

Compiling a Higher-Order Call-by-Value
Functional Programming Language to a RISC
Using a Stack of Regions

MARTIN KOCH
myth@diku.dk

TOMMY HØJFELD OLESEN
hojfeld@diku.dk

Department of Computer Science
University of Copenhagen
October 11, 1996

Abstract. We describe an SML-to-PA-RISC compiler. The main topic is *inter-procedural register allocation*. We use the known technique of processing the functions in bottom-up order in the call graph. Each function is processed with a fundamentally intra-procedural algorithm, but inter-procedural information is employed. We get considerable speed-up by allowing functions to have individual linking conventions.

The per-function part of the register allocator works on the source language, rather than the target language: It runs before the code generation, and directs the code generation. Measurements suggest that this method can compete with graph-colouring methods. Another good sign is that this register allocation on the source language encompasses short-circuit translation of Boolean expressions in a nice way.

The compiler also *schedules instructions*, and it *rearranges* and *duplicates* basic blocks to avoid jumps.

Everything is implemented in the *ML Kit*, a compiler based on *region inference*, which infers, at compile-time, when memory can be allocated and deallocated and thus makes garbage collection unnecessary. In a phase prior to the translation described in this report, the region inference annotates the source program with allocation and deallocation directives. We discuss how to deal with these directives, but region inference itself is not discussed.

We present measurements of the effect of different ingredients in our compiler. On average, we compile our benchmarks to code that runs in 0.57 of the time of the code generated by SML/NJ version 0.93, and in 0.75 of the time of the code generated by another version of the ML Kit that uses an *intra-procedural*, graph-colouring register allocation.

Contents

1	Introduction	5
1.1	Why register allocation is interesting	5
1.2	Limitation of the job	7
1.3	Overview of the report	8
1.4	Prerequisites	9
1.5	How to avoid reading the whole report	10
1.6	Thanks to	10
1.7	Overview of the compiler	10
1.8	Notation	13
2	Source language	14
3	Intermediate language	20
3.1	RISC	20
3.2	Administrating regions	21
4	Intermediate code generation	24
4.1	Representing data at run-time	24
4.2	Tuples	27
4.3	Constructed values	28
4.4	Regions	31
4.5	References	33
4.6	Functions	34
4.7	Recursive and region polymorphic functions	40
4.8	Exceptions	44
5	Inter-procedural register allocation	55
5.1	Why register allocation?	55
5.2	Why inter-procedural?	56
5.3	Our approach to inter-procedural register allocation	57
5.4	Exploiting information about registers destroyed by a function	58
5.5	Exploiting information about parameter registers	60
5.6	Design decisions conclusion	61
5.7	Call graph	62
5.8	Linking convention	64
5.9	Dealing with recursion	66
5.10	Processing a strongly connected component	67
5.11	Revised overall algorithm	72
5.12	Relation to other approaches	73
6	Per-function register allocation	77
6.1	General approach	77
6.2	Translating an expression	80
6.3	What kinds of values can be allocated to registers	82

6.4	Register allocation strategy	85
6.5	Liveness and hostility analyses	86
6.6	Choosing a register for a value	87
6.7	Heuristic for choosing a register	89
6.8	Which values are spilled	92
6.9	Placing the spill code	93
6.10	Placing spill code intra-procedurally	97
6.11	Placing spill code inter-procedurally	98
6.12	Our spill code placement strategies	100
6.13	Comparison with other approaches	105
7	Development of the inter-procedural part of the algorithm	110
7.1	Overview of the back end	110
7.2	Closure analysis	111
7.3	Sibling analysis	117
7.4	Closure representation analysis	118
7.5	Converting functions to functions of several arguments . . .	120
7.6	Building the call graph	123
7.7	Finding strongly connected components	124
7.8	Traversing the strongly connected components graph . . .	124
7.9	Finding the equivalence classes of λ 's	126
7.10	Potentially recursive applications	126
7.11	Approximating the set of registers that will be destroyed by the code for an expression	128
8	Development of the per-function part of the algorithm	130
8.1	The ω -analysis	131
8.2	Temporary values	134
8.3	The descriptor δ	137
8.4	Allocating regions	140
8.5	Defining and using values	142
8.6	Put points	144
8.7	Control flow forks	149
8.8	Boolean expressions	151
8.9	Function application	157
8.10	Exceptions	164
8.11	Processing a call graph node, λ	172
8.12	The remaining constructs	177
9	Target code generation	182
9.1	Linearising the code	182
9.2	Tailoring the back end to a PA-RISC	186
9.3	Instruction scheduling	187
10	Implementation notes	194
10.1	The correspondence between the report and the code	194

