute a continuous memory area but a list of linked region pages, see Chapter 2.

We illustrate some of the problems of measuring and comparing garbage collection algorithms. These problems are relevant when evaluating our implementation.

A comprehensive survey of garbage collection algorithms is found in Wilson's paper [58]. The survey has been very helpful both as a survey and as a source of reference. The two phase garbage collection abstraction presented below is borrowed from the survey. Another excellent source is the book "Garbage Collection – Algorithms for Automatic Dynamic Memory Management", by Richard Jones and Rafael Lins [30].

10.1 Fundamentals

Garbage Collection may be defined in several ways and a definition often used is that *garbage collection is a mechanism which automatically reclaims storage for the purpose of reuse.* This is of course what we use garbage collection for, but actually region inference does the same thing and the two methods are very different. It is better to characterize garbage collection as an *automatic and dynamic method for reclaiming storage for the purpose of reuse.* Region Inference is then characterized as *an automatic and static method for reclaiming storage for the purpose of reuse.*

A static method does not interrupt program execution and is probably more time efficient than a dynamic method. Allocation and deallocation statements are inserted into the program at compile time. A dynamic method interrupts program execution and scans the memory to find the set of values needed for the rest of the execution. This takes time but is also more precise if the set of live objects can be determined precisely.

We use two simple *garbage collection abstractions.* The first abstraction consists of two phases:

1. the detection or marking phase is the phase that splits the allocated storage into a set of *live* objects[1] and a set of *dead* objects. This is called the *garbage detection* phase [58].

2. in the second phase, we collect all dead objects and thereby free storage for new objects to be allocated. This is called the *garbage reclamation* phase [58].

As we will see, many garbage collection algorithms fits into the above two–phase abstraction. A one–phase abstraction is possible using two heaps:

1. find all live objects, and each time a new live object is found copy it into a new heap. Afterwards the old heap may be freed.

An object is *dead*, opposed to *live*, if it is no longer needed by the running program.[2] It is not necessarily the case that a garbage collector finds all dead objects, (i.e., conservative garbage collection). However, the set of live objects must always contain, at least, all objects needed to finish the computation. If not, the garbage collector is not *safe.*

How do we split the memory into dead and live objects and ensure that none of the live objects are deallocated? The set of live objects are normally defined to be the set of objects that can be reached from a *root–set.* A possible root–set contains globally defined variables, variables allocated in activation records and registers. The root–set depends on when and where the program may be interrupted and how the compiler is organized. The root–set for the ML-Kit is defined in Chapter 13.

Given the set of live objects (*Live*) we require that:

[1] In this section, we use the word object for one or more memory cells allocated as one unit (e.g., a closure).

[2] Note, that live as used in this chapter is different from live used in liveness analyses, see Chapter 5. If an object is needed by the program then it is defined live here. In the liveness analyses, however, there may exists objects defined live by the liveness analysis that are not needed by the program, (i.e., the liveness analysis is an approximation to live as defined here).

> for all $obj \notin Live$, the object obj cannot be reached by any pointer traversal from any object $live \in Live$.

If an object $obj \notin Live$ can be reached from another object $live \in Live$ then we cannot guarantee that obj is not needed to finish the computation.

To find the set of live objects it is necessary to know whether or not a given object contains scalars and/or pointers. For now, we assume objects to be tagged with such information. Tagging in the ML-Kit is discussed in Chapter 12.

We discuss two approaches to find the set of live objects. The first, called *reference counting* is suitable for implementing *real–time* garbage collection (i.e., the maximal program interruption time is bounded). The other approach called *tracing* traverses memory starting from the root–set. The algorithms presented here do not achieve real–time garbage collection but can be modified to do so.

10.2 Reference Counting

Reference counting [16] is a garbage collection technique which in its basic version comes close to real–time collection. Each object holds a *reference count*, which is the number of pointers pointing at the object.

An object is considered live, iff its reference count is not zero. The reference count is initialized to one when the object is allocated for the first time and incremented each time the object is allocated again (i.e., a new pointer is pointing at the object).

An object is deallocated by decrementing the reference count, and the object is reclaimed when the reference count reaches zero. The object being reclaimed may point at other objects. If so the reference count of the "point at" objects are decremented too, and if those reference counts reach zero they must be reclaimed too. Deallocating an object may therefore start a series of deallocations and reclamation of objects. This is why the basic reference counting algorithm is not a real–time garbage collector; one deallocation may lead to an unbounded number of deallocations. However, the problem is easily solved with lazy deallocation. An object is deallocated by pushing a reference to the object onto a stack of deallocated but non reclaimed objects. References to objects are then popped from the stack and reclaimed with a given frequency that cope with the real–time requirements. One possibility is to reclaim one object at each allocation.

Garbage detection is performed by the bookkeeping of reference counts. The garbage reclamation phase is when objects are deallocated. The bookkeeping of reference counts is limited to a small number of instructions but they are executed every time an object is allocated and deallocated.

A reference count takes up space, and a word is necessary if a precise bookkeeping of reference counts is needed. A shorter reference count field can be used as for instance three bits with the counts $1 \ldots 7$ and ∞. The reference count ∞ cannot be decremented and objects with the count ∞ must be reclaimed with another garbage collection technique. A hybrid garbage collector using a one–bit reference count and a tracing garbage collector (e.g., mark–sweep or mark–compact, see the sections below) has been proposed [59]. The idea is that while the tracing garbage collector is idle, the mark bit may be used to distinguish between objects known to have a reference count of one and objects having another reference count. Deallocating an object with a reference count of one may be reclaimed instantly whereas other objects are reclaimed by the tracing garbage collector. The method delays the invocation of the tracing garbage collector significantly in environments where most objects are referenced only once.

In programs based solely on reference counting, it is necessary to make a conservative approximation of the maximum number of possible references to each object, when deciding the size of the reference count. This is in general difficult to estimate, especially for a compiler where it depends on how the programmer uses the programming language being compiled. For instance, in Standard ML, the function $\lambda a.(a, a)$ creates two references to a each time the function is applied (we assume a represents a boxed value). Manipulation of recursive data structures may create many pointers to the same value. If we have the lists l_1 and l_2 then the function

```
fun splice l1 l2 = l1 @ l2
```

called with `splice` l_1 l_2 creates another pointer to the elements in list l_1 if we use the implementation of `@` as defined in SML90 [35].[3] The number of pointers to the elements in l_2 remains the same except for the first element.

A major problem with reference counting is the inability to reclaim cyclic structures. It is possible to have a cycle between two objects even though there is no pointer from the root–set to the objects.

Cycles occur rarely in Standard ML, but here is an example showing that they can occur, see Figure 10.1.

```
datatype cycle = A of cycle ref
               | B of cycle ref
               | None;
val res =
  let
    val n = None
    val a = A (ref n)
    val b = B (ref a)  (* 1 *)
    val _ = case a
              of A(r) => r:= b (* 2 *)
```

[3]The function is defined as `fun nil @ M = M | (x::L) @ M = x::(L @ M)`.

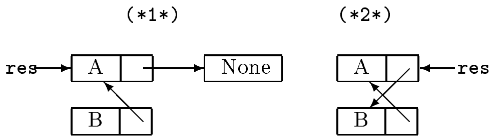

Figure 10.1: A possible representation of the values at the two situations `(*1*)` and `(*2*)`. When the reference is updated the value `None` will be deallocated, and hence freed.

```
                  | _     => print "Crash"
  in
    a
  end
```

When the variable `res` is assigned the value of `a` (A) then the reference count is two (from `res` and the value of `b` (B)). When `res` is deallocated then the reference count for A is decremented to one. As no live value points at A and the reference count is one, the values A and B are never reclaimed.

It turns out that a purely functional language creates cyclic data structures in a restricted manner, which can be treated specially and reference counting is then applicable [7]. It is also possible to handle more general circular structures, such as doubly linked lists. Each circular structure is considered a single group. We have a reference count for each group and if the count reaches zero then the entire group is deallocated [11].

The bookkeeping of reference counts puts a proportional factor on the work done by the running program; the bookkeeping uses time on both allocation and deallocation of objects. Reference counting may therefore not be time efficient, when there are many short lived values, which is common in functional languages.

It is possible to defer the bookkeeping of local variables [18]. Instead of updating reference counts on all objects we consider heap–allocated objects only, that is, if a heap–allocated object points at another object then we update the reference count. The reference count does not include pointers from stack allocated objects. Instead of reclaiming an object when the reference count reaches zero we insert the object in a list of potentially dead objects. The stack is scanned with a given frequency and potentially dead objects are only dead (and reclaimed) if they are not pointed at by any stack allocated object.

Basic reference count algorithms consider the heap as one chunk of data. Objects removed from the heap are inserted into one or more free lists. Allocating an object involves searching the free lists.

The most important advantages of reference counting is the suitability for real–time garbage collection and the fast reclamation of deallocated objects.

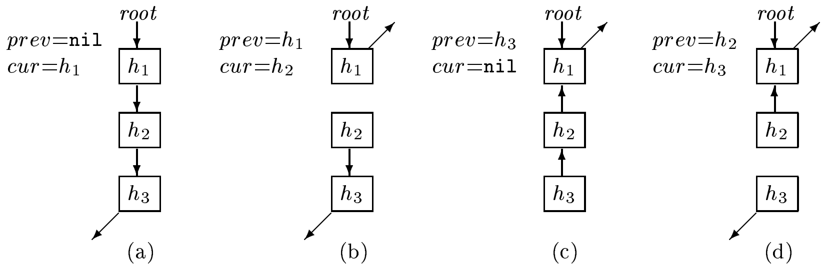

Figure 10.2: The process of pointer reversal at four points in the algorithm. Arrows pointing into "nowhere" represents `nil` pointers. We assume the elements to be records with a field n donoting the pointer to the next element.

10.3 Mark–Sweep Collection

A mark–sweep collector [34] finds the live objects by following pointers from a root–set. The objects reached are marked as live objects (i.e., detection of live values). The heap is then sweeped for dead objects, that is, space not occupied by marked objects is reclaimed (garbage reclamation) and inserted in a free list.

The mark–sweep principle must perform at least two traversals of the heap where the reclamation traversal (in its basic version) must touch every address in the heap. As with reference counting, the heap is fragmented in chunks of varying size containing either live or reclaimed objects. The fragmentation problem is also known from explicit allocation by the programmer. For instance, the functions `malloc()` and `free()` are normally implemented using one or more free lists containing chunks of reclaimed objects of varying size [32, page 185].

To build the free lists it is necessary to search for the unmarked objects. A scan through the entire heap may be necessary. The reclamation scan may be optimized by building a data structure (for instance a bitmap) saying where to find marked and unmarked objects while marking the live objects.

The marking phase involves considerable problems. A recursive algorithm visiting nodes in either breadth or depth first order may run out of stack space. An iterative algorithm pushing visited nodes on a stack for later revisiting may also run out of stack space.

A linear and constant space algorithm can be obtained by using *pointer reversal*. When traversing the heap, all branch points (i.e., objects with unvisited children) must be saved somewhere for later revisiting. Pointer reversal saves this information in the heap (i.e., in already allocated objects). Consider Figure 10.2 with an example heap containing a single linked list. While we traverse the list we reverse pointers such that we can go back again.

Pointer reversal on a single linked list can be implemented using three variables, *prev*, *cur* and *next*. Initially we have the situation in Figure 10.2(a) with $prev$ = `nil` and $cur = root$ (i.e., h_1). We then advance to the next element (h_2) by setting $next = cur \rightarrow n$, $cur \rightarrow n = prev$, $prev = cur$ and $cur = next$. If variable v points at object o then we let $v \rightarrow n$ be the field n in o containing the pointer out of o. The pointer for h_1 is now reversed, see Figure 10.2(b). We continue until the situation in Figure 10.2(c) is reached and it is now possible to go back to h_1 from h_3. We work back again by setting $next = prev \rightarrow n$, $prev \rightarrow n = cur$, $cur = prev$ and $prev = next$, see Figure 10.2(d). We stop when the original heap has been reconstructed (i.e., $prev$ = `nil`). Additional flags (besides the mark flag) are needed when objects contain more that one successor object but the method remains the same.[4]

Instead of storing mark bits in the headers of heap allocated objects we can use an external bitmap and reserve one bit for each object with a granularity corresponding to the smallest object allocated. On an 32 bits machine with the smallest object being a word the bitmap size is $\frac{1}{32} = 3.1\%$ of the heap space. On 64 bit architectures we have an overhead of 1.6%.

The sweep phase may be combined with allocation (lazy sweeping) [29]. The mark phase remains the same but we do not sweep the entire heap and build free lists. Instead we sweep the heap imcrementally each time an object is allocated by sweeping until an empty chunk of memory is found. If none is found we initiate the marking phase and sweeps the newly marked heap until a chunk of memory is found. The advantages are that no free lists must be managed and the overall garbage collection delays are smaller. We only need a global *sweep pointer* pointing at the address where the sweeper should start next time.

Live objects spread through out the heap may invoke paging more often than with a compacted heap (Section 10.4). The cache systems may also suffer from having more global pointers and with an exhausted heap we may initiate garbage collection more often because it is harder to accomodate the allocation requirements than if all reclaimed objects were merged into one larger chunk.

10.4 Mark-Compact Collection

A mark–compact collector solves the fragmentation problem of mark–sweep collectors and thereby makes allocation faster.

Instead of sweeping the address space, the live objects are compacted into one continuous block of memory. The rest of the memory area is then

[4] A fun exercise is to implement the simple pointer reversal algorithm using two variables *prev* and *cur* only and the identity $(A \oplus B) \oplus B \equiv A$, where $\oplus$ is the exclusive–or operation. The same identity can also be used to decrease the number of assignments in the algorithm sketched above from four to three [45].

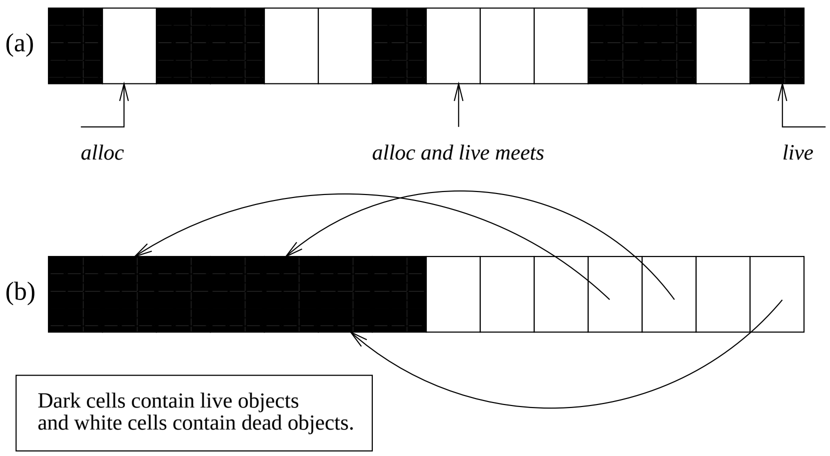

Figure 10.3: The two–finger method traverses the heap in both directions looking for free space and live objects to relocate at the same time. The initial heap is shown in (a) and (b) is after relocation. Note the relocation pointers inserted in the relocated objects in the top half of the heap. They are used when updating pointers in the bottom half (dark area) of the heap.

freed.

The obvious advantages with compaction compared to sweeping are that no free lists must be managed. Also, allocating objects is simpler as both large and small objects are allocated from the same continuous area of memory. Only a pointer has to be moved in order to allocate an object. The garbage collector is initiated when the allocation pointer exceeds the free memory area.

However, as with a mark–sweep collector, two or more parses over the address space are necessary. Many algorithms exists for compacting live objects and we review three of them. For a more detailed comparison of compacting algorithms including the calculation of symbolic time formulas consult [15].

The two–finger method [43] is simple and fast but in its basic version it is suitable for fixed sized objects only. The first phase marks all live data. The second phase scans the heap using two pointers, see Figure 10.3. The *alloc* pointer starts at the bottom of the heap and looks for free space for relocated objects. The *live* pointer starts at the top of the heap and looks for live objects to relocate. Objects are relocated by copying them into the space pointed at by *alloc*. After copying, the object pointed at by *live* is overwritten with the new address pointed at by *alloc*. The copying ends when the two pointers meet. Pointers in the bottom half are then updated by traversing the bottom half and changing all pointers pointing into the top half containing relocated objects, see Figure 10.3.

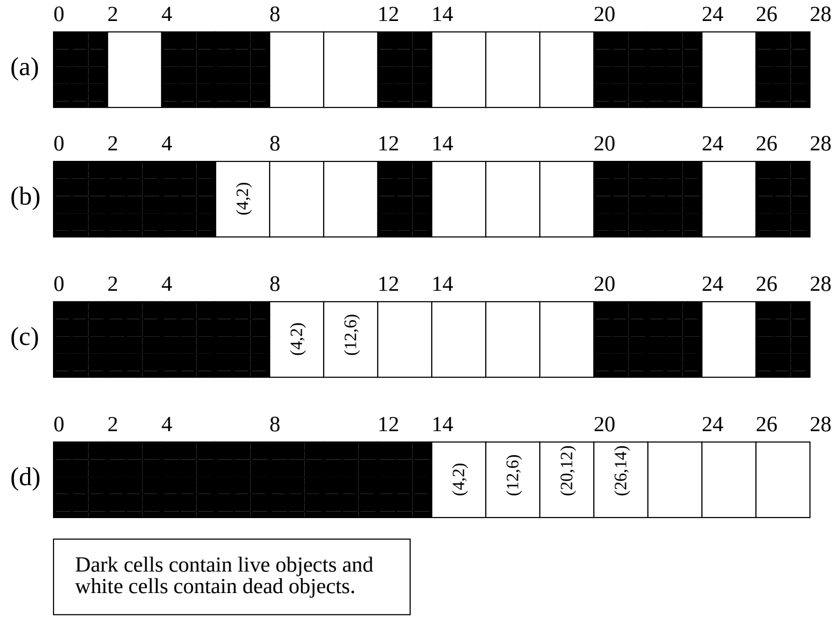

Figure 10.4: The break–table method inserts a break table into the free cells in the heap. For each object moved, the old address of the object together with the total amount of free space found so far is added to the table. The new address of the object at address 20 is 20-12=8.