10.2	Some deviations from the report in the implementation . . .	194
11	Assessment	197
11.1	How the measurements were performed	197
11.2	Benchmark programs	198
11.3	Speed	200
11.4	The importance of the different ingredients of the inter-proce- dural register allocation	201
11.5	The per-function part of the register allocation	203
11.6	The importance of the number of registers	204
11.7	Linearising the code	207
11.8	Instruction scheduling	210
11.9	An example	210
11.10	Memory consumption	213
11.11	Conclusions	213
11.12	Directions from here	215
	REFERENCES	216
	SYMBOL TABLE	221

1 Introduction

1.1 Why register allocation is interesting

One reason many programmers do not use higher-level languages is that they are not implemented as efficiently as lower-level languages. This project is an attempt to compile a higher-level language efficiently to a RISC architecture.

One of the most important considerations when compiling to a RISC is *register allocation*. A RISC has a finite number of *registers* to hold data and a set of operations (*instructions*) that work on these registers. There is also a *memory* with a, conceptually, infinite number of *memory cells* that can hold data when the registers do not suffice. Before a value in a memory cell can be used, it must be transferred (*loaded*) to a register, and this takes a long time. Transferring values from registers to memory cells (*storing*) also takes a long time.

Register allocation is the job of deciding which values are put in which registers and at which points in the program values are loaded and stored. The goal is to produce a RISC program that loads and stores as seldom as possible.

Values that are not “live” at the same time can share the same register. So to some extent register allocation is a problem of packing as many values as possible in the registers.

In most cases there will, however, be points in the program where there are more live values than registers, and then some values must reside in memory. Hence, the register allocation problem is also a problem of deciding which values are kept in registers and which are kept in memory.

Since it takes a long time to access the memory, it is the least frequently used values that should be placed in memory. A value can be used frequently in two ways: it can be used at many points in the program, and it can be used in a part of the program that is executed frequently (e.g., a loop).

A value may, however, be used frequently in one part of the program and infrequently in another. Consequently, it would be nice to be able to have a value in a register in some parts of the program and in memory in other parts.

Thus, the register allocation problem in all its generality is: to decide, for each value, and for every point in the program, whether that value is in memory or in a register, and in the latter case, in which register. This should be done in such a way that a value is always in a register when it is used, and such that as few loads and stores as possible need be executed.

Devising an algorithm that solves this problem optimally for any program and in a reasonable amount of time is out of the question. We must content ourselves with a solution that “does well” on “many” programs.

Furthermore, it is a problem of such generality that one must break it into simpler sub-problems to find a solution. Therefore, in practice, the register allocation problem gives rise to a proliferation of problems.

There is no single, correct way to split the general problem into simpler

sub-problems, and the resulting sub-problems can be attacked in many ways. Thus, register allocation is a problem with much room for inventiveness, and since it is so important, there is much to gain in run-time of the compiled programs from a good solution. This makes it a fun problem.

A common way of splitting the general register allocation problem into simpler sub-problems is as follows: Pack the values in registers as well as possible (using some heuristic). Values that do not get a register in this process are kept in memory and loaded every time they are used. Thus, after this process, it is almost completely decided which values are in which registers at each point in the program. There is some freedom still, however: a value in memory must be loaded before it is used, but it does not have to be loaded right before the use. In a final phase, this freedom is utilized to place loads and stores beneficially, e.g., if possible, outside instead of inside a loop.

As far as we know, all register allocators that divide the general register allocation problem this way have used a framework called *graph colouring*. They have varied the heuristic used to “colour the graph”, the heuristic used to decide which values do not get a register, and the way the final phase places the loads and stores.

The graph-colouring framework is conceptually nice, but the graph may become big, it may take a long time to colour it, and a value is either allocated to a register for all of its life or not at all. We have tried another way of splitting the general register allocation problem than the graph-colouring framework. Roughly, we choose registers for values while generating the code.

This way we avoid some of the disadvantages with graph colouring. On the other hand, our strategy may be worse at packing values in registers than a graph-colouring register allocator.

Another respect in which the general register allocation problem is normally simplified is to consider only small parts of the program at a time. A common approach is to split the program into whole functions or into sequences of instructions without jumps, and then perform the register allocation independently for each part of the program.

An advantage of processing the program in small parts is that control flow is less complicated. Another is that it will allow the same value to be put in a register in some parts of the program and in memory in other parts (even though the graph-colouring framework is used). Finally, processing the program in small parts is necessary if the register allocation algorithm runs in time that is quadratic in the size of its input, as these algorithms often do.

The big disadvantage is that, at the boundaries of each program part, all values must be in specific places, often in memory.

It is especially important in a functional language, where function calls are very frequent, that these are implemented efficiently. Therefore, we have not confined our register allocator to work only on one function at a time: we have developed an *inter-procedural* register allocation. It is not practically

possible, however, to consider the whole program in one go, so the inter-procedural register allocation actually considers the functions of the program one at a time, but the register allocation is done for each function using information about the register allocation of other functions.

1.2 Limitation of the job

The setting of this project is more specific than the title.

First, the source language is *Standard ML* (SML) (Milner et al., 1990, Milner and Tofte, 1991).

Second, our translation is only one of many phases; the input program has already been parsed, type-checked, etc., so our source language is not really SML, but rather an intermediate, functional language.

Third, the main topic is inter-procedural register allocation. We will also implement instruction scheduling and a few other things that seem relevant. There are many other relevant issues with which we will not try to deal: closure representation analysis is important because our source language is a higher-order functional language; data representation analysis is relevant because the language is polymorphically typed; many other issues that are relevant in compilers for imperative languages are also relevant (e.g., common sub-expression elimination, constant propagation, loop invariant code motion, strength reduction, and peep-hole optimisations).

Fourth, programs in SML use memory in a way such that the memory cannot be allocated on a stack, as it can in, e.g., Pascal or C. This is a pity, as memory management with a stack is cheap, both in run-time and memory consumption. Instead, SML is normally implemented using a heap with *garbage collection* (e.g. the Standard ML of New Jersey compiler (SML/NJ) (Appel, 1992)). This takes more time at run-time and makes the memory consumption of programs comparatively large. A different approach is *region inference* (Tofte and Talpin, 1993, 1994), which, to some extent, forces the memory usage of SML into the stack model.

Region inference infers, at compile-time, when memory can be allocated and deallocated. This happens in a phase prior to the translation described in this report, and our source language contains directives from the region inference telling when to allocate and deallocate a region, and in what region a given data object is to be placed. We discuss how to deal with these directives, but region inference itself is not discussed.

Fifth, we do not actually implement all of SML: we omit Modules, incremental compilation, and real numbers. We do not discuss the run-time system; it is implemented by Elsmann and Hallenberg (1995).

Sixth, the target machine is Hewlett-Packard's PA-RISC 1.1 (Hewlett-Packard, 1992).

Although the setting thus is rather specific, some aspects are of more general interest. The parts of the translation and register allocation dealing with the region annotations in the source language are, of course, region inference specific, but the translation and register allocation of other constructs

in the language should be generally applicable in compiling higher-order, call-by-value functional languages. The register allocation and translation to the intermediate RISC language is almost independent of the concrete RISC architecture chosen. The same goes for the basic block ordering and duplication. The instruction scheduling can to some extent be re-tailored to other architectures.

We have specified our algorithms formally, and they have been implemented in the *ML Kit*, which is a region-inference-based compiler (Birkedal et al., 1996), and we have also carried out measurements of the effect of different ingredients in our algorithms.

1.3 Overview of the report

The report can be divided in four parts: introductory material (chapters 1–2), design discussions (chapters 4–6), development of formal specifications of the algorithms (chapters 7–9), and assessment (chapter 11). After this overview of the report, there is an overview of the ML Kit and a section on notation. Notice the table of symbols at the end of the report.

Chapter 2 explains our source language, the region-annotated intermediate functional language of the ML Kit.

Chapter 3 explains the intermediate language.

Chapter 4 discusses what intermediate code to generate for each construct in the language. Since we want to concentrate on register allocation, we choose simple solutions in this chapter; there is, e.g., no closure representation analysis or data representation analysis. Except for the exception constructs, we have made the same design decisions as Lars Birkedal in the existing intermediate-code generator, *COMPILE-LAMBDA* (partly documented in (Birkedal, 1994)).

Chapter 5 discusses how to make inter-procedural register allocation. Our inter-procedural strategy uses the idea of (Steenkiste and Hennessy, 1989) of processing the functions in the call graph in bottom-up order. Each function is processed with some fundamentally intra-procedural register allocation algorithm, but because the call graph is processed bottom-up, inter-procedural information about the callees of the function being processed will be available and can be exploited. The chapter discusses what inter-procedural information can be exploited and how it can be exploited. It is also discussed how the call graph can be constructed, since this is not straightforward in a higher-order functional language. Another problem with a higher-order functional language is that more than one function can be applied at a given application. This limits the freedom to treat functions individually. A detailed specification of the inter-procedural part of our algorithm is developed in chapter 7.