It is possible, allthough cumbersome, to implement the two–finger method with variable sized objects where different sized compaction blocks are needed, that is, one *alloc* pointer for each object size is needed.

The next method uses a *break table* stored in the free address space in the heap [26]. The break table contains enough information about the relocated objects in order to update pointers pointing at relocated objects. First live objects are marked. The next phase moves objects and builds the break table that, for each object moved, contains the start address of the object before moving and the total size of free space found so far (i.e., $(addr, size)$), see Figure 10.4. There is always free space for the break table as long as one table entry (pair) is smaller than the smallest heap–allocated object [26]. As objects are moved toward lower addresses it is necessary to move the break table toward higher addresses. This can be done by rolling the table such that only fractions of the table is moved for each relocated object. When all objects are moved the table is sorted after address of object (i.e., $addr$).

Addresses pointing at relocated objects are resolved by scanning the compacted half of the heap and for each pointer p find the entries (a_1, s_1) and (a_2, s_2) in the break table such that $a_1 \leq p < a_2$. The new address for p is then $p - s_1$. For instance, in Figure 10.4 the new address of the relocated object at address 26 is 26-14=12.

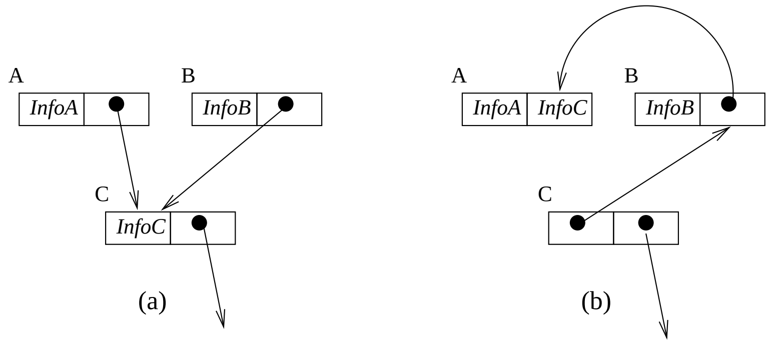

Figure 10.5: The original heap is shown in (a). After threading, the header field of object C (*infoC*) is moved into the pointer field of object A and a pointer to object B is stored in the header field in C. Updating the pointers to C in A and B is then done by following the pointer out from C.

The complexity of the algorithm is $O(n \log n)$ due to the sorting and searching into the break table (n is the size of the heap). However, in practice an almost linear complexity can be engineered [25].

The last compacting method we discuss is based on *threading* and is related to pointer reversal used for marking live objects [24, 31]. Suppose we have two objects A and B pointing at another object C, see Figure 10.5(a). If we move object C then the pointers into C must be updated with the new address of C. This can be done easily by first building the structure in Figure 10.5(b). Given object C we know which objects are pointing at C and can easily update the pointers with the new address of C without loss of information including the header field in object C. The update of some or all addresses is done before objects are moved to their new locations.

Besides marking live objects, a forward and backward scan of the heap are needed (i.e., complexity $O(n)$ where n is the size of the heap). The forward pass handles forward pointing pointers and the backward scan handles pointers pointing backwards, consult [31] for further detail. The method assumes that pointers only point at the header of other objects. This is always the case in the ML Kit. However, the method also assumes that a header field is large enough to contain an address (necessary for threading) and that the address is distinguishable from non addresses. This is not the case in the ML Kit. For instance, nullary value constructors occupy one word only even if tagging is enabled, see Chapter 12.

10.5 Copying Garbage Collection

A simple copying garbage collector (called stop and copy) [23] uses two address spaces, also called *semi spaces*. Only one semi space is used at the time (i.e., the *current* semi space) in which objects are allocated. Garbage collection is initiated when the current semi space is full. Live objects are traversed and copied from the current semi space (also called the *from–space*) into a new semi space, called the *to–space*. The copied objects are compacted in the to–space. After copying an object, the fields in the copied–from object are replaced with *forward pointers* pointing at the fields in the new object in to–space. We assume that forward pointers can be distinguished from other pointers and values. For each object traversed in from–space we know whether or not the object has already been copied and if an instance already exists in to–space then a forward pointer tells us the new address of the object. The live objects in from–space are copied exactly once. After all live objects in from–space has been traversed, we let to–space be the current semi space and the from–space is not used until garbage collection is initiated again. As pointers are updated when objects are copied into to–space, only one scan of the live objects is needed.

As with mark–compact collectors, there is no fragmention and allocation is done efficiently by moving an allocation pointer.

We allocate about twice the memory necessary for the running program which may be a problem for some memory demanding programs. The virtual memory systems used in modern computers [42, 28] partly solves the problem by swapping the non current semi space out of memory when memory demand is high. However, the time involved in paging and swapping may be significant descreasing the overall performance of simple stop and copy algorithms

10.5.1 Simple Stop and Copy

In this section we describe a simple stop and copy algorithm based on recursion [23]. Cycles are easily handled by marking each object with a visited flag. When an object is first met, the mark is set. The mark is then checked to see whether or not an object has already been visited. However, assuming we can distinguish forward pointers from other pointers and non pointers then the forward pointers gives us the same information as a mark flag. This only requires that all fields in an object are updated with their new forward pointers before decendants of the object are traversed. Consider the object

$\downarrow$	4	$\downarrow$

with two pointer fields and a scalar field.[5] When the object is first met we make a new instance of the object in to–space, and update all three fields in the from–space object with forward pointers into the fields of the new to–space object. We then recursively traverse the pointer objects in the to–space object. Note that if pointers always point at the start of an object then we only need one forward pointer for each object.

[5] We use $\downarrow$ to denote pointers and numbers to denote non pointer fields.

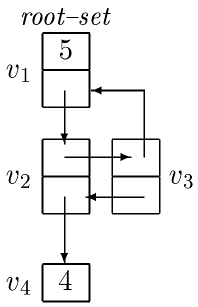

Figure 10.6: Pointer fields are shown with arrows and scalar fields with their scalar value. The root–set consists of the object v_1.

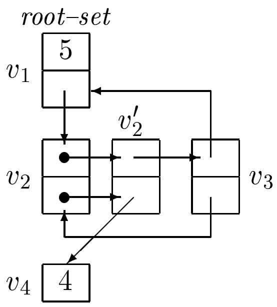

Figure 10.7: Forward pointers are shown as arrows with a filled circle. The two semi spaces to–space and from–space are not shown. We assume objects named with a "pling" (e.g. v_2') to be allocated in to–space and other objects to be allocated in from–space.

We assume that root–set objects are never moved and therefore always leave pointers into root–set objects unchanged. However, this depends on the application.

Consider the example heap in Figure 10.6. The root–set object contains one pointer field pointing at v_2. A copy of v_2 is put in to–space named v_2', and the two pointer fields in v_2 are changed into forward pointers, see Figure 10.7.

The first pointer field in v_2' is then traversed which first produces a new instance of v_3. Forward pointers are put in v_3, see Figure 10.8.

The first pointer field in v_3' points at the root–set object and is left unchanged. The second pointer field in v_3' points at a forward pointer in v_2 and is updated accordingly. We are now done with object v_3 and return the address of object v_3' so that the pointer object in v_2' can be updated, see Figure 10.9.

The second pointer field in v_2' is then traversed and a copy of v_4 is allocated in to–space (v_4'). The scalar field in v_4 is replaced by a forward pointer to v_4'. The address of v_4' is returned and the second pointer object in

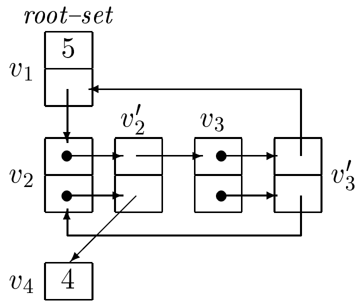

Figure 10.8: The object v_3 has been copied into to–space (v_3') and forward pointers are put in v_3. Note, that there are no changes to object v_2' yet.

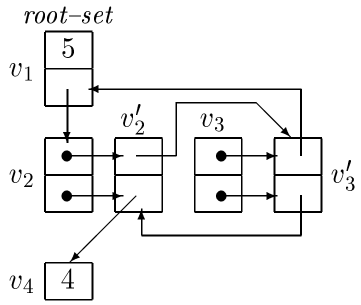

Figure 10.9: The first pointer field in v_2' is updated after return from object v_3. Note the change in the second pointer field in object v_3'.

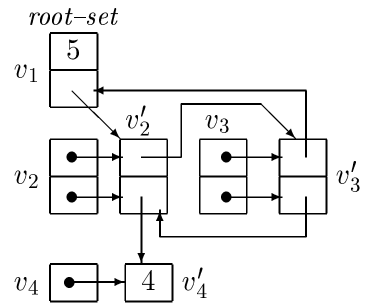

Figure 10.10: Objects with forward pointers may now be freed.

v_2' is updated to point at v_4'. We are now done with v_2 and the address of v_2' is returned to v_1 and the pointer field is updated to point at v_2', see Figure 10.10. The objects $v_2, \ldots, v_4$ residing in from–space may now be freed.

The following C function implements the algorithm sketched above with the restriction that pointers must point at the start of an object. The pointer *objPtr* points at a live object.

```
int gc(int *objPtr) {
  int *new, *i;
  if is_in_root_set(objPtr)
    return (int)objPtr;
  else {
    if is_forward_ptr(*objPtr)
      return clear_forward_ptr(*objPtr);
    else { /* We point at an unvisited object */
      new = copy_val(objPtr);
      for(i=first_obj_ptr(*new),i,i=next_obj_ptr(i))
        if is_ptr(*i)
          *i = gc(*i); /* Update field i. */
      return (int)new;
    }
  }
}
```

The function *copy_val*(*objPtr*) copies the object pointed at by *objPtr* and introduces forward pointers. The function *is_in_root_set*(*objPtr*) tests whether or not *objPtr* points at a root–set object and the function *is_forward_ptr*(**objPtr*) tests wheter or not the point at object is already copied. If it is copied then the forward pointer is returned after clearing the bit that identifies the pointer as a forward pointer. The loop runs through all fields in the copied object and calls *gc* recursively on all pointer fields.

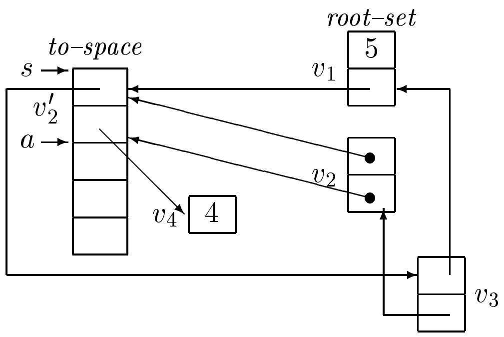

Figure 10.11: We loop over the to–space area to collect the live values. The scan pointer s points at the next field to collect.

Recursion is expensive because we perform a function call for each live pointer value in the heap. It takes time to setup a function frame with temporary values etc. A deep call tree may invoke a fatal error by exhausting the stack. Data structures, as large search trees, in Standard ML may create a deep call tree with the above method.

10.5.2 Cheney's Algorithm

The disadvantages of recursion are easily avoided using the to–space area to record the values that have been copied but not collected yet [14]. We do a breadth first traversal instead of a depth first traversal as the recursive algorithm above.

The algorithm uses an allocation pointer a and a scan pointer s both pointing into the to–space area. The allocation pointer points at the address where the next relocated object is stored. The scan pointer points at the current object (or field) being garbage collected. Each time an object (or field) is collected the scan pointer is adjusted to point at the neighbor object (or field). The algorithm ends when $s = a$.

Consider the example heap in Figure 10.6 again. The root–set object v_1 is not copied into to–space. The first value we copy is v_2 and then we scan the to–space area and collect all other live values. As in the recursive algorithm we insert forward pointers when copying v_2. We also update the pointer field in v_1 to point at v'_2, see Figure 10.11

The next field to collect, pointed at by s, is an object pointer pointing at v_3. The value v_3 is copied into to–space, see Figure 10.12.

The second object pointer in v_2 points at a scalar object that is copied into to–space. The first pointer object in v'_3 points at the root–set value and is left unchanged. The second object pointer in v'_3 points at a forward pointer and is updated accordingly. Now the scan pointer points at the scalar object which is left unchanged. The algorithm stops when $a = s$, see

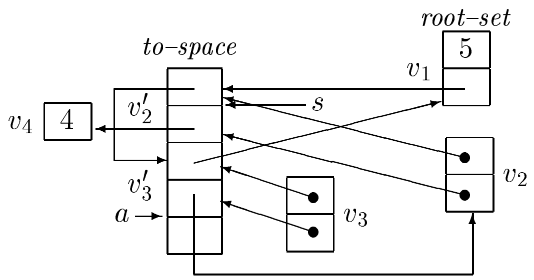

Figure 10.12: The first field in v_2' is now collected and afterwards s points at the second object pointer in v_2'.

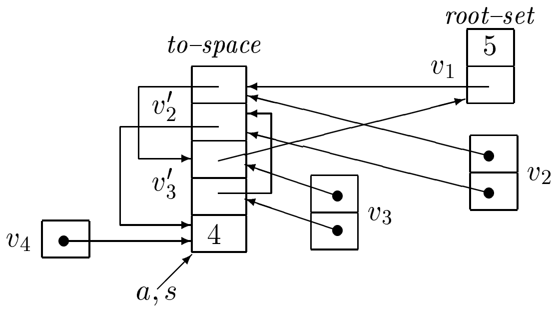

Figure 10.13: The objects v_2, v_3 and v_4 may now be freed.

Figure 10.13.

The algorithm is fast because it is driven by a simple loop. First we loop through the root–set values and copy the initial objects into to–space. Then we loop over the to–space area until all objects have been collected; there are no other external structure that has to be updated. Note, that we can not deallocate objects containing forward pointers in from–space before all values in to–space have been collected. The overall algorithm is as follows:

1. $a = \mathit{first\_addr\_in\_to\_space}$
2. $\forall$ *objPtr* $\in$ *root_set* : `*`*objPtr* `=` *gc*(`*`*objPtr*)`;`
3. `for`(*s*=*first_addr_in_to_space*`;`*s*`<`*a*`;`*s*`++`) `if` (*is_ptr*(`*`*s*)) `*`*s* `=` *gc*(`*`*s*)`;`
4. make to–space the current semi space.

The function *gc* is slightly different from the recursive algorithm.

```
int gc(int *objPtr) {
   if is_in_root_set(objPtr) then
```

```
        return objPtr;
    else {
        if is_forward_ptr(*objPtr) then
            return clear_forward_ptr(*objPtr);
        else /* We point at an unvisited value */
            return copy_val(objPtr);
    }
}
```

The funtion *copy_val* ajusts the allocation pointer.

10.5.3 Generational Garbage Collection

A copying garbage collector is effective on small objects with a short life time; an object with a life range that starts and ends between two successive garbage collections is collected for free. However, large objects with a longer life range are expensive because they are copied at each collection. Especially global data are copied over and over again.

Generational garbage collection solves this problem in the case that most objects allocated have a short life range and the number of pointers from old objects into newer objects are small. An older cell can point to a newer cell if the older cell is modified after it is created. This is only possible through references in Standard ML.

The heap is divided into a number of generations, $G_1, \ldots, G_n$ and the newest generation G_1 is garbage collected most often. As objects survive garbage collections they get older and are eventually copied into older generations that are collected less often. Given a root–set a naive algorithm must traverse all live objects including objects in older generations in order to find all live objects in the younger generations. It is better if we can limit the search to include younger generations only. This is possible if there are no pointers from an older generation into a younger generation. Each time we have a pointer into an older generation we then stop traversing. However, with references in Standard ML we may have references from older generations into younger generations, but they are relatively rare. In SML/NJ [5] the problem is solved by maintaining a list of addresses of cells in older generations which have been updated after they were created. Letting the list be part of the root–set we are sure to get all live data in the younger generations and still only traverse the younger generations. This is only efficient because the list in practice is small.

The Caml Light garbage collector [44] is generational with two generations and hybrid using a stop and copy collection of the young generation and incremental mark–sweep collection of the old generation. The mark–sweep collector reduces the time used on copying because older objects are not moved.

10.6 Comparing The Algorithms

We have chosen the criterias below as a basis for comparing the basic garbage collection algorithms described in this chapter.

User interaction tolerance: for how long time and how often is the user program interrupted.

Reference counting gives small but many user interactions which may give the impression of a smoother program execution compared to the tracing algorithms with fewer but longer interactions. It is easier to have reference counting fulfill some real–time requirements than a tracing algorithm. The lazy sweeping method [29] improves the basic mark–sweep method but still has to trace all live data. Both reference counting and the tracing algorithms can be made incremental but their implementations and especially tracing algorithms tend to be complicated.

We do not consider real–time properties to be important for the garbage collector in the ML Kit. Garbage collection is an add on only and region inference is the main memory allocation strategy which handles most real–time requirements. Small real–time applications should be developed using region inference only.

Precision criteria: how precise is the algorithm in reclaiming dead objects.

Reference counting does not reclaim cyclic structures. All tracing algorithms reclaims all but live data iff a precise root–set can be calculated. We want the garbage collector in the ML Kit to reclaim all garbage based on the root–set given, that is, we want cyclic structures to be reclaimed properly.

Aggressiveness: for how long time are objects allocated after they have died.

Reference counting is aggressive because an object is reclaimed as soon as the count reaches zero. Tracing algorithms wait until they are triggered by, for instance, an exhausted heap. Region inference shares the eagerness of reference counting where entire regions are relaimed as soon as they are inferred to be dead. The only difference is that reference counting is dynamic and region inference is static. Region inference already recycles the heap efficiently [10, 49]. We expect region inference to recycle most of the allocated memory such that the rate in which memory is filled with dead objects is low.

Root–set computation: the tracing algorithms need to compute a root–set. It is possible to compute a root–set in the ML Kit, see Chapter 13.

Tag information: what kind of tagging is necessary to implement the algorithm.

Reference counting requires several bits for the reference counts opposed to the tracing algorithms which need a single bit only (for each word) to differentiate between scalars and pointers.[6] This is indeed a serious drawback of reference counting in our framework because we have a large number of objects. The problem tends to disappear when the number of objects gets smaller as for instance in file systems where the number of open files is easily handled using reference counting. A file descriptor is reclaimed when the reference count reaches zero.

Overhead/performance: what kind of overhead is imposed by the algorithm and what is the asymptotic performance complexity. We note that complexity formulas involving constants must be read carefully because, the size of the constants may be more important than the actual complexity. This is indeed the case with garbage collection algorithms. All algorithms we have presented are linear (in the size of live data or the heap) except for the break table method with complexity $n \log n$, where n is the size of the heap.

Say that allocating an object gives overhead a, then if it is done often it is significant whether the constant is large or small.

Let H be the size of the heap and let U be the amount of allocated (used) memory, (i.e., $U \leq H$). Let N be the number of times an object is allocated (and deallocated).

The time used on reference counting is $t_{RC} = c_1 N$, where c_1 is time used on allocation and deallocation including bookkeeping of reference counts and free lists.

The mark–sweep method uses time $t_{MS} = c_2 U + c_3 H$ where c_2 is the time used on allocation and tracing the heap and c_3 is the time used on sweeping. Using lazy sweeping the term $c_3 H$ dissapears from t_{MS} and the overhead c_2 also includes the searching for a chunk of memory. With lazy sweeping the complexity depends on live objects only and not the size of the heap, as is the case for mark–compact collectors.

The mark–compact method uses time $t_{MC} = c_4 H \log c_5 U$ using the break table method and $t_{MC} = c_6 H$ using threading. The constant c_4 includes time to move objects, update pointers and roll the break table. The constant c_5 is the overhead in sorting and searching the break table. The constant c_6 includes the work of threading and unthreading pointers.

---

[6] The presentation is simplified because a single bit is not always enough to differentiate between scalars and non scalars, see Chapter 12. But it is still, by far, better than reference counts.

The copy method uses time $t_C = c_7 U$ where c_7 includes the time used to copy live objects and update forward pointers.

Allocation is more expensive for mark–sweep collectors based on free lists than for compacting collectors. This is important in Standard ML where the allocation frequency is high and the average life range small. The SML/NJ'0.93 compiler reclaims, in average, 98.7% of the heap at each garbage collection [5, page 206]. The compating collectors use time proportional to the heap size and not the size of live data. This favors the copying method except in the case where we have large objects with longer life ranges. This is however not the average case in Standard ML programs where most objects allocated are tupples and closures.

Implementation: considering the region heap (being a linked list of region pages) it is difficult to see how an efficient allocation strategy is based on free lists. We then need a free list for each allocated region.

Region inference makes most regions contain objects of the same size. However, some regions do not and the two–finger method is therefore not suitable.

The threading algorithm assumes that all objects have a pointer sized header field. We believe this to be an expensive restriction especially due to the large amount of constructors that can be implemented using one word only.

The break table method assumes the smallest heap allocated object to be of size two words. This is not the case in the ML Kit where for instance a nullary value constructor occupies one word only. It is still possible to use the method with the cost that not all garbage may be removed [25]. The region heap also complicates the break table method significantly because the break table must be rolled between different non continuous region pages.

The simple recursive copy method works with the region heap as long as we can allocate into the region heap. Let each region have two lists of region pages, a from–list and a to–list. After garbage collection we free all from–lists and, for each region, make the to–list the current list of region pages. This also minimizes the overhead of data used. Instead of having an overhead of one semi space the overhead is the number of region pages used to hold live data.

We believe the most promising choice for the ML Kit is a copying garbage collector. It is simple to implement, works with regions and only needs a minimum amount of tagging. We refine the Cheney method to work with regions in Chapter 11.

Chapter 11

Garbage Collection of Regions.

In Chapter 10 we found the most promising garbage collection method for regions to be simple non generational copying garbage collection. In this chapter we show how the copying priciple developed by Cheney [14] is extended to work with regions. We also describe extensions to region descriptors and region pages.

11.1 Algorithm Using Recursion

It is easy to refine the simple recursive copying garbage collector from Section 10.5.1 to work with regions. The algorithm does not put restrictions on the number of to–spaces and from–spaces.

We extend the infinite region descriptors to hold an additional list of region pages so that each region has its own from–space and to–space. An infinite region descriptor then contains the following fields (consult [22] for more information about each field in the descriptor):

1. allocation pointer (a), which points at the next available address in the region page currently being allocated into. When garbage collecting, a points into to–space, and otherwise a points into from–space.
2. pointer to the first region page (fp) in from–space.
3. pointer to the end of the region page currently being allocated into. This is called the border pointer, b. The pointer b points at the border of to–space when garbage collecting and otherwise at the border of from–space.
4. pointer to the previous region descriptor on the region stack, p.
5. pointer to the first region page in to–space, (fp').

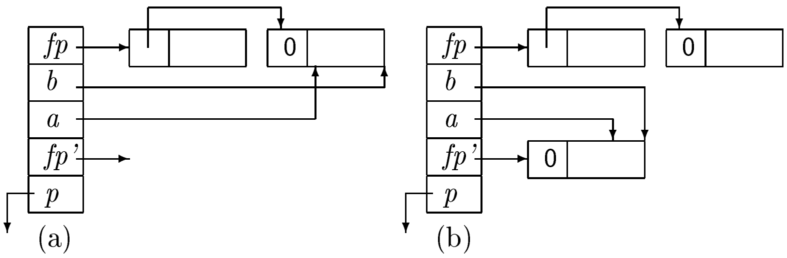

Figure 11.1: Part (a) shows an example region descriptor with from–space (*fp*) and to–space (*fp'*) at a time where garbage collection is not performed. Values are allocated in from–space. The pointer *fp'* may point at anything. Part (b) shows what happens during garbage collection. The allocation and border pointers now work on the to–space. After garbage collection the from–space region pages are freed and the to–space region pages are turned into from–space region pages.

We note that it is not necessary to have two instances of the allocation and border pointers because at any time we either allocate into from–space or into to–space. The region pages in from–space contain garbage after a garbage collection phase and are moved to the free list for recycling. The pointer *fp* is then set to point at to–space, that is, to–space is then the current space in which objects are allocated, see Figure 11.1.

11.2 Cheney's Algorithm and Regions

Cheney's algorithm from Section 10.5.2 does not, as is, work with regions because we do not have a single to–space but an unbounded number of to–spaces, that is, an unbounded number of scan and allocation pointers. The algorithm ends when all scan and allocation pointers are equal. The stop criteria is then a fixed point criteria:

$$\forall r \in \mathit{Reg} : r \rightarrow a = r \rightarrow s,$$

where Reg is the set of infinite region descriptors on the region stack, $r \rightarrow a$ is the allocation pointer and $r \rightarrow s$ is the scan pointer in region r. A naive implementation of the fixed point criteria looping through all region descriptors on the stack is indeed not an efficient solution. There may only be a small number of regions not fulfilling the stop criteria.

Consider Figure 11.2 with a data structure that imposes an interesting behavior on Cheney's algorithm extended to regions. At any time during garbage collection, at most two regions will contain one copied and non collected value. With the naive implementation of the fixed point criteria the region stack is traversed for almost every copied value, see Figure 11.3.

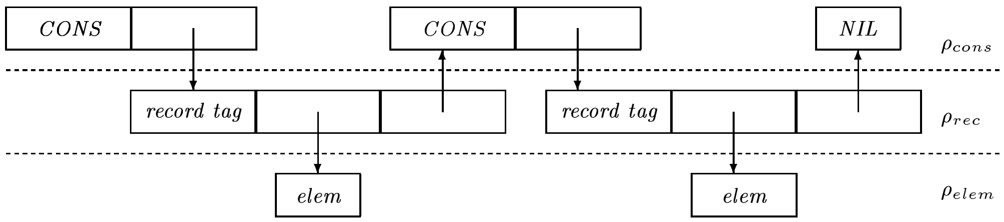

Figure 11.2: The figure shows an example data structure allocated in three regions. It is a possible implementation of lists where cons and nil cells are allocated in ρ_{cons}, records in ρ_{rec} and elements in ρ_{elem}. We note that lists are implemented more efficiently in the ML Kit and that this figure serves only as an illustration.

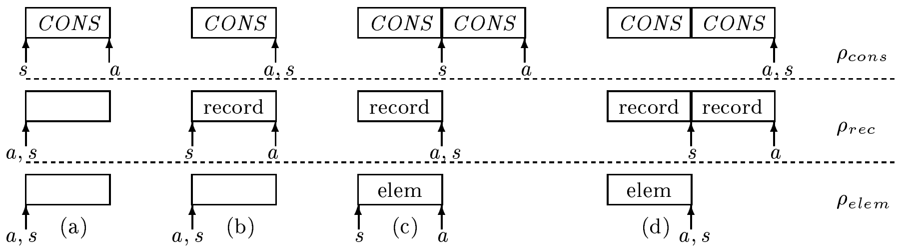

Figure 11.3: It may be necessary to loop through the region stack many times because only a fraction of the set of live values in a region is copied into to–space each time the region is collected. Scanning region ρ_{cons} gives one new value in ρ_{rec}, (a)→(b). Scanning region ρ_{rec} gives one new value in ρ_{cons} and ρ_{elem}, (b)→(c). Scanning ρ_{elem} does not introduce new values but scanning ρ_{cons} introduces one new value in ρ_{rec}, (d).

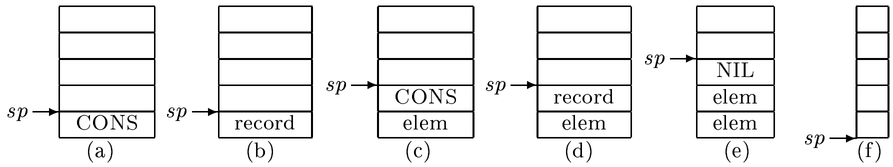

Figure 11.4: The figure shows seven snapshots of the scan stack (a) – (f) when collecting the data structure in Figure 11.2. The maximal height of the stack is $n+1$ where n is the number of elements in the list.

11.2.1 A Stack of Values

The problem with more than one to–space is avoided using a *scan stack* containing pointers to values copied into to–space areas but not traversed yet. Each time a value is copied into to–space, a pointer to the value is pushed on the scan stack.

It is not necessary to have a scan pointer in each region descriptor, and not even on the scan stack. We simply pop the top pointer and collect the value it points at. We are done when the scan stack is empty.

There is a small overhead for each copied value in both time and space compared to Cheney's algorithm, but it solves the problem of having many to–spaces. There is no recursion and the algorithm is not vulnerable to recursive data structures. However, many objects may be pushed on the scan stack at the same time. This extra space usage is a fatal problem. Figure 11.4 shows the scan stack while collecting the data structure in Figure 11.2.

11.2.2 A Stack of Regions

Instead of storing pointers to values in the scan stack we can store pointers to region descriptors containing non collected values [47].

Consider the region descriptor in Section 11.1, Figure 11.1. We extend the region descriptor with a *region status*:

NONE if there are no non collected values in to–space.

SOME(s) if there are non collected values in to–space. The pointer s then points at the next non collected value in to–space (i.e., s is the scan pointer).

Each time a value (v) is copied into to–space (v'), the region status is checked, and if NONE, then the status is changed into SOME(a) where a is the allocation pointer pointing at the address where v' is stored. A pointer to the region is pushed onto the scan stack. If the region status is

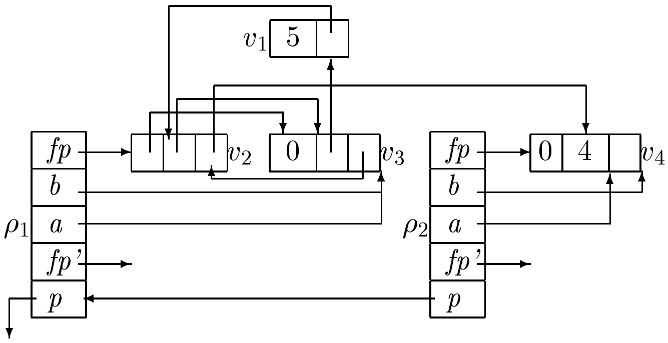

Figure 11.5: We have drawn the values v_2, v_3 and v_4 as allocated in the two regions. The region ρ_1 has two region pages allocated with a value in each one and a pointer to the next region page. A null pointer is written 0. We have not drawn the backward pointer from a region page to the region descriptor. Only one region page is necessary in region ρ_2. The value v_1 is allocated on the stack and is in the root–set.

SOME(*ptr*) then nothing is done, because the value v' is automatically collected next time the pointer to the region descriptor is popped from the scan stack. The region status of the region currently being collected is changed to NONE when we are done with the region. This prevents the region to be pushed on the scan stack while it is actually being collected.

A minor optimization is to push the scan pointer on the scan stack directly (instead of a pointer to the region) and then only manage a status bit in the region descriptor. We use Cheney's algorithm locally on each region and are done when the scan stack is empty.

The algorithm is best illustrated on our running example in Figure 10.6 on page 179. Assume we have two regions, ρ_1 and ρ_2, where value v_2 and v_3 are allocated in ρ_1 and value v_4 is allocated in ρ_2, see Figure 11.5.

To initialize the scan stack we collect all values in the root–set (i.e., v_1). This may cause values allocated in regions, to be copied into to–spaces. After initialization, all regions containing values in to–spaces will have the region status SOME(*ptr*) and are pushed on the scan stack.

Unfortunately, we must check the region status each time we copy a value. This is only feasible if we, given a pointer to a copied value inside a region page, can find the region descriptor holding the region status. This is done by having all region pages of fixed size and aligned. Given a pointer (*val_ptr*) into a region page, the start of the region page is found by masking out the offset (*val_off*) of the pointer relative to the base address (*base_ptr*) of the region page.

Say that each region page has size 1 Kb. and is aligned at 1 Kb. addresses. The first 10 bits of the base pointer to the region page are then zero. Every pointer into the region page then consists of the base pointer

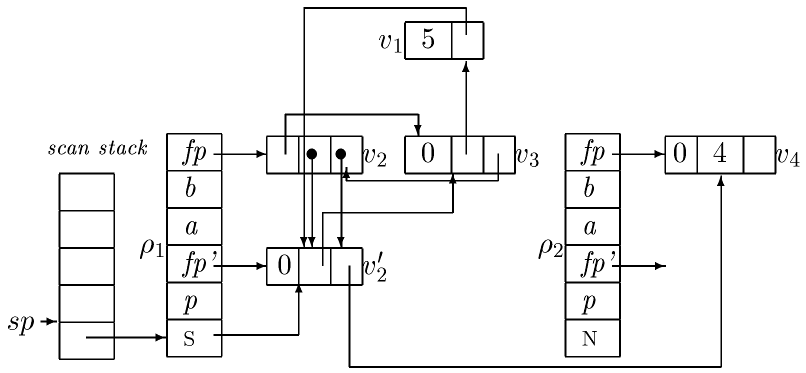

Figure 11.6: We reduce pointer spaghetti by not drawing the border, allocation and previous region descriptor pointers. The status SOME(ptr) is drawn as S. Likewise, the status NONE is drawn N. The scan stack contains region ρ_1. Forward pointers are drawn with a bullet.

plus an offset which is held in the first 10 bits:

$$base\_ptr = val\_ptr \texttt{ AND } \boxed{0_{31} \cdots 0_{10} 1_9 \cdots 1_0}$$

and

$$val\_ptr = base\_ptr \texttt{ OR } val\_off$$

The region descriptor is located by storing a pointer in each region page descriptor pointing back to the region descriptor. Given a pointer into a region page, the corresponding region descriptor is found by a binary `AND` operation and one dereferencing.

Consider Figure 11.5. After initialization the value v_2 is copied into to–space in region ρ_1, see Figure 11.6.

We pop the first pointer off the scan stack that points at region ρ_1. We therefore collect value v_2' and the two values v_3 and v_4 are copied. While scanning the to–space of region ρ_1, we update the region status to point at the next non collected value. The region ρ_1 is not pushed on the scan stack when value v_3 is copied because v_3' is collected after value v_2'. That is, we continue to scan to–space of region ρ_1 until no non collected values are available.

After collecting value v_3' we are done with region ρ_1 and ρ_2 is on the scan stack, see Figure 11.7.

It is actually not necessary to change the status of region ρ_2 when copying value v_4 because the value only contains scalars.

The maximal depth of the scan stack is limited by the number of region descriptors on the region stack; at any time, at most one pointer to each region descriptor is on the scan stack. We can use the machine stack as a scan–stack and the likelihood that the machine stack is exhausted during

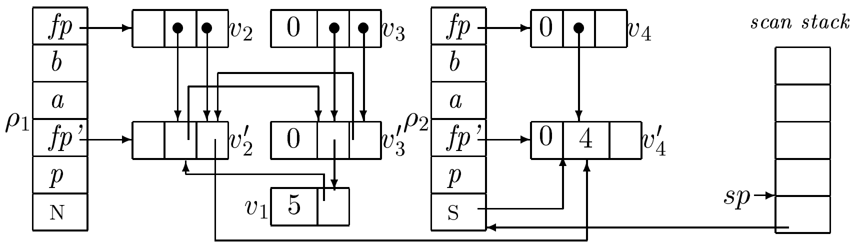

Figure 11.7: Several pointers are not shown to reduce pointer spaghetti. A pointer to region ρ_2 is now on the scan stack. No more values are collected because value v_4' is a scalar.

garbage collection is no different from the likelihood that the machine stack is exhausted when not using garbage collection except for a constant factor less than two (i.e., the scan stack occupies less space than the function frames that hold the infinite region descriptors). It is therefore feasible to assume that the machine stack can hold the scan stack.[1]

Consider the data structure in Figure 11.2. We saw in Section 11.2 that the naive implementation of the fixed point criteria behaved poorly on this data structure. We have the same behaviour with the above algorithm in that each region will change status for almost every value copied. However, this only implies a small constant overhead and not a linear overhead in the size of the scan stack. We do not expect this overhead to be significant.