Chapter 6 discusses how to do the register allocation of each function in the call graph, i.e., the per-function part of the register allocation. The most interesting characteristic of this part is that we use the structure of the source language to do the register allocation. The register allocation is

done *before* the code generation, and directs the code generation. A more usual way would be to generate code first using “virtual registers”, and then map these to real registers in the register allocation phase afterwards. Also, we do not use graph colouring as many register allocation algorithms do. Generally, the benefit from doing the register allocation on a higher-level language is that there is more information available. This chapter discusses the general issues in making the register allocation on the source language: how the register allocation can direct the code generation, the kinds of values eligible for allocation to a register, how to choose registers for values, how to decide what values are kept in memory, and how to place the code that transfers values between registers and memory. The chapter develops the register allocation for the construct `let $x = e_1$ in e_2` as an example. This should give an impression of the central ideas in our register allocation; the register allocation for the other constructs of the source language is developed in chapter 8.

Chapter 7 develops the inter-procedural part of the algorithm. Notably, it contains an explanation of the closure analysis which is necessary to build the call graph, because functions are values. The closure analysis is based on the region annotations in the source language, and was invented by our supervisor, Mads Tofte. The chapter also explains how functions that take tuples as arguments (the way to pass multiple arguments in SML) are implemented efficiently. The introduction to this chapter gives a list of all phases in the back end. To get an overview, look ahead to this list (p. 110).

Chapter 8 describes the per-function part of the algorithm. First, a liveness analysis is presented; then the register allocation for the different constructs of the source language is developed. We have developed the register allocation for, among other things, the constructs that deal with regions, the constructs that allocate in regions, the exception constructs, and for short-circuit translation of conditional expressions.

Chapter 9 concerns the translation from the intermediate language to PA-RISC assembly language. This translation includes reordering and duplicating basic blocks to avoid jumps, and instruction scheduling.

Chapter 11 presents measurements from our implementation. We compare with the existing back end in the ML Kit and with SML/NJ. We also measure the importance of different ingredients in our register allocation, and the effects of different phases in our back end.

1.4 Prerequisites

You may need basic knowledge of compiler writing for parts of the report.

We explain our source language, but only briefly. It is basically SML, so knowing SML or some other functional language will be a help for the reader. Knowledge of SML corresponding to the level in (Paulson, 1991) is quite sufficient.

Knowledge about region inference is a necessary prerequisite for parts of the report. (Tofte and Talpin, 1993) gives the most detailed account.

Large parts can, however, be read without knowing anything about region inference.

1.5 How to avoid reading the whole report

If you know SML and what region annotations are (or do not care about them), you can to a great extent skip chapter 2. Likewise, if you can guess what a generic RISC language is, you may want to make do with the first part of section 3.2 about the extra instructions for administrating regions and skip the rest of chapter 3. If you know how to translate SML to a RISC, you can skip chapter 4. A quick tour through that chapter is sections 4.2, 4.4, and 4.6.

The central chapters 5 and 6 are fairly self-contained. A quick tour is 5.4, 5.7, 5.8, 5.9, and 5.11; then 6.2, 6.4–6.6, and 6.12.

Chapters 7 and 8 contain all the details and depend on the preceding two. If you are shamelessly into quick tours, brutally skip them; or maybe read 7.1, the first part of 7.2, 7.5, 7.8, 8.2, 8.5, 8.6, and the first part of 8.9.

The rest of the report is almost self-contained. A quick tour is the first parts of 9.1 and 9.3; then 11.3, 11.4, 11.7, and 11.8.

For completeness, we have included a lot of technical detail in the report, especially in chapters 7–9. We have endeavoured to put the more technical parts in the last part of each section, such that the reader not interested in all the details can skip to the next section when it gets too technical and detailed.

1.6 Thanks to

In connection with this project, we want to thank Finn Schiermer Andersen, Lars Birkedal, Erik Bjørnager Dam, Martin Elsmann, Sasja Frahm, Arne John Glenstrup, Niels Hallenberg, Jo Koch, Torben Mogensen, Kristian Nielsen, and Mads Tofte.

1.7 Overview of the compiler

Here is an overview of the ML Kit compiler, of which we have made the back end.

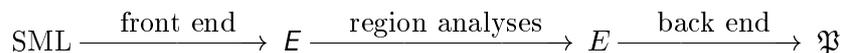


Fig. 0. *Phases in the ML Kit compiler.*

The simple functional language E is approximately the *bare language* subset of SML in the definition (Milner et al., 1990) except that patterns have been compiled into simple patterns. Next, E is a similar *region annotated* language, and \mathfrak{P} is PA-RISC assembly language.

This report is only about the back end, but we give an overview of the two other phases here.