The complexity of the algorithm is $t_C = cU$ where U is the amount of live data, see Section 10.6. The constant c includes the time used to copy live objects, update forward pointers, handling region status and the scan stack. The algorithm is slower than the original Cheney algorithm but with a constant factor only.

11.3 A Revised Region and Region Page Descriptor

We have already discussed the changes necessary to the region descriptor in order to adopt the Cheney algorithm for regions. We introduced a pointer *fp'* pointing at to–space. However, notice that while garbage collecting a region we never allocate into from–space. It is therefore not necessary to have the from–space linked to the region descriptor. We just have to make sure that from–spaces are not freed until after garbage collection.

Before garbage collection, we traverse the region stack and move the region pages in all from–space areas (pointed at by *fp*) into one linked list

[1] We do not use the machine stack in the initial implementation but a fixed sized array.

of region pages. We reserve a global pointer *from_space* to point at the first region page. The pointer *fp* in each region descriptor is then initialized to point at a fresh region page from the free list, now being the to–space. Garbage collection is performed as described above and when all values have been collected the region pages pointed at by *from_space* are garbage and inserted into the free list.

While collection is in progress, we allocate new region pages from the free list and not from the list pointed at by *from_space*. This is not more expensive than having both *fp* and *fp'*, because after collection we should append all region pages pointed at by *fp* to the free list anyway. With the revised method, we remove the pages from the region descriptors before collection, but wait to after collection before we insert them in the free list.

We implement the region status (NONE or SOME) with a single bit in one of the address fields in the region descriptor. The actual scan pointer is pushed on the scan stack instead of a pointer to the region descriptor. We then conclude that no additional fields are added to the region descriptor in order to implement copying garbage collection.

Each time an object is copied into to–space we need access to the region descriptor to allocate a new object and to check the region status. The region descriptor is found by storing an additional pointer in the region page descriptor pointing at the region descriptor. The region page descriptor then contains two pointers:

1. a pointer to the next region page, n
2. a pointer pointing at the region descriptor, rd

11.4 The Garbage Collection Algorithm

In this section we present the garbage collection algorithm in pseudo code. First we traverse the region stack and move all region pages into a global list of from–space region pages. We allocate a fresh region page for each region. The variable *topRegion* points at the infinite region descriptor at top of the region stack. The global variables *fromSpaceBegin* and *fromSpaceEnd* point at the first and last from–space region page. There is always at least one region allocated (i.e., *topRegion* never points at `NULL`).

```
fromSpaceBegin = NULL;
fromSpaceEnd = addr_of_last_region_page(topRegion);
for (rd=topRegion;rd != NULL;rd=prev_region(rd)) {
    lastRegionPage = get_last_region_page(rd);
    next_ptr(lastRegionPage) = fromSpaceBegin;
    fromSpaceBegin = addr_of_first_region_page(rd);
    first_region_page_ptr(rd) = fresh_region_page();
}
```

We then garbage collect all objects in the root–set.

$$\forall objPtr \in root\_set : *objPtr = gc\_obj(*objPtr)$$

where the function *gc_obj* garbage collect an object.

```
int gc_obj(int obj) {
   if (is_in_root_set(obj) || is_scalar(obj)) then
      return obj;
   else {
      /* We know obj is a pointer */
      objPtr = (int *) obj;
      if is_forward_ptr(*objPtr) then
         return clear_forward_ptr(*objPtr);
      else /* We point at an unvisited value */
         return (int) copy_obj(objPtr);
   }
}
```

The function *copy_obj* copies the value from a from–space region page into the to–space area of the region where the object belongs. An integer (tested with *is_scalar*) is unboxed and is returned as is. The region status is checked and the region pushed on the scan stack if necessary.

```
int *copy_obj(int *objPtr) {
   rd = get_region_descriptor(objPtr);
   new_obj_ptr = copy_val(objPtr,rd);
   if status(rd) = NONE {
      push_scan_stack(new_obj_ptr);
      set_status_SOME(rd);
   }
   return new_obj_ptr;
}
```

The function *get_region_descriptor* returns a pointer to the region descriptor in which the object is allocated. The region descriptor is found as shown in Section 11.2.2; region pages contain a pointer back to the region descriptor. The function *copy_val* allocates space and copies the object into to–space. A forward pointer is inserted in `*objPtr` pointing at the new object. The allocation pointer in the region descriptor is adjusted.

After copying all root–set objects we iterate until the scan stack is empty.

```
while not_empty_scan_stack() {
   scanPtr = pop_scan_stack();
   rd = get_region_descriptor(scanPtr);
   while scanPtr != get_alloc_ptr(rd) {
```

$$
\begin{array}{l}
\quad\quad \forall\, \mathit{field} \in \mathit{scanPtr} : *\mathit{field} = \mathit{gc\_obj}(*\text{field}); \\
\quad\quad \mathit{scanPtr} = \mathit{scanPtr} + \mathit{size\_of\_obj}(\mathit{scanPtr}); \\
\quad \}; \\
\quad \mathit{set\_status\_NONE}(\mathit{rd}); \\
\}
\end{array}
$$

We are done garbage collecting when the scan stack is empty. The from–space region pages are now inserted into the free list.

$$
\begin{array}{l}
\mathit{next\_ptr}(\mathit{fromSpaceEnd}) = \mathit{freeList}; \\
\mathit{freeList} = \mathit{fromSpaceBegin};
\end{array}
$$

The variable *freeList* is a pointer to the first region page in the free list.

11.5 Finite Regions

So far we have discussed garbage collection of infinite regions only and found a promising algorithm. Unfortunately, finite regions complicate matters. We have three kinds of objects in the ML Kit.

1. objects allocated in infinite regions. These objects are traversed and copied by the revised Cheney algorithm.

2. objects allocated in finite regions residing in activation records on the machine stack. These objects are traversed but not copied (i.e., they may contain pointers to other objects).

3. constants in the data area of the program binary. They are neither traversed nor copied. It is not necessary to traverse a constant because it never points at a heap allocated object.

How do we differentiate between the objects and how do we treat objects in finite regions? Constants are easily recognized because we use a bit in the tag of objects to denote constants. The result of following a pointer to a constant is the pointer itself and no further processing.

A pointer p pointing at an object o in a finite region is recognized by a range check on the machine stack boundaries (i.e., $\mathit{stackBot} < p < \mathit{stackTop}$). An object in a finite region may be in two states:

1. never traversed, that is, p is the first pointer found pointing at o.

2. already traversed and updated, that is, p is not the first pointer found pointing at o.

In the first case we can either recursively follow the fields in o and update o or save the address of o on a scan stack for later processing. In the second case we may not follow the pointers in o and update o because they have already been updated.

We chose to put the object on a scan stack for later processing such that we still have a non recursive algorithm. The size of the scan stack is potentially unbounded. We have not found this bound to be a problem in our implementation. Chains of pointers from finite regions to finite regions, not intervened by a pointer into an infinite region, do not seem to be long. Every time we have a pointer to an object in an infinite region then the region is pushed on the scan stack (if not already there). With two scan stacks, one for finite regions and one for infinite regions, then by always popping from the stack with finite regions (before popping from the stack with infinite regions) we keep the size of the two stacks small.[2]

To see wheter a finite region has already been traversed we need a mark on the region. It is possible, however difficult, to mark a finite region when it has been processed. The problem is that we must reset the marks when we are done garbage collecting. This requires information about where the finite regions reside on the machine stack. One possibility is to record it in the frame map at each application point (see Chapter 13).

If we, given a pointer p into an infinite region, can figure out wheter p points into from–space or into to–space then we know whether p has already been processed. If p points into to–space then p has already been pocessed. We implement this by marking all region pages in from–space when from–space is build and reset the marks when from–space is merged with the free list.

We do not have these problems in the Cheyney algorithm. Wheter an object has been traversed or not is recognized by the presence or absence of a forward pointer. Also, the scan pointer never passes an object twice in to–space.

11.5.1 Recursive Data Structures

If we have two pointers pointing at an object o in a finite region then we traverse o twice because there is no tag on o saying that o has already been traversed. However, all fields in o pointing into to–space are never followed. What if a pointer p in o points at another finite region? Then we follow p because p is the same whether it has been followed or not (i.e., the point at object is not moved). Our algorithm may therefore follow pointers out of objects in finite regions more than once which unfortunately take time. We have not had time to measure this overhead.

[2]We only use one scan stack in the implementation and even in that case we have never had a problem with the size of the scan stack.

What happens if we have a cyclic data structure involving finite regions only? Then we may loop infinitely because the regions are continously pushed on the scan stack; we may follow the same pointer from one finite region into another finite region many times. However, if the cyclic data structure involves an infinite region then we are safe; no object in an infinite region is traversed more than once.

The region inference rules and the multiplicity analysis (Section 2.3) do not allow cyclic data structures involving finite regions only. To create a cyclic data structure we must use references.

The region inference rules require that the region(s) containing the value that a reference points at are always the same for each reference. Consider a refernece `val` a `= ref` v where v must reside in region ρ. If we update a with a `:=` v' then v and v' must both reside in region ρ. The multiplicity analysis then gives ρ multiplicity ∞ and v, v' are allocated in an infinite region.

We must always have at least one update in order to create a cycle and we conclude that a cycle always involves at least one infinite region. We note that this argument has not been proved.

We consider changing the implementation to use marks on finite regions such that all objects are traversed only once. The above unproved argument is then insignificant. This requires the frame map to be extended with information about the placement of finite regions in the function frames because the marks must be reset after a garbage collection.

11.6 Garbage Collect a Few Regions Only

Experience has shown that most space leaks are localized around a few global (i.e., older) regions, that is, a few regions contain most of the dead values. Inspired by generational garbage collection (see Section 10.5.3) it seems resonable to localise garbage collection to a few global regions in the same way that generational garbage collection concentrates on the younger generations.

Generational garbage collection uses the property that generations contain objects of approximately the same age. This is not the case in region inference. Global regions may contain old and new objects. We have lots of pointers from younger regions into older regions.

Figure 11.8 illustrates the problem of localising the search for live objects. We want to concentrate on the older regions (drawn black in the figure) but must traverse the younger regions because the yonger regions contain pointers into the older regions.

It may be possible to limit the search by marking region pages as dirty if they contain a pointer into an older region that is included in the set of regions to reclaim. Then the search for live objects can stop at region pages

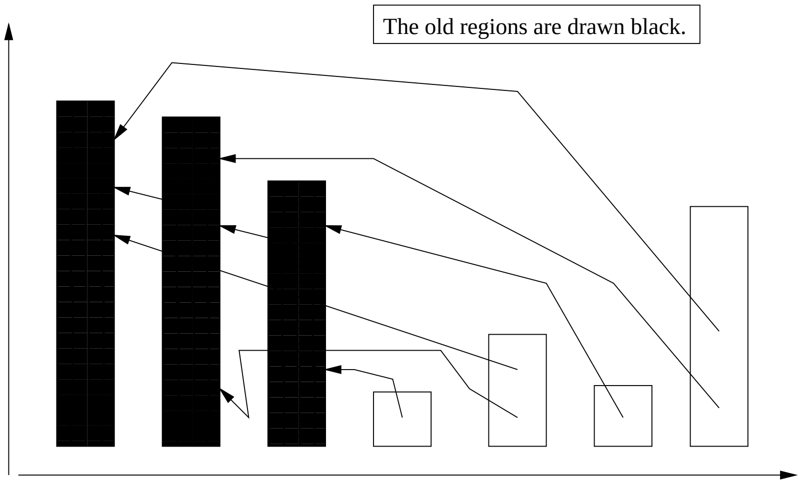

Figure 11.8: Pointers can only go from newer region to older regions. We have darkened the older regions being the regions that we would like to concentrate on.

not marked dirty. We have not investigated this further.

We may also have pointers from older regions into younger regions but it happens rarely and the garbage collector cannot handle them properly. Consult [51, page 10] for an example. It is possible to detect such a pointer at compile time and the compiler refuses to compile a program with such a pointer when garbage collection is enabled. The problem is that such a pointer is turned into a dangling reference when the younger region is deallocated and the garbage collector cannot handle dangling references.

A generational garbage collector can limit the search to include younger generations only. Only a small set of pointers point from old generations into young generations. In SML/NJ'0.93, a set of addresses of cells in older generations, containing pointers into younger generations, is maintained (*oldPtrs*). If a pointer p, pointing into an older generation, points at an address included in *oldPtrs* then p is followed; otherwise p is not followed. This is efficient because the set *oldPtrs* is small.

It does not seem possible to reduce the set of live objects to traverse in order to garbage collect global regions only because many pointers exists from younger regions pointing into older regions (i.e., many region pages are dirty). However, it may still be a considerable saving to reclaim objects in a few regions only, that is, traverse all live objects but only copy objects in the regions that contain most garbage. The regions to garbage collect may be chosen with one of the heuristics:

1. only garbage collect the n regions allocated in the bottom of the region stack, that is, the n oldest regions. Experience shows that they contain most space leaks and therefore also most dead objects.

2. only garbage collect regions with more than m region pages allocated. This concentrates garbage collection to regions containing lots of data and therefore also the likelihood that some of the data is dead. However, it is not necessarily the case. Young regions may contain large data structures as the result of a local computation (e.g., a large syntax tree). An extra field in the region descriptor may be necessary to store the number of region pages currently allocated.

3. only garbage collect regions which have grown with a certain amount since the last garbage collection. This may require an extra field in the region descriptor containing the number of region pages after the last garbage collection.

11.7 Using Only One Global Region

Managing the region heap does involve more computing compared to having only one or a few heaps (i.e., region pages). It may therefore be the case that for some programs (i.e., programs where memory is not recycled efficiently by region inference) it is better to use an ordinary generational copying garbage collector for the infinite regions, that is, to throw away all region information about infinite regions. We may then use a traditional garbage collector for all data allocated in infinite regions. However, we can still use the region information for finite regions and still allocate them on the machine stack, which may be a significant improvement compared to allocating all objects on the heap.

11.8 When to Increase The Heap

We do not know the demand for memory when we start executing a program. The runtime system must require memory from the operating system if the program requires more memory than already requested. The garbage collector requires the heap size to be larger than the total size of region pages currently allocated to regions. A copying garbage collector using two semi spaces requires the heap to be twice as big as the size of one semi space. Our garbage collector only requires the heap size to be the size of from–spaces plus the size of to–spaces. The size of to–spaces is the size of all region pages used to hold live values after a garbage collection phase.

We need a simple heuristic to decide when to request more memory and also how much. A simple heuristic is to initiate garbage collection every time the free list is exhausted. During garbage collection we request heap space from the operating system corresponding to the size of all to–spaces. However, this will continuously increase the heap size at every garbage collection even though the program may run efficiently with less memory.

An heuristic based on the ratio of heap size to live data, γ, is used in SML/NJ [3]. Let h be the total number of allocated region pages and let l be the number of live region pages (which is less than the number of used region pages). The current ratio is $\gamma = \frac{h}{l}$. Let γ_0 be the desired ratio. If γ_0 is too small ($\ll 2$) then the performance of garbage collection degrades (i.e., we garbage collect too often). If γ_0 is too big we use too much memory and probably have poor locality of reference.

We initiate garbage collection when the number of region pages n in the free list is less than $\frac{h}{\gamma_0}$. This means that the free list cannot hold the amount of live variables that we expect there is; given γ_0 and heap size h we expect the current amount of live data, calculated in region pages, to be $\frac{h}{\gamma_0}$. For instance, if $\gamma_0 = 3$ we initiate garbage collection when $n < \frac{h}{3}$ because we expect l to be $\frac{h}{3}$.

Given γ_0 we have three cases where more memory is requested:

1. a large object is allocated which requires more memory than is available in the free list. We request $\lceil s \rceil_m$ region pages where s is the size of the object and $\lceil \cdot \rceil_m$ rounds up to the nearest number divisible by m (i.e., we let m be the smallest number of region pages that we can require from the operating system). Garbage collection is initiated afterwards.

2. after garbage collection we have l equal to the size of to–space. We request $n = \lceil \gamma_0 l - h \rceil_m$ new region pages from the operating system if $\gamma = \frac{h}{l} < \gamma_0$.

3. we run out of region pages during garbage collection and requests a constant sized chunk of memory, m say, and then continue garbage collection.

The first case is likely not to happen. The third case happens if we have lots of live data or if garbage collection is initiated too late. The rounding in 1 and 2 makes sure that we request a decent amount of memory and not one region page only.

Finding the best value for γ_0 is important to get optimal performance. We invesitigate this in Chapter 15.

Chapter 12

Data Representation

Tagging is normally used for two reasons in Standard ML compilers: polymorphic equality and garbage collection. The ML Kit uses a type based translation that translate programs with polymorphic equality into programs without polymorphic equality such that tagging is not required [21]. The translation handles all Standard ML programs except rare programs using *non–regular* datatype declarations. For instance, it is not possible to build an explicit equality function working on the following datatype declaration; actually it is not even possible to define a function f that can be mapped on the elements in the datatype:

$$\begin{array}{l} \texttt{datatype}\ \alpha\ \mathit{dat\_rec}\ \texttt{=}\ A\ \texttt{of}\ \alpha \\ \qquad\qquad\qquad\qquad \texttt{|}\ B\ \texttt{of}\ (\alpha \times \alpha)\ \mathit{dat\_rec} \\ \texttt{val}\ a\ \texttt{=}\ A(1) \\ \texttt{val}\ b\ \texttt{=}\ B(A(1,1)) \end{array}$$

The values a and b both have type `int` *dat_rec* but it is not possible to build an equality function that compares $A(1)$ and $(B(A(1,1)))$. The function

$$\begin{array}{l} \texttt{fun}\ f\ (A(a), A(b))\ \texttt{=}\ a\ \texttt{=}\ b \\ \ \ \texttt{|}\ \ f\ (B(a), B(b))\ \texttt{=}\ f(a, b) \end{array}$$

gives a type error when compiled:

```
datatype_ex_tags.sml:10.22-10.28 Error:
operator and operand don't agree [circularity]
  operator domain: ''Z dat_rec * ''Z dat_rec
  operand:         (''Z * ''Z) dat_rec * (''Z * ''Z) dat_rec
  in expression:
    f (a,b)
```

The value $A(1)$ has type `int` *dat_rec* and $A(1,1)$ has type (`int` $\times$ `int`) *dat_rec*. We need polymorphic recursion to type such a function. The ML

Kit supports a polymorphic equality function based on tagging for these rare programs and the compiler instructs the user to enable tagging if necessary.

If the ML Kit is used without garbage collection then no tagging is necessary which is an important feature of the ML Kit compared to other Standard ML compilers. However, the garbage collector implemented in the ML Kit does require tagging in order to distinguish pointers from non pointers.

In the following we discuss the data representation used when garbage collection is enabled. We use uniform representation such that all values or pointers to values occupy one word. As in SML/NJ we distinguish between objects containing pointers and objects containing scalars only [4]. The garbage collector may skip traversing objects containing scalars only whereas fields in objects containing pointers must be traversed.

We assume all pointers are word aligned such that the two least significant bits are zero. We use the least significant bit to differentiate between pointers and scalars, that is, the least significant bit is always 1 for scalars. Integers, booleans and units are all scalars and represented as follows: integer i as $2i+1$, value *true* as 3, value *false* as 1 and the unit value *()* as 1. Consult [22] for a discussion about the arithmetic operations working on tagged integers.

The ML Kit can call C functions and results from C functions can be stored in ML values (e.g., records). It is a fatal error if the garbage collector decodes a C value as a pointer value. We therefore require all C values to be tagged as integers with the least significant bit set.

Boxed values are represented by a pointer pointing at the value allocated in a region either in the region heap or on the stack.[1] We reserve the first three bits of boxed values to hold a *descriptor* describing the *kind* of value. A word in memory is written $\boxed{b_{31}b_{30}\cdots b_1 b_0}$ with the least significant bit (b_0) at right.

We have an *immovable* bit saying wheter the value is a constant.

12.1 Scalar and Pointer Records

A *scalar* record is a record containing scalars only. It is not necessary to traverse a scalar record. A *pointer* record is a record containing one or more values that must be traversed. We use the same tag for scalar and pointer records but encode the fields that must be traversed in the descriptor. The descriptor for a record is as follows:

s	o	i	tag

[1] The implementation does not allocate constants (e.g., strings and reals) in a region; they are constants in the program. Such values are never copied by the garbage collector.

The field s (with 13 bits) denotes the size of the record excluding the descriptor.[2] The field o (with 13 bits) denotes the number of fields to skip. We make sure that the values to traverse are packed at the end of the record. We traverse $s - o$ values. The descriptor has offset zero and the first field in the record has offset 1. The field i denotes the immovable bit. The last field *tag* is the descriptor tag being 5 bits. We only use the three least significant bits. Bit three and four are always 0.

12.2 Tagging Objects

The region allocated objects are listed below.

Real: a real is implemented with double precision, that is, two words for the real and three words including the descriptor. Doubles must be double aligned on some architectures, including the HP PA–RISC [41]. Reals are allocated in regions containing reals only and region pages are double aligned. If a real occupies three words then each second allocated real would not be double aligned. We therefore represent reals with four words.

3	3	i	110	–	d_1	d_2

The first word contains the descriptor for a scalar record of size 3 words.

String: a string is represented by a list of *string fragments*. This allows strings to be larger than the size of one region page. Copying a string is done by copying all string fragments into a new list of string fragments. The header of a string contains the string size, descriptor and first string fragment:

size	i	001	*fsize*	*next*	w_1	$\cdots$	w_{fsize}

A string fragment contains a header with the size of the string fragment (*fsize*) and a pointer to the next string fragment (*next*). The *size* field is 26 bits. Consult [22] for more information about the implementation of strings.

Record: all values in a record must be traversed so it is represented as a pointer record.

n	0	i	110	v_1	$\cdots$	v_n

[2]We believe 13 bits for the size field is adequate. A record cannot be larger than a region page and it is unlikely that we will use region pages larger than 8Kb.

The record contains n fields.

Nullary value constructor: a nullary constructor occupies one word containing a tag and *constructor tag* (*c–tag*). The constructor tag denotes the contructor in the datatype binding.

c–tag	*i*	010

The constructor tag occupies 26 bits. Polymorphic equality requires distinct tags on value constructors.

Unary value constructor: a unary constructor occupies two words where the second word contains the value:

c–tag	*i*	011	*value*

The constructor tag (*c–tag*) occupies 26 bits.

Reference: polymorphic equality requires a distinct tag on references.

–	*i*	101	*value*

Ordinary closure: an ordinary closure contains a code pointer in the second word and free variables in the following words. Free variables are either lambda variables, exception constructors or region variables. Lambda variables and exception constructors are traversed by the garbage collector but region variables are not! A region is not a value and a pointer to a region is not part of the set of live values. A pointer to a region may have the two least significant bits arbitrarily set (i.e., the storage mode and multiplicity bits). We get arbitrary results if the garbage collector follows a region pointer because it points either at a finite region containing an arbitrary value or an infinite region descriptor which is not a region allocated object. A closure is therefore divided into a part containing values and a part containing pointers to regions.

We put the code pointer at offset one and the regions after the code pointer. Exception constructors and free variables are packed at the end of the closure. A closure is represented as a pointer record with n fields and the first value to traverse at offset $o+1$:

n	o	i	110	*code_ptr*

reg_ptr_1	$\cdots$	reg_ptr_r	$value_1$	$\cdots$	$value_v$

The encoding makes sure that the garbage collector can copy the entire closure and only traverse the value containing part of the closure.

Letrec closure: a shared closure is similar to an ordinary closure except that the code pointer is missing:

n	o	i	110	reg_ptr_1	$\cdots$	reg_ptr_r	$value_1$	$\cdots$	$value_v$

We have n fields in the closure and $value_1$ is at offset $o + 1$.

Region vector: a region vector contains pointers to regions and is represented at a scalar record.

n	n	i	110	reg_ptr_1	$\cdots$	reg_ptr_n

Tables: tables are divided into *table fragments* in the same way that strings are divided into string fragments. The table fragments are maintained in a binary search tree such that lookup and update are done efficiently. The header of a table contains a descriptor, the table size and the first node in the tree. A tree node contains two pointers for the children and a table fragment:

$size$	i	111	$child_1$	$child_2$	w_1	$\cdots$	w_n

All table fragments are of the same size n and up to $n - 1$ fields may be unused in the last table fragment. We note that it is necessary to zero all table fragments before returning from the allocation function because garbage collection may be initiated while a ML function initializes the table. We note that vectors are implemented as strings (i.e., they are immutable). We use 26 bits for the size field.

Exception name: exception names are used by all exception constructors and consists of two words containing an *exception number* and a *name of exception*. The unique exception number is generated at runtime and implements the generative nature of exceptions. The name of an exception is a pointer to the syntactic name of the exception found in the source program. The name is printed if the exception is never handled after it has been raised.

2	2	i	110	*exn. number*	*name of exn.*

Exception names are implemented as scalar records; the string containing the name of the exception is a constant string and the exception number is also a constant.

Nullary exception constructor: a nullary exception constructor is represented as a pointer record of size one word. The record field contains a pointer to the exception name.

1	0	*i*	110	*name ptr.*

Unary exception constructor: a unary exception constructor is similar to a nullary exception constructor except that it also carries a value:

2	0	*i*	110	*name ptr.*	*exn. value*

Consult [27] for more information about the implementation of exceptions.

Objects represented as a scalar record are never seen by the polymorphic equality function. Of all objects represented with a pointer record the type system makes sure that the record value is the only value seen by the polymorphic equality function. Objects allocated in finite regions, (i.e., in function frames) use the same layout such that we have a uniform access no matter where the object is allocated. Table 12.1 shows the descriptor and content type of each region allocated object. The *content type* being either **pointer** or **scalar** says how the object is scanned by the garbage collector. Objects with content type **pointer** must be traversed to find the boxed values and objects with content type **scalar** are not traversed.

Polymorphic equality (implemented with tags) and garbage collection require tags for different purposes. Polymorphic equality requires that a subset of the region allocated objects can be distinguished precicely. For instance, a unary value constructor must be distinguished from a reference even though they both occupy two words. A garbage collector may view them as the same kind of object because they are both of size two words where the first field contains the tag and the second field a value either boxed or unboxed. The garbage collector requires that all region allocated objects are tagged including region vectors and closures.

There are two kinds of heap allocated objects in In SML/NJ [4]. Objects containing pointers and objects containing scalars only. All objects are allocated as a record where the first field contains the size of the record. This makes it possible to use a small number of different tags only for all the different objects. We use a few more tags in the ML Kit because we have special implementations of tables and strings.

12.3 Eliminating Fragmented Objects

It is possible to eliminate the fragmented objects by considering objects larger than a region page as a special object and allocate them in a separate list of large objects. Each infinite region descriptor then has two lists of region pages; one with constant sized region pages and one with region pages of varying size each holding precisely one object. This complicates

Region allocated value	**Content type**	**Descriptor (binary)**	**Eq.**
String	**scalar**	001	*
Nullary value constructor	**scalar**	010	*
Unary value constructor	**pointer**	011	*
Reference	**pointer**	101	*
Real	**scalar**	110	
Exception name	**scalar**	110	
Region vector	**scalar**	110	
Record	**pointer**	110	*
Nullary exception constructor	**pointer**	110	
Unary exception constructor	**pointer**	110	
Ordinary closure	**closure**	110	
Letrec closure	**closure**	110	
Tables	**pointer**	111	*
Forward pointer	–	$x00$	–

Table 12.1: We have thirteen different region allocated objects and a forward pointer. The tags are in binary notation with the least significant bit at right. The forward pointer contains the tag 00 and a pointer in the same word. The column **Eq.** marks the values recognized by the polymorphic equality function based on tagging. We must be able to distinguish these values precisely.

the garbage collector because two scan pointers for each region descriptor is needed. However, the elimination of fragmented objects may give a significant performance gain and we never have to copy a large object. A special region page containing one large object only is trivially compacted. We have not implemented region pages of varying sizes.

Chapter 13

Root–set and Descriptors

The *root–set* is the set of root values in the graph of live values in the program, that is, the transitive closure of the root–set is the set of live values.

We must find the root–set at any program point where garbage collection can be initiated. The root–set changes dynamically as the program executes and at some program points it may be harder to determine the root–set than at other program points.

The garbage collector must be safe meaning that the garbage collector never reclaims a live object. We also want the garbage collector to be as eager as possible, that is, to reclaim as many objects as possible. Minimizing the root–set increases the number of objects reclaimed; objects not being in the transitive closure of the root–set are reclaimed. A conservative garbage collector cannot minimize the root–set and therefore reclaims fewer objects. To be safe the root–set must at least include the values necessary to build a graph holding the values that are potentially used for the rest of the computation.

We simplify the root–set computation by allowing garbage collection to be initiated at application points only. It is fairly easy to capture the state of the machine stack at each application point and thereby determine the root–set. Applications happen so often that we believe it is flexible enough to garbage collect at application points only.

The compiler described in Part II is organized such that a root–set can be build using machine registers and the machine stack. It is possible to calculate the set of live values, assigned machine registers or allocated on the machine stack, for each function at each application point. We calculate a *frame map*, representing live values, over the function frame and call convention. We also calculate a *register map* over the machine registers. The maps are statically determined for each application point and are inserted into the program binary such that we can find the maps while traversing the machine stack looking for live values.

Offset	Value	Live
0	ph_4	*
1	ph_5	*
2	xs'	*
3	$r22$	
4	$r22$	
5	$r22$	
6	$r22$	
7	$r24$	
8	$r24$	
9	$r24$	
10	$r24$	
11	$k77$	*
12	$r25$	
13	$r25$	

Table 13.1: The function frame has size 14 words and contains 4 live values at the application to *foldl*. The offset is computed from top of the frame. The live values are marked in the third column and is computed by a single backward scan of the program similar to algorithm $\mathcal{F}$ in Chapter 6.

13.1 A Function Frame

A function frame can be viewed as a constant sized memory area holding local values and regions either finite or infinite. Regions are not part of the root–set. Local values are in the root–set if they are live. Table 13.1 shows the function frame for function *fn_xs* in Figure 7.2 and 7.3 on page 139 and 140 at the first application to *foldl*.

The function frame changes as the function evaluates but the frame is fixed at every application point. It is therefore possible to calculate a frame map, represented as a bit vector, describing the cells holding a live value. We use a bit for each word in the function frame and if 1 then the cell holds a value which is in the root–set. The frame map over the function frame in Figure 13.1 is

$$00100000000111,$$

where the least significant bit (at right) represents the top word (ph_4) in the function frame (i.e., offset 0).

frame g
arguments to g
return address to f
results to f

Figure 13.1: Function frame and call convention frame on the machine stack for an example function g. We assume a function f has called g. The call convention frame is allocated at the application to g and the frame for g at entry to g.

13.2 Call Convention

A call convention allocated on the machine stack contains three parts: arguments to the called function, the return address and cells in which result values are stored, see Figure 13.1.

The call convention frame may contain live values in both the argument and result part. This is statically determined at each application point in the function. Note that the live values in a call convention is a property of the called function and not the caller. We let the frame map (computed in the previous section) include both the function frame and call convention frame allocated below the function frame. The frame map is then represented as a bit vector covering both the function frame and call convention frame.

13.3 Callee Save Registers

We use callee save registers for values with a life range that crosses function calls because callee save registers are flushed only in functions using the registers, see Chapter 5. It is impossible to know statically where a callee save register, defined in a function f, is flushed. For instance, say that ph_1, ph_2 and ph_3 are callee save registers and live at an application in f to a function g. The function g uses two callee save registers ph_1 and ph_2 that are flushed at entry to g. Also, at an application to a function h, in g, the register ph_1 is live and assume all registers are flushed at entry to h, see Figure 13.2.

The live registers ph_1 and ph_2 in f are flushed in g but only because g uses the two registers. The two registers ph_1 and ph_2 may not be flushed if another function g' is called from the same application point. At the call to h from g only ph_1 and ph_3 are live (and not ph_2 even though ph_2 is used somewhere in g). We have <u>underlined</u> the flushed registers that contain live values in Figure 13.2. The flushed registers, containing live data, are found at garbage collection time with a backward scan of the machine stack

Live: ...
Flush: $\underline{ph_1}$, ph_2, $\underline{ph_3}$ }*Frame for h*

Live: ph_1
Flush: $\underline{ph_1}$, $\underline{ph_2}$ }*Frame for g*

Live: ph_1, ph_2, ph_3
Flush: ph_1, ph_2, ph_3 }*Frame for f*

Figure 13.2: For each function f, g and h we statically compute the callee save registers live at an application point and the callee save registers flushed at entry to a function. For instance, ph_1 and ph_2 are flushed at entry to function g and ph_1 is live at the application to h. The underlined registers contain live data and are in the root–set. The stack grows upwards.

computing liveness information.

At each application point we statically compute two callee save register maps: a map with all live callee save registers *liveCalleeRegs* and a map with flushed callee save registers *flushedCalleeRegs*. We always have *liveCalleeRegs* $\subseteq$ *flushedCalleeRegs*. In the example in Figure 13.2 we have:

$$\begin{array}{ll} \mathit{flushedCalleeRegs}_f = \{ph_1, ph_2, ph_3\} & \mathit{flushedCalleeRegs}_g = \{ph_1, ph_2\} \\ \mathit{liveCalleeRegs}_f = \{ph_1, ph_2, ph_3\} & \mathit{liveCalleeRegs}_g = \{ph_1\} \\ \mathit{flushedCalleeRegs}_h = \{ph_1, ph_2, ph_3\} & \mathit{liveCalleeRegs}_h = \{\dots\} \end{array}$$

At garbage collection time we start at the bottom of the machine stack and for each function frame compute the set of live and flushed callee save registers *liveAndFlushed*, that is, the set of flushed callee save registers in the root–set.

The set *live* holds currently live callee save registers and initially $\mathit{live} = \emptyset$. At the function frame for f we have $\mathit{liveAndFlushed}_f = \mathit{live} \cap \mathit{flushedCalleeRegs}_f = \emptyset$ and $\mathit{live} = (\mathit{live} \setminus \mathit{flushedCalleeRegs}_f) \cup \mathit{liveCalleeRegs}_f = \{ph_1, ph_2, ph_3\}$. At g we have $\mathit{liveAndFlushed}_g = \mathit{live} \cap \mathit{flushedCalleeRegs}_g = \{ph_1, ph_2\}$ and $\mathit{live} = (\mathit{live} \setminus \mathit{flushedCalleeRegs}_g) \cup \mathit{liveCalleeRegs}_g = \{ph_1, ph_3\}$. At h we have $\mathit{flushedAndLive}_h = \mathit{live} \cap \mathit{flushedCalleeRegs}_h = \{ph_1, ph_3\}$.

Consider the function frame in Table 13.1. Now that we use the above register maps to find flushed callee save registers in the root–set then it is unnecessary to include callee save registers in the frame maps. A callee save register being flushed in a function does not necessarily say that the flushed value is in the root–set. We therefore remove all callee save registers from the frame maps. The machine registers ph_4 and ph_5 are callee save registers in the frame map shown in Section 13.1. The frame map is then:

001000000001

13.4 Frame Descriptors

We have now defined the frame and register maps necessary to compute the root–set. The maps are computed at compile time and inserted into the target code in a *frame descriptor*. Each application has its own frame descriptor containg the following fields:

$$\left.\begin{array}{|l|}\hline \mathit{frameMap} \\ \hline \mathit{frameSize} \\ \hline \mathit{offsetToReturn} \\ \hline \mathit{flushedCalleeSaveRegs} \\ \hline \mathit{liveCalleeSaveRegs} \\ \hline \end{array}\right\}\ \mathit{frame\ descriptor}$$

The last four fields each occupies one word assuming no more than 32 callee save registers. It may be possible to pack the first four fields into less than four words. The *offsetToReturn* field is explained in the next section. The frame size does not include flushed callee save registers but does include the size of the call convention allocated below the function frame. The frame map occupies $\lceil \frac{\mathit{frameSize}}{32} \rceil$ words assuming a 32 bit target machine.

The fields are stored with the frame map at top because the frame descriptor is read from the bottom, see next section.

The smallest frame descriptor size is 5 words and the code size overhead is $5\mathit{noOfApp}$ where $\mathit{noOfApp}$ is the number of applications in the source program.

13.5 Stack Layout

The machine stack holds function frames and call conventions. Figure 13.3 shows an example machine stack with two functions f and g at the top. Function f has been called from a function allocated further down the stack, say h and g is called from f. Say that function g calls function p.

Garbage collection is always initiated at entry to a function, that is, before the function frame of the called function is allocated. We flush the machine registers before calling the garbage collector.

All arguments are live at the time we garbage collect; either allocated in the call convention frame or in the flushed machine registers. The result part does not contain any live values.

The garbage collector is given three arguments:

number of arguments in the call convention frame below the machine registers.

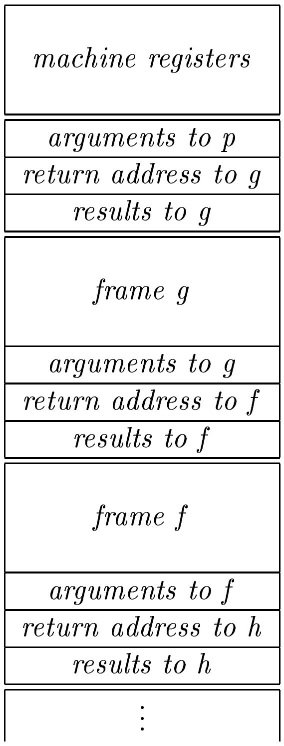

Figure 13.3: The layout of an example machine stack when the garbage collector is called at the application to p from g. The machine registers are flushed at the top of the stack. The call convention to function p has been allocated but the function frame for p is not allocated. The stack grows upwards.

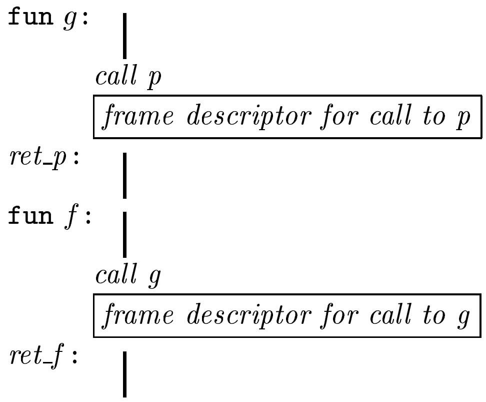

Figure 13.4: Frame descriptors are inserted into the target code at the addresses preceding the return addresses. Given the return address we can find the frame descriptor. The vertical lines denote arbitrary code.

size of the call convention frame below the machine registers including return address and results.

register map describing arguments passed in machine registers. This is a one word bit vector (assuming 32 machine registers).

The three arguments are uniquely determined at entry to a function. The stack pointer points at the first cell above the machine registers. Given the stack pointer and the three arguments, the garbage collector can determine the argument part and the address of the top most function frame (g). The garbage collector knows the number of machine registers on the target machine. The garbage collector also knows the cell containing the return address to function g (given the number of arguments).

The return address to function g is used to find the frame descriptor for function g. The frame descriptor is stored in the program binary at the addresses preceding the return address, see Figure 13.4.

Given the return address we can read the frame descriptor fields in the order: *LiveCalleeSaveRegs*, *flushedCalleeSaveRegs*, *offsetToReturn*, *frameSize* and *frameMap*. The fields *flushedCalleeSaveRegs* and *frameSize* give us the size of the function frame, including the call convention below the function frame on the machine stack. We therefore know the address of the next function frame on the machine stack. The field *offsetToReturn* gives us the cell on the machine stack containing the return address where we find the next frame descriptor.

Using the frame descriptors we can traverse the machine stack from top to bottom. To calculate the *liveAndFlushed* sets, see Section 13.3, we need to traverse the stack bottom up. This is done by reserving a single word

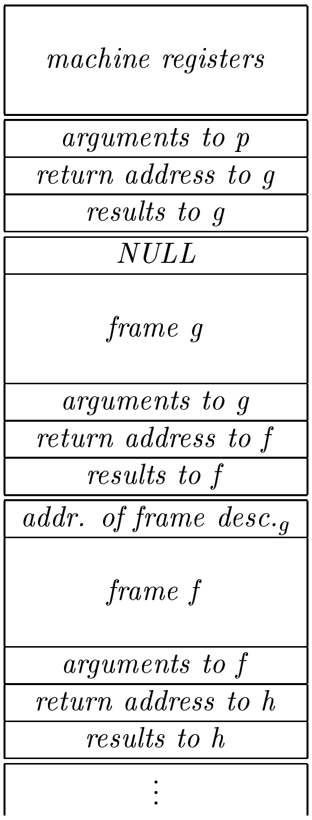

Figure 13.5: The address of the frame descriptor at the application to function g, from f, is stored in the reserved word in the function frame for f. We use NULL to denote the top most function frame. The stack grows upwards.

in each function frame. On the way down the machine stack we store the address of the frame descriptor of the preceding function frame, see Figure 13.5

On the way up the stack we calculate the *liveAndFlushed* set. Consider Figure 13.5. At the function frame for f we know the frame descriptor for g. Inside the function frame for g we find the address of the frame descriptor for f which is used to calculate $liveAndFlushed_f$.

13.6 Implementation

The frame descriptor, as presented, includes all information necessary for callee save registers. However, we do not use callee save registers in the initial implementation and we have therefore simplified the descriptor. A descriptor, in the implementation, contains a frame map and size of frame plus call convention. Neither have we implemented the calculation of live and flushed callee save registers. Currently, a single top down pass on the machine stack is sufficient to calculate the root–set.

Part IV

Measurements

Chapter 14

Performance of the ML Kit backend

We have divided the assessments in two chapters. In this chapter we focus on the backend compiler. The garbage collector is tested in Chapter 15.

We still need to implement a few essential phases in the backend with the register allocator being the most important. To test multiple argument passing we need a phase to un–curry functions and a phase to un–box records. These deficiencies make it uninteresting to compare our compiler with other compilers and we refrain from doing so. However, we note that the compiler compiles all of the Standard ML basis library and the test suite that comes with the ML Kit.[1]

One of the design goals was to implement a backend compiler organized as a series of small phases which should make the compiler easier to comprehend and debug and at the same time not compromise compilation speed. Implementing the backend has shown that this strategy indeed made the task easier. The backend was implemented in a little more than two month which includes adjustments to the runtime system.[2] Tagging, bit vector calculations and the garbage collector algorithm was added in less than two weeks. Section 14.1 gives an overview of the SML modules that implement the new backend. To see the effect on compile times we compare our backed with the backend of the ML Kit version 3 compiler (KitV3) in Section 14.3.

We investigate the effect of enabling tagging in Section 14.4. Tagging has a negative effect on both execution times, memory usage and size of binaries. We let tagging include boxing of lists, that is, if tagging is enabled, then extra descriptors are inserted on boxed values and lists are represented boxed [21].

[1]Some modules (e.g., arrays and words) in the basis library do not compile with tagging enabled because tagging of those modules has not yet been implemented in the runtime system.

[2]The runtime system is basically the same as in the ML Kit version 3.

Section 14.2 introduces the benchmark programs used in this chapter and Chapter 15.

Compiler timings were measured on a Sun UltraSPARC 1 with 256Mb. of main memory. Executables were run on a HP 9000/735 with 240Mb. of main memory. The ML Kit is compiled using Standard ML of New Jersey, version 110.0.3.

14.1 Files

The modules shown in Table 14.1 implement the new backend compiler in the ML Kit. The garbage collector algorithm is implemented in module `GC.c` (658 lines of C code) in the runtime system.

14.2 Benchmark Programs

All the benchmark programs compile without the Standard ML basis library and are used to test both the backend compiler and garbage collector. We use the following programs:

kitlife_old: the game of life using lists. This version has not been optimized for region inference.

kitlife35u: as kitlife_old but optimized for region inference, that is, kitlife35u uses less memory than kitlife_old when compiled with region inference enabled.

kitkb_old: Knuth–Bendix completion. This version has not been optimized for region inference.

kitkbjul1: as kitkb_old but optimized for region inference.

kitkbjul9: as kitkbjul1 but more optimized for region inference, that is, kitkbjul9 uses less memory than kitkbjul1.

kitsimple: A spherical fluid–dynamics program. It performs a series of floating point operations. It has not been optimized for region inference.

kitreynolds2: build a large balanced binary tree and search it. The program performs well with region inference.

kitreynolds3: same as kitreynolds2 but uses lots of memory with region inference.

kitqsort: an implementation of quick sort optimized for region inference.

SML Modules in the Backend Compiler

File	Lines of code	Notes
`CALC_OFFSET.sml`	49	
`CalcOffset.sml`	496	includes bit vector computation.
`CALL_CONV.sml`	54	
`CallConv.sml`	292	implements the call convention.
`CLOS_CONV_ENV.sml`	66	
`ClosConvEnv.sml`	349	environment for closure conversion.
`CLOS_EXP.sml`	147	
`ClosExp.sml`	2178	ClosExp language and closure conversion algorithm.
`CODE_GEN.sml`	29	
`CodeGen.sml`	2031	generates code for HP PA–RISC.
`FETCH_AND_FLUSH.sml`	42	
`FetchAndFlush.sml`	365	insert fetch and flush statements.
`LINE_STMT.sml`	191	
`LineStmt.sml`	884	LineStmt language and linearization algorithm.
`REG_ALLOC.sml`	54	
`RegAlloc.sml`	764	register allocation algorithm.
`SUBST_AND_SIMPLIFY.sml`	50	
`SubstAndSimplify.sml`	311	substitution and simplify algorithm.
`Total`	8352	

Table 14.1: The modules that implement the new backend compiler. Signature files are written with capital letters and functor files with both capital and small letters.

Benchmark Programs

Program	**Lines Of Code**
kitlife_old	205
kitlife35u	288
kitkb_old	708
kitkbjul1	769
kitkbjul9	762
kitsimple	1115
kitreynolds2	91
kitreynolds3	92
kitqsort	157
kittmergesort	130
professor_game	347

Table 14.2: Size of benchmark programs.

kittmergesort: an implementation of merge sort slightly optimized for region inference.

professor_game: finds all solutions to a 4×4 puzzle by searching the set of possible solutions. This program has not been optimized for region inference.

The size of the benchmark programs is shown in Table 14.2. We include kitreynolds2 and kitreynolds3 in the test because they exhibit extreme behaviour with region inference.

14.3 Compilation Speed

Table 14.3 shows the compiler timings for our backend compiler. The abbreviations used are: closure conversion (CC), linearization (LS), dummy register allocation (RA), fetch and flush (FF), calculate offsets (CC), bit vector calculation (CBV), simplify and substitution (SS) and code generation (CG).

Table 14.4 shows the compile timings for KitV3. The abbreviations used are [22, 49, 50]: compile lambda (CL), copy propagation (CP), dead code elimination (DE), register allocation (RA) and code generation (CG).

The closure conversion, linearization and fetch and flush phases are included in the compile lambda phase in KitV3. In general it seems a little more expensive to have three phases instead of one but we must keep in mind that the compile lambda phase has been highly optimized. We have not done any time profiling on the phases in our backend. Even in the case that we

Compilation Speed of New Backend

Program	CC	LS	RA	FF	CO	CBV	SS	CG	Total
kitlife_old	0.19	0.07	0.02	0.23	0.03	0.16	0.08	0.49	1.27
kitkbjul9	1.24	0.17	0.08	1.11	0.27	0.65	0.42	2.02	5.96
kitsimple	1.91	0.43	0.24	1.64	0.36	0.81	0.56	3.15	9.10
kitreynolds3	0.08	0.00	0.04	0.11	0.02	0.07	0.04	0.18	0.54
kitqsort	0.05	0.00	0.00	0.07	0.01	0.03	0.05	0.11	0.32
kittmergesort	0.08	0.00	0.02	0.09	0.05	0.05	0.03	0.15	0.47
professor_game	0.28	0.12	0.03	0.34	0.04	0.25	0.11	0.62	1.79

Table 14.3: All timings are in seconds and do not include time used on garbage collection. The register allocation phase (RA) is the dummy register allocator as described in Chapter 5.

Compilation Speed of KitV3

Program	CL	CP	DE	RA	CG	Total
kitlife_old	0.30	0.14	0.08	0.45	0.23	1.20
kitkbjul9	1.32	0.63	0.39	3.18	0.88	6.40
kitsimple	2.36	0.90	0.72	11.46	1.52	16.96
kitreynolds3	0.10	0.03	0.06	0.18	0.03	0.40
kitqsort	0.06	0.02	0.01	0.11	0.03	0.23
kittmergesort	0.07	0.02	0.01	0.17	0.05	0.32
professor_game	0.32	0.10	0.10	1.32	0.25	2.09

Table 14.4: All timings are in seconds and do not include time used on garbage collection.

cannot optimize either CC, LS or FF then we believe the time difference is a small price for the simplicity gained in less complicated modules.

We cannot compare the timings for register allocation because we use the dummy register allocator. However, we believe that we gain some time on doing register allocation on the linearizated code instead of three address code as is done in KitV3. We have a smaller number of nodes in the syntax tree to process (e.g., records have not been compiled into three address code instructions). The register allocator described in Chapter 5 performs coalescing and we believe that it will not be necessary to use a separate copy propagation phase.

We expect to incorporate dead code elimination into the phases where it is easy to implement and a separate phase should not be necessary. For instance, it is easy to remove code between a tail call and the next label. We have to wait for the code produced by the register allocator described in Chapter 5 but we do not expect to find code, without side effects, that calculate and define variables that are never used by the program.

We are surprised by the compile timings for our code generator being higher than the timings for the code generator in KitV3 . Our code generator should be as simple as the code generator in KitV3 and we probably have a few time inefficiencies that can be eliminated. Even though the code generator generates code for allocation directives, records etc., then it is basically driven by simple macro unfolding.

14.4 Effect of Tagging

We have two important inefficiencies when tagging is enabled. We add an extra descriptor on boxed values which is expensive in both execution time and memory usage. With tagging disabled it is possible to represent lists unboxed [21]. Tagging effects execution time, memory usage and the size of binaries. Table 14.5 shows the execution time and memory usage for programs with tagging enabled and disabled. Memory usage is measured in number of region pages requested from the operating system, that is, a region page used in many regions during evaluation is counted as one region page. The size of a region page is 1Kb. Timings are measured with the unix program `time`. We report the best user time obtained after two or more runs of the program.

We obtain considerable savings by disabling tagging in both execution time and memory usage. Our results, especially on memory usage, with tagging enabled are not as good as the results reported by Elsman [21]. We believe it comes from the tagging of objects that are not seen by the polymorphic equality function. For instance, exception names, region vectors, exception constructors, closures and shared closures are not tagged in the experiments reported by Elsman.

Effect of Tagging on Time and Memory Usage

Program	Tagging Disabled Time	Memory	Tagging Enabled Time	Memory
kitlife_old	41,18	7200	44,41(8%)	13110(82%)
kitlife35u	44,45	30	46,84(5%)	60(100%)
kitkb_old	54,96	29250	65,04(18%)	46020(57%)
kitkbjul1	49,05	2550	58,39(19%)	4440(74%)
kitkbjul9	46,26	2010	55,26(19%)	3690(84%)
kitsimple	101,43	600	111,16(10%)	1260(110%)
kitreynolds3	25,98	16530	29,48(13%)	41490(151%)
kitreynolds2	10,88	30	11,05(2%)	30(0%)
kitqsort	11,74	5100	16,60(41%)	12750(150%)
kittmergesort	5,38	1200	7,77(44%)	3000(150%)
professor_game	15,60	4170	18,41(18%)	9900(137%)

Table 14.5: All running times are in seconds. Memory usage is measured in number of region pages requested from the operating system. One region page is 1Kb. The machine stack is not included. The overhead $\frac{Tag-NoTag}{NoTag}$ is written in percentages.

The size of binaries is shown in Table 14.6.

The results are mostly similar to the results reported by Elsman considering that our binaries are smaller. For instance, we report an overhead of 17% on kitsimple and Elsman reports an overhead of 10% where the size of the binary for kitsimple with tagging enabled is 360Kb. and with tagging disabled 328Kb.

The object files are in general smaller in the new backend than in the old backend. We believe it comes from more compact code generated and that we use stub code when allocating into infinite regions. In the old backend we inline the code for allocating into an infinite region. In the new backend we jump to some stub code containing the code for allocating into an infinite region.

Size of Binaries

Program	**Tagging Disabled Size of Binary**	**Tagging Enabled Size of Binary**
kitlife_old	38553	51401(33%)
kitkbjul9	153425	184481(20%)
kitsimple	217697	254945(17%)
kitreynolds3	16209	19145(18%)
kitqsort	10737	13273(24%)
kittmergesort	13737	17177(25%)
professor_game	58417	66953(15%)

Table 14.6: The size of binaries is in bytes. We show the size of the stripped object file. We have used the unix program `gstrip`. The overhead $\frac{Tag-NoTag}{NoTag}$ is written in percentages.

Chapter 15

Performance of the Garbage Collector

In this chapter we present performance figures for the garbage collector.

Section 15.1 measures the garbage collector combined with region inference. Section 15.1.3 compares the garbage collector with region inference in the ML Kit and tries to answer the following question concerning the case when region inference and garbage collection are combined: of the memory that is reclaimed, what proportion is reclaimed by region management and what proportion is reclaimed by the garbage collector? The heap size influences how often garbage collection is performed. We use different heap sizes in Section 15.2 and seek the optimal ratio of heap size to live data (i.e., γ_0). Section 15.3 looks at the expense of inserting bit vectors into the binaries.

We focus on the cost of garbage collection and not the total cost of storage management, that is, we do not investigate the cost of allocation and region manipulation. Tarditi and Diwan [46] argue that it is necessary to measure the total cost of storage menagement because the time used on garbage collection may be less than the time used on other storage management tasks. Actually, tagging, allocation and checking for garbage collection seems as expensive as garbage collection and they report the total cost of storage management in SML/NJ0.91 to be 19% to 46%; much higher than the time used on garbage collection. Indeed it is interesting to perform an analysis of the low–level details of the storage management scheme in the ML Kit but such a comprehensive analysis is beyond the scope of this project.

15.1 Cost of Garbage Collection

A popular garbage collection method used in Standard ML compilers is generational garbage collection [3, 5]. Generational garbage collection is very effective on Standard ML programs because they allocate a wast amount of data with tiny life ranges and the wast majority of garbage is reclaimed in the younger generations. The overhead of a well implemented generational garbage collector on Standard ML programs is 5 to 10%. The overhead depends on various parameters including the heap size contra the size of live data. In an early version of SML/NJ an heap size of 7 times the amount of live data gave an overhead of garbage collection of only 6% [3].

The garbage collector in the ML Kit cannot make use of the short life range properties of Standard ML programs and must traverse all live data at each garbage collection phase and not only regions containing newly allocated data. However, region inference reclaims most short lived data at almost no cost and it is therefore interesting to see how the combination of simple copying garbage collection and region inference performs compared to generational garbage collection. If region inference reclaims enough short lived data such that the slower garbage collector is used at a frequency corresponding to the frequency of garbage collecting the oldest generations in a generational garbage collector then the performance may be almost identical.

Many system parameters makes these simple judgements doubtful. For instance, it is hard to estimate how region inference influence on locality of reference having the region heap represented as a linked list of region pages. Also, our simple garbage collector uses a breadth first search to find live data and not a depth first search (or an approximately depth-first search [30, page 135]) which is better at obtaining good locality of reference. Our garbage collector is also more complex in that infinite regions are fragmented and finite regions are allocated on the machine stack. These complications cost valuable time at garbage collection time.

We define the overhead of garbage collection to be the time used on garbage collection excluding time used on tagging, that is, the time used in tracing and copying data. We time a program with tagging enabled (t_{tag}) and with both tagging and gc enabled (t_{gc}). Time used on garbage collecting is then $t_{gc} - t_{tag}$. Table 14.5 on page 227 shows our test programs with region inference and tagging enabled. Garbage collection is disabled.

We measure memory usage as the number of region pages requested from the operating system, that is, a region page can be used in many regions during evaluation but is counted as one region page only. It is not in general the case that all region pages are filled with objects and we need the region profiler [27] to measure the memory usage accurately. However, the region profiler has not been ported to the new backend so counting region pages is the best we can do. It is difficult to estimate the inaccuracy that this incurs on the results but we believe the general guidelines to hold. All timings are measured with the unix program `time`. We report the best user time obtained after two or more runs of the program.

The garbage collector is an initial implementation and contains several inefficiencies. We find it plausible that we can gain a speed up of 1.5 to 2 by

GC Enabled and RI Disabled

Program	rp_{total}	rp_{ts}	t_{gc}	**GC time**	**# GC's**
kitlife_old	120	24	56.72	12.31(28%)	2833
kitlife35u	120	24	57.02	10.18(22%)	2833
kitkb_old	5280	1747	236.60	171.56(264%)	675
kitkbjul9	390	91	114.19	58.93(107%)	3977
professor_game	120	17	32.06	13.65(74%)	3201

Table 15.1: All timings are in seconds. Memory usage rp_{ts} is the maximal number of region pages in to–space found after a garbage collection. rp_{total} is the number of region pages requested from the operating system. Each region page is 1 Kb. The GC time is the difference $t_{gc} - t_{tag}$ and the overhead $\frac{GCtime}{t_{tag}}$ in percentages.

removind unnecessary debug code, unnecessary conditionals and optimizing loops.

15.1.1 Simple Stop and Copy

In this section we measure the simple copying garbage collector without region inference, that is, we let region inference use global regions only. All values are allocated in global regions which is equivalent to turning region inference off.

We set $\gamma_0 = 3.0$ (see Section 11.8) and $m = 120$, where m is the smallest amount of region pages that can be requested from the operating system (i.e., when we call `malloc`). Table 15.1 shows the effect of enabling garbage collection when region inference is disabled. The time used on garbage collection **GC time** is calculated as the running time minus the time used when both tagging and region inference are enabled (Table 14.5 on page 227). Using the time when region inference is enabled as base time may be somewhat misleading. However, it has not been possible to find the base time for all the programs when region inference was disabled because the system ran out of memory. We belive the timings and overhead we report in Table 15.1 to be a little (but not significantly) higher than they should be.

As expected, it is evident that a generational garbage collector with an overhead of 5 to 10% is by far better than our simple stop and copy garbage collector with region inference turned off. Besides the lack of generations we also have a more complicated heap split in region pages.

The optimized and unoptimized versions of kitlife behaves similarly but there is a huge difference in the behavior of the two kitkb programs. We believe the set of live values in kitkb_old is so big that tracing and copying are a major burden on the garbage collector. This is consistent with the

GC Enabled and RI Enabled

Program	rp_{total}	rp_{ts}	t_{gc}	**GC time**	**# GC's**
kitlife_old	738	236	49.42	5.01(11%)	41
kitlife35u	90	28	50.38	3.54(8%)	3
kitkb_old	12519	4163	73.50	8.46(13%)	21
kitkbjul1	1764	578	62.91	4.52(8%)	31
kitkbjul9	585	185	60.26	5.00(9%)	49
kitsimple	2067	679	125.10	13.94(13%)	6
kitreynolds2	30	3	12.75	1.70(15%)	1
kitreynolds3	30	7	32.81	3.33(11%)	2822
kitqsort	5994	1995	22.46	5.86(35%)	14
kittmergesort	4512	1494	9.24	1.47(19%)	7
professor_game	149	40	19.56	1.15(6%)	149

Table 15.2: All timings are in seconds. Memory usage rp_{ts} is the maximal number of region pages in to–space found after a garbage collection. rp_{total} is the number of region pages requested from the runtime system. Each region page is 1 Kb. The GC time is the difference $t_{gc} - t_{tag}$ and the overhead $\frac{GCtime}{t_{tag}}$ in percentages.

general experience that tuning a program for regions often makes them run faster on a system that uses garbage collection and no region inference.

15.1.2 Region Inference plus Simple Stop and Copy

In this section we measure the performance of garbage collection when region inference is enabled and all regions are reclaimed each time a garbage collection is initiated. We set $\gamma_0 = 3.0$ and $m = 30$.

In general we get much better timings with an overhead ranging from 6 to 35% and an average of 13%, see Table 15.2. Comparing the kitlife and kitkb programs we see that the unoptimized versions use more time on garbage collection but with a few percentages only. The kitqsort program has a large overhead probably because the live set is big. The kitreynolds2 program is highly region inference optimized and we garbage collect one time only.[1] The kitreynolds3 program, however, uses lots of memory with an extremely small live set and the garbage collector is very effective. The kitsimple, kittmergesort and professor_game have not been region inference optimized and they all perform well.

Comparing Table 15.1 and 15.2 we see that the size of to–space (i.e., rp_{ts}) in general is bigger when region inference is enabled. We measure number of region pages requested from the operating system. With the live

[1]By construction, we always garbage collect at least one time.

objects allocated in many different regions then we need more region pages and each region page is less full. Because the to–space area increases then the total memory usage rp_{total} also increases. In general rp_{total} should be equal to $rp_{ts}\gamma_0$.

Table 15.3 shows the memory usage obtained by enabling region inference, tagging and garbage collection. The results are interesting because we have different savings in the range from -276% to 100% where -276% means that we use 276% more region pages when garbage collection is enabled.

We use more memory when enabling garbage collection on programs that perform well without garbage collection. For instance, kitlife35u, kitsimple and kittmergesort. First of all tagging is expensive and secondly the garbage collector needs both a from–space and to–space. However, we get amazing optimizations on other programs: kitlife_old, kitkb_old, kitkbjul9, kitreynolds3 and the professor_game. The only program that has been region inference optimized is kitkbjul9. The general guideline is that garbage collection should not be enabled on programs that perform well on region inference and enabling garbage collection may in fact have a negative effect. However, if you have a non region inference optimized program then its likely that you save space by enabling the garbage collector.

Notice, that it is a guideline only and the effect of enabling garbage collection very much depends on the program. For instance, we save memory, using garbage collection, on a region optimized program as kitkbjul9. We use more memory, using garbage collection, on kitqsort being region optimized. We save memory, using garbage collection, on the professor_game not being region optimized. We use more memory, using garbage collection, on kitsimple not being region optimized. Many factors (e.g., amount of garbage generated, fragmentation in region pages, size of live set) influence the effect of enabling or disabling garbage collection.

15.1.3 Region Inference contra Garbage Collection

Experience has shown that region inference reclaims most of the garbage generated but we do not have any measurements saying how much that actually is. We have seen that the result of enabling garbage collection very much depends on the target program and in general garbage collection achieves good results on non region inference optimized programs. However, can we estimate how much memory region inference recycles even on non region inference optimized programs? We expect region inference to recycle the vast amount of memory. We can do a simple estimation using Table 15.1 and 15.2, see Table 15.4.

We see that region inference gives us big savings on all programs both in the number of times we perform garbage collection and the time we use on garbage collection, hence region inference has a major impact on the memory usage.

Memory Usage

Program	RI_GC	RI_TAG	$RI\_TAG - RI\_GC$	RI	$RI - RI\_GC$
kitlife_old	738	13110	12372(94%)	7200	6462(90%)
kitlife35u	90	60	-30(-50%)	30	-60(-200%)
kitkb_old	12519	46020	33501(73%)	29250	16731(57%)
kitkbjul1	1764	4440	2676(60%)	2550	786(31%)
kitkbjul9	585	3690	3105(84%)	2010	1425(71%)
kitsimple	2067	1260	-807(-64%)	600	-1467(-244%)
kitreynolds2	30	30	0(0%)	30	0(0%)
kitreynolds3	30	41490	41460(100%)	16530	16500(100%)
kitqsort	5994	12750	6756(53%)	5100	-894(-18%)
kittmergesort	4512	3000	-1512(-50%)	1200	-3312(-276%)
professor_game	149	9900	9751(98%)	4170	4021(96%)

Table 15.3: Memory usage RI_GC is rp_{total} when region inference and garbage collection are enabled. RI_TAG is rp_{total} when region inference and tagging are enabled. RI is rp_{total} when region inference is enabled and tagging/garbage collection disabled. The overheads $\frac{RI\_TAG-RI\_GC}{RI\_TAG}$ and $\frac{RI-RI\_GC}{RI}$ are in percentages.

Garbage Collection contra Region Inference

Program	t_{GC_RI}	t_{GC_NORI}	$\#\ GC_{GC\_RI}$	$\#\ GC_{GC\_NORI}$
kitlife_old	5.01	12.31	41	2833
kitlife35u	3.54	10.18	3	2833
kitkb_old	8.46	171.56	21	675
kitkbjul9	5.00	58.93	49	3977
professor_game	1.15	13.65	149	3201

Table 15.4: All timings are in seconds. t_{GC_RI} is the time used on garbage collection when region inference is enabled and t_{GC_NORI} is when region inference is disabled. Likewise $\#GC_{GC\_RI}$ is number of times we garbage collect when region inference is enabled and $\#GC_{GC\_NORI}$ is when region inference is disabled.

Memory Recycled by Region Inference

Program	Approx. RI
kitlife_old	94%
kitlife35u	99%
kitkb_old	99%
kitkbjul1	99%
kitkbjul9	99%
kitsimple	99%
kitreynolds2	100%
kitreynolds3	31%
kitqsort	70%
kittmergesort	100%
professor_game	98%

Table 15.5: We have approximated RI by calculating RI for each garbage collection cycle and then report the average. We have $\gamma_0 = 3.0$ and $m = 30$.

With the garbage collector we can estimate the fraction of garbage reclaimed by region inference. The fraction depends on the garbage collection strategy used; eventually region inference reclaims all garbage (i.e., when the program ends) and the fewer times we garbage collect the more data is reclaimed by region inference. However, it is resonable to define the data reclaimed by garbage collection to be data that is reclaimed "too late" by region inference corresponding to the chosen value of γ_0. This fraction gives a more precise indication of how eager region inference is.

Let g_i be garbage collection phase i. Let L_i be the amount of live data after g_i (i.e., size of to–space) and let A_p be the total amount of data allocated in the period between g_i and g_{i+1}. Let L_{i+1} be the amount of live data after g_{i+1} and A_{i+1} be the amount of allocated data before g_{i+1} (i.e., size of from–space). The amount of data reclaimed by garbage collection is $A_{i+1} - L_{i+1}$ and the amount of data reclaimed by region inference is $L_i + A_p - A_{i+1}$. The total amount of data reclaimed is $L_i + A_p - L_{i+1}$ and we get the fractions:

$$RI = \frac{L_i + A_p - A_{i+1}}{L_i + A_p - L_{i+1}} \quad \text{and} \quad GC = \frac{A_{i+1} - L_{i+1}}{L_i + A_p - L_{i+1}}$$

We did these calculations on the benchmark programs and Table 15.5 shows, in percentages, how much memory is approximately recycled by region inference.

The results are extraordinary but notice again, that memory is measured in region pages. We believe this has a significant and positive effect on RI because every time a region is deallocated or reset then we count an entire region page even though the region page is not full. However, even with that inaccuracy we belive the results to be promising. Notice, that we only measure the region heap, that is, we do not consider the values stored in finite regions which are also recycled automatically by region inference.

Figure 15.1 and 15.2 shows the two most interesting cases with GC (i.e., 100-RI) as a function of time. Consider Figure 15.1. At garbage collection number 5, the garbage collector reclaims a little less than 6% of the memory reclaimed since garbage collection number 4. Hence, region inference reclaims more than 94% of the garbage in that period.

15.2 Heap Size

The garbage collection algorithm presented in Chapter 11 takes time proportional to the number of live objects, that is, no time is used on reclaiming garbage. We found the complexity to be $t_C = cU$ where U is the amount of live data, and the constant c includes the time used to copy live objects, update forward pointers, handling region status and the scan stack. We can approximate the cost of garbage collection per object. Let H be the heap size. The amount of garbage is then $G = H - U$ and the cost per garbage object is

$$\frac{cU}{H-U} = \frac{c}{H/U - 1}$$

Assuming the amount of live data U is approximately the same between two garbage collections then the formula predicts that garbage collection gets cheaper as the heap size increases.

Appel did a similar calculation with a simple copying garbage collector and found the number of garbage collections and time used on garbage collection to descrease as the heap size increased [2].

There are several issues in the ML Kit that may influence the above prediction.

1. garbage collection is combined with region inference and if region inference reclaims most of the dead objects then the need for garbage collection descreases.

2. for large heap sizes the live objects are likely to be scattered out on the entire heap. As heap size increases we get more global pointers which may have a significant performance degradation on the virtual memory system and cache performance. We get more page faults and the cache is used less efficiently.

3. The garbage collector must manage more region pages with a large heap.

Memory Reclaimed by Garbage Collection

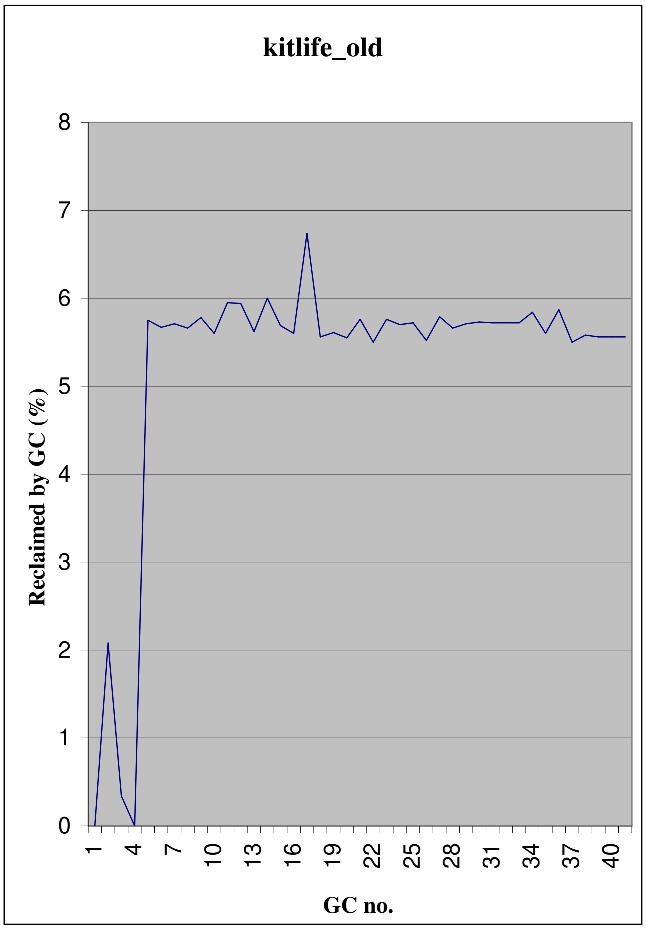

Figure 15.1: The figure shows the amount of memory (in percentages) reclaimed by garbage collection as a function of time. At garbage collection number 2, the garbage collector reclaims approximately 2% of the memory reclaimed since garbage collection number 1. Hence region inference reclaims 98% of the garbage in that period. Time is the garbage collection cycle number.

Memory Reclaimed by Garbage Collection

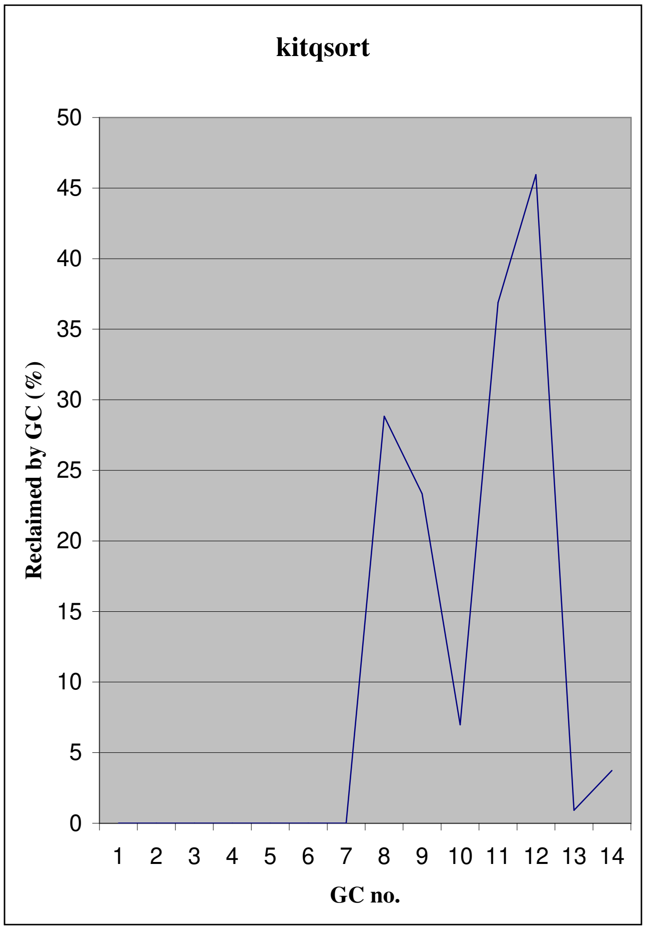

Figure 15.2: The figure shows the amount of memory (in percentages) reclaimed by garbage collection as a function of time. At garbage collection number 10, the garbage collector reclaims approximately 7% of the memory reclaimed since garbage collection number 9. Hence, region inference reclaims 93% percent of the garbage in that period. Time is the garbage collection cycle number.

Garbage Collection and Heap Size(γ_0)

Program	γ_0	rp_{total}	*GC time*	*#GC*	*time/#GC*	rp_{ts}
kitkb_old	2.2	9261	21.77	72	0.30	4196
kitkb_old	2.5	10292	11.12	35	0.32	4105
kitkb_old	3.0	12519	8.46	21	0.40	4163
kitkb_old	3.5	13820	7.49	16	0.46	3940
kitkb_old	5.0	18490	5.28	10	0.53	3692
kitsimple	2.2	1658	20.45	16	1.28	740
kitsimple	2.5	1722	16.05	9	1.78	677
kitsimple	3.0	2067	13.94	6	2.32	679
kitqsort	2.2	4463	34.87	67	0.52	2023
kitqsort	2.5	5087	13.23	28	0.47	2023
kitqsort	3.0	5994	5.86	14	0.42	1995
kitqsort	5.0	9975	2.04	6	0.34	1989
kittmergesort	2.2	4155	4.76	20	0.24	1875
kittmergesort	2.5	3765	2,79	12	0.23	1494
kittmergesort	3.0	4512	1.47	7	0.21	1494

Table 15.6: GC time is in seconds and is the time used on garbage collection (i.e., $t_{gc} - t_{tag}$). Heap size (rp_{total}) is the number of region pages requested from the operating system. To–space (rp_{ts}) is the largest size of to–space after a garbage collection cycle measured in number of region pages.

The heap size is controlled by the ratio γ_0, see Section 11.8 on page 201. In general we have $h \approx l\gamma_0$, where h is the heap size and l is the amount of live data. By increasing γ_0 we indirectly increase the heap size. We have measured the benchmark programs with different values of γ_0.

All programs behaves similarly regarding the number of times we garbage collect. As γ_0 increases the number of times we garbage collect decreases. However, the total time used on garbage collection remains almost the same for some programs: kitlife_old, kitlife35u, kitkbjul1, kitkbjul9, kitreynolds2, kitreynolds3 and the professor_game. Table 15.6 shows the programs where the time used on garbage collection decreases significantly.

Why does the time on garbage collection not descrease as γ_0 increases on all the programs? An important factor is that the programs not listed in Table 15.6 are the programs with the smallest to–spaces (see Table 15.2). This means that less time is used on each garbage collection cycle and the total time used on garbage collection may not increase significantly even though we garbage collect more often.

Table 15.6 shows that the time used on each garbage collection cycle is

Code Size and Bit Vectors

Program	Size (no gc.)	Size (with gc.)	# FN	# App
kitlife_old	51401	57849(13%)	67	148
kitkbjul9	184481	207273(12%)	301	449
kitsimple	254945	290337(14%)	401	794
kitreynolds3	19145	22105(15%)	49	45
kitqsort	13273	14681(11%)	25	13
kittmergesort	17177	19049(11%)	31	26
professor_game	66953	73937(10%)	107	98

Table 15.7: The table shows the size of stripped object files with and without garbage collection in bytes. The overhead comes from extra code at entry to each function and the decriptors inserted at every application point, see Chapter 13. The overhead $\frac{GC-NoGC}{NoGC}$ is written in percentages. The two last columns show the number of functions and the number of non tail calls in the program.

either increasing, constant or descreasing depending on the program. We would expect a constant behavior because the size of to–space is likely to be independent of γ_0. However, we see some variations in the to–space column. An argument for an increasing time per garbage collection cycle as γ_0 increases is that more time is spent manipulating region pages in the garbage collector. We need more analyses in order to make further conclusions.

We believe a value of $\gamma_0 = 3.0$ works well on most programs. However, some programs benefit from adjusting γ_0 depending on the amount of memory one wants to use.

15.3 Bit Vector Size

Table 15.7 shows the size of executables with and without garbage collection. Tagging is enabled in both columns. We have measured the size of the object files, that is, the runtime system is not included in the sizes reported. The object files are stripped with the unix program `gstrip`.

The bit vectors used in the implementation have an overhead of one word plus the words used to hold the frame map. The smallest bit vector is two words. The code inserted at entry to each function is 12 words. The code fetches a flag from memory and if the flag is set then the garbage collector is called. Keeping the flag in a register would reduce the code to 9 words.

Given the number of functions and number of non tail calls we see that most of the overhead comes from entry code to functions. For instance, kitreynolds3 uses 2352 bytes on the entry code and 608 bytes on bit vectors. This is a little more than 3 words per bit vector. Notice, that because all variables are spilled the function frames are larger than they should be. With the register allocator we expect to get a saving in memory used on bit vectors. On kitsimple we use 19248 bytes on entry code and 16144 bytes on bit vectors. This gives an average bit vector size of 5 words.

Allocating finite regions in function frames increases the size of the frame maps. The reason for kitsimple to use 5 words may be that many reals are allocated in finite regions and with tagging enabled each finite region containing a real has size 4 words. Extra space may also be used to keep reals double aligned.

Chapter 16

Conclusion

In this chapter we discuss the contributions and some of the interesting results obtained by combining region inference and garbage collection. We then discuss future work.

16.1 Contributions

We have designed and implemented a new backend for the ML Kit. It has been designed to be simple but still produce good quality code. Not having a register allocator unfortunately makes it impossible for us to verify the quality of the code.

The structure of the backend, with many small phases, has proven to simplify the implementation greatly (i.e., the new backend separates aspects of compilation which were intertwined in the old backend). Each phase concentrates on a few minor tasks and is easier to implement and debug than large and complicated phases that perform many tasks at the same time. The succes of having many phases depends on the ability to express the information computed in each phase in the intermediate languages. It has been challenging to design the intermediate languages and we believe we have succeeded; the LineStmt language is reused throughout the backend and it incorporates all information computed in the different phases. The ability to define polymorphic datatypes in Standard ML is a key feature in the definition of LineStmt in the implementation. The backend has been implemented in a little more than two months and already compiles the Standard ML basis library.

The compilation speed of the new backend is faster than the old backend on most of the benchmark programs. We believe, that this will also hold after implementing the register allocator; the time used on register allocation can be gained from optimizing some of the other phases including the code generator.

We have developed a garbage collector that works with regions. We found that the region management scheme fits well with a simple Cheyney copying garbage collector where each infinite region has its own scan pointer. We solved the stop criteria problem using a scan stack of regions. Finite regions gave us some problems, mainly because they are allocated on the machine stack and therefore not copied. We solved the problems by marking to–space and from–space region pages explicitly. The solution gave us a few more tests when tracing objects but the garbage collector still performs well.

Tagging of values is implemented such that the same tags can be used by the garbage collector and polymorphic equality function. We have seen that tagging is very expensive, both in terms of space and time usage. The size of object files is also significantly smaller when tagging is disabled. The fact that using region inference without garbage collection requires no tags is a significant benefit of the region management scheme.

We have implemented an efficient root–set computation method based on bit vectors inserted into the object files. Object files compiled with garbage collection enabled are 10 to 15% larger than the corresponding object files without garbage collection. This is a resonable increase considering the benefits, at runtime, of having bit vectors stored in the object files.

We have seen dramatic effects of combining region inference and garbage collection in Chapter 15. For instance, a program optimized for region inference (e.g., kitkbjul9 contra kitkb_old) may in fact give memory savings with garbage collection, even if garbage collection is used without region inference. However, region inference does reclaim the vast amount of memory. We have seen that the combination of region inference and garbage collection performs well with an average execution time overhead of 13% even though the garbage collector implementation has not been optimized.

The number of garbage collections (and time spent on garbage collection) descreases drastically for all programs if region inference is enabled. This indicates that region inference does have the same effect as young generations in a generational garbage collector: a garbage collection cycle with region inference enabled corresponds to a major garbage collection cycle in a generational garbage collector.

Enabling region inference and garbage collection may have a negative effect on memory usage compared to using garbage collection only. This is mainly because of fragmentation problems, that is, many region pages are almost empty. The fragmentation problem is a fundamental problem with the region heap and it seems necessary to use both large and small region pages to solve the problem.

A very positive result is that with the combination of garbage collection and region inference, then the region management scheme reclaimes the vast majority of the memory.

We have also seen that the number of garbage collections decreases as the heap size increases (i.e., if γ_0 increases). We found that, for most programs, $\gamma_0 = 3.0$ is a resonable compromise between time spent garbage collecting and wasted space.

16.2 The ML Kit

The ML Kit is a research compiler that is freely available. The current version is version 3 and can be found at

`http://www.diku.dk/research-groups/topps/activities/mlkit.html`

Version 3 does not include the new backend and garbage collector.

16.3 Future Work

It is important that we tune the implementation such that we can get more information about the combination of region inference and simple copying garbage collection. The register allocator and region profiler are two critical components missing.

We have seen that fragmentation problems are essential for region inference. Considering the increase in number of region pages used when enabling region inference and garbage collection contra garbage collection without region inference then there must be a lot of nearly empty region pages. It seems plausible that a smaller region page size may solve some of the fragmentation problems. However, it should be investigated if it is possible to implement a more flexible system with two or more sizes of region pages. Is it the case that the overall memory usage will benefit from having some regions use small region pages only and some regions use large region pages only? Is it possible to improve memory performance using different sized region pages without compromising execution time?

To get more detailed information about the storage management system in the ML Kit a comprehensive analysis looking at all aspects of storage management is needed [46]. Detailed information about the time used on manipulating regions, allocating into infinite regions, tracing etc. may lead to further optimisations. It is also possible to include the memory sub system performance, that is, how paging and cache performance are influenced by region inference and the garbage collector [46, 20]. It is especially difficult to guess how the region heap influences sub system performance as two region pages allocated for the same region may be far apart in the address space.

Adding threading together with a graphics interface to the ML Kit would make it possible to implement business like applications where the user starts several threads running in different windows. Threading influences garbage collection because garbage collection may only happen at gc–points, see Chapter 13. Diwan et al. make sure that all threads reaches a gc–point before they initiate garbage collection [19]. It is also interesting to see if a database interface using methodologies from the world of functional programming can be designed and implemented. For instance, is it possible to map, fold and apply a function on all or some of the entries in a table?

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] Andrew W. Appel. Garbage Collection Can Be Faster Than Stack Allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[3] Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software–Practice and Experience*, 19(2):171–183, 1989.

[4] Andrew W. Appel. A runtime system. *Lisp And Symbolic Computation*, 3:343–380, 1990.

[5] Andrew W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

[6] Andrew W. Appel. *Modern compiler implementation in ML.* Cambridge University Press, 1998.

[7] Thomas H. Axford. Reference Counting of Cyclic Graphs for Functional Programs. *The Computer Journal*, 33(5):466–470, 1990.

[8] Peter Bertelsen and Peter Sestoft. Experience with the ml kit and region inference. Department of Mathematics and Physics Royal Veterinary and Agricultural University, Copenhagen, Draft 1 of December 13 1995.

[9] Lars Birkedal. The ML Kit compiler – working note. Technical report, DIKU, Department of Computer Science, University of Copenhagen, July 1994.

[10] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 171–183, St. Petersburg, Florida, January 1996. ACM Press.

[11] Daniel G. Bobrow. Managing Reentrant Structures Using Reference Counts. *ACM Transactions on Programming Lamguages and Systems*, 2(3):269–273, July 1980.

[12] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *sigplan*, 26(6):192–203, June 1991. pldi91.

[13] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 85–94, June 1982.

[14] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[15] Jacques Cohen and Alexandru Nicolau. Comparison of Compaction Algorithms for Garbage Collection. *ACM Transactions of Programming Languages and Systems*, 5(4):532–553, October 1983.

[16] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[17] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[18] L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM*, 19(7):522–526, 1976.

[19] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282. SIGPLAN, ACM Press, June 1992.

[20] Amer Diwan, David Tarditi, and Eliot Moss. Memory Subsystem Performance of Programs Using Copying Garbage Collection. Technical report, School of Computer Science, Carnegie Mellon University, December 1993. CMU-CS-93-210.

[21] Martin Elsman. Polymorphic Equality – No Tags Required. In *Second International Workshop on Types in Compilation*, March 1998.

[22] Martin Elsman and Niels Hallenberg. An optimizing backend for the ML Kit using a stack of regions. Student Project, Department of Computer Science, University of Copenhagen (DIKU), July 5 1995.

[23] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage–Collector for Virtual–Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[24] David A. Fisher. Bounded workspace garbage collection in an address–order preserving list processing environment. *Information Processing Letters*, 3(1):29–32, July 1974.

[25] John P. Fitch and A. C. Norman. A note on compacting garbage collection. *Computer Journal*, 21(1):31–34, February 1978.

[26] B. K. Haddon and W. M. Waite. A compaction procedure for variable–length storage elements. *Computer Journal*, 10:162–165, August 1967.

[27] Niels Hallenberg. A Region Profiler for a Standard ML compiler based on Region Inference. Student Project, Department of Computer Science, University of Copenhagen (DIKU), June 14 1996.

[28] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, second edition, 1996. ISBN: 1-55860-329-8.

[29] R. John M. Hughes. A semi–incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1082, November 1982.

[30] Richard Jones and Rafael Lins. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management.* John Wiley & SONS, 1997. ISBN: 0-471-94148-4.

[31] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):26–30, July 1979.

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall Software Series, second edition, 1988. ISBN: 0-13-110362-8.

[33] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[34] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3:184–195, 1960.

[35] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[36] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed Closure Conversion. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg Beach, Florida, 21–24 January 1996.

[37] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[38] Tommy Højfeld Olesen and Martin Koch. Compiling a Higher-Order Call-by-Value Functional Programming Language to a RISC Using a Stack of Regions. Master's thesis, Department of Computer Science, University of Copenhagen, October 1996.

[39] Hewlett Packard. *PA-RISC Assembly Language Reference Manual*. Hewlett Packard, fourth edition edition, January 1991.

[40] Hewlett Packard. *PA-RISC Procedure Calling Conventions Reference Manual*. Hewlett Packard, second edition edition, January 1991.

[41] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett Packard, third edition edition, February 1994.

[42] David A. Patterson and John L. Hennessy. *Computer Organization & Design, The Hardware/Software Interface*. Morgan Kaufmann Publishers, 1994. ISBN: 1-55860-281-X.

[43] Robert A. Saunders. The LISP System for the Q–32 Computer. In Edmund C. Berkeley and G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, pages 220–231. The M.I.T. Press, fourth edition, March 1974. ISBN: 0 262 59004 2.

[44] Peter Sestoft. The garbage collector used in Caml Light. e-mail., October 1994.

[45] Laurent Siklóssy. Fast and read–only algorithms for traversing trees without an auxiliary stack. *Information Processing Letters*, 1(4):149–152, June 1992.

[46] David Tarditi and Amer Diwan. Measuring the Cost of Storage Management. Technical report, School of Computer Science, Carnegie Mellon University, May 1995. CMU-CS-94-201.

[47] Mads Tofte. Garbage Collection of Regions. This note outlines how one can garbage collect regions, June 1997.

[48] Mads Tofte. A Brief Introduction to Regions. In *Proceedings on the 1998 ACM International Symposium on Memory Management (ISMM '98)*, pages 186–195, 1998.

[49] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-report 97/12, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, April 1997.

[50] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, Dept. of Computer Science, University of Copenhagen, 1998. (http://www.diku.dk/research-groups/topps/activities/kit3/manual.ps).

[51] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, July 1993.

[52] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[53] Mads Tofte and Jean-Pierre Talpin. Region-based memory management for the typed call-by-value lambda calculus. Submitted for publication, April 1995.

[54] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and Speed in Linear–scan Register Allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, June 1998.

[55] Magnus Vejlstrup. Multiplicity inference. Master's thesis, Department of Computer Science, University of Copenhagen, September 1994. report 94-9-1.

[56] Mitchell Wand and Paul Steckler. Selective and Lightweight Closure Conversion. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 435–445. ACM Press, January 1994.

[57] Jim Welsh and John Elder. *Introduction to Pascal.* Prentice Hall International, third edition, 1998. ISBN: 0-13-491549-6.

[58] Paul R. Wilson. Uniprocessor Garbage Collection Techniques, extended remix – DRAFT. January 1994.

[59] David S. Wise and Daniel P. Friedman. The one–bit reference count. *BIT*, 17(3):351–359, September 1977.