The *front end* comprises the following phases:

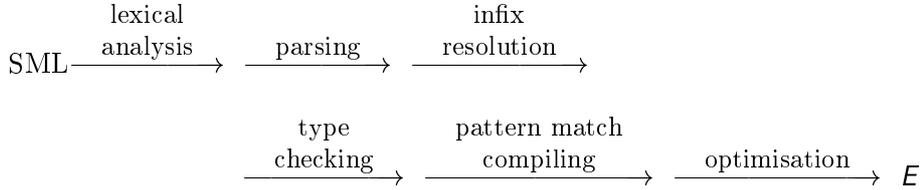


Fig. 1. *Front end phases.*

The type checking phase ensures that only type correct E -programs are produced. The pattern match compiling phase compiles patterns into simpler constructs. The optimiser performs optimising transformations on E . See (Birkedal et al., 1993) for a description of the front end, although especially the optimiser has changed since then.

The *region analyses* are (Birkedal et al., 1996):



Fig. 2. *Region analyses.*

A *region* is a collection of data objects for which memory may be allocated at the same time and deallocated at the same time. The *region inference* decides which data objects should be put in the same regions. Expressions that create new data objects when they are evaluated are annotated with a region: $(10,4)$ may be translated into $(10,4)$ **at** $r17$, indicating that the pair should be *put* in region $r17$. The region inference also decides when a region should be *introduced* and *discharged*. This information is annotated in the program too:

```
letregion  $\varrho$  in  $e$ 
```

declares the region ϱ in a sub-expression e of the program, allowing sub-expressions of e to have the annotation “**at** ϱ ”. Operationally, **letregion** ϱ **in** e introduces a new region, binds ϱ to it, evaluates e , and then discharges the region. This behaviour implies a last-introduced-first-discharged order for regions, i.e., regions can be kept in a stack. Below we discuss what it means to *introduce* and *discharge* a region, and what it means to *put* data in a region.

The *multiplicity inference* decides for each region whether there is a bound on the number of times data will be put into it, when the program is evaluated.

While evaluating (the useless)

```
letregion r17 in (10,4) at r17,
```

data is only put into $r17$ once. Therefore the amount of memory needed for $r17$ can be determined at compile-time, and we say $r17$ has *known size*. This

amount can be allocated, when `r17` is *introduced* and used later when data has to be *put* into the region.

If, on the other hand, the sub-expression e of `letregion ρ in e` builds a list in ρ , a bound on the number of times data is put into ρ cannot in general be determined at compile-time, because the size of the list cannot in general be determined at compile-time. In that case we say ρ has *unknown size*, and it is not possible to allocate all the memory needed for ρ when it is *introduced*, instead memory must be allocated each time data is *put* into the region. For both types of region, *discharging* a region amounts to deallocating all the memory that was allocated for it.

The *physical size inference* uses type information to infer the amount of memory needed for the regions of known size. Together, the physical size inference and the multiplicity inference change each *known-size* region ρ of the region annotated program into $\rho:i$, where i is the number of words needed for ρ ; each *unknown-size* region ρ is changed into $\rho:?$. The example above will be translated to

`letregion r17:2 in (10,4) at r17:2,`

assuming a pair of integers can be accommodated in two words.

One must distinguish *regions* from *region variables*. The ρ 's above are *region variables* which will be bound to *regions* at run-time. The distinction is necessary because a region variable can be bound to different regions at run-time, namely if the binding `letregion`-expression is evaluated more than once. In this respect a region variable is like any other variable—e.g. x of `let $x = e_1$ in e_2` may be bound to different values, during execution of the program, if the `let`-expression is evaluated more than once.

The *storage mode analysis* tries to discover when memory allocated for an unknown-size region can be reused. It will sometimes enable memory to be deallocated earlier than when regions are discharged. Like the other analyses, the storage mode analysis annotates the program. While the multiplicity and physical size inferences influence the way regions are treated fundamentally, the storage mode analysis is more of an add-on; to keep things simple, we will ignore the storage mode annotations in this report (they are not ignored in the implementation).

Concerning region inference, see (Tofte and Talpin, 1993, 1994), of which the former contains an algorithm. See (Birkedal et al., 1996), for descriptions of the other region analyses.

The *back end* is what the rest of this report is about:

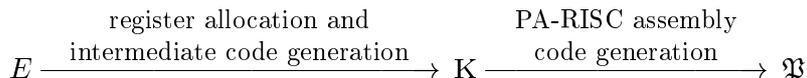


Fig. 3. *Back end.*

Its source language, E , is the target language of the region analyses, i.e., it is a functional language with the region annotations sketched above. We

describe E in the next chapter. The intermediate language K is a RISC language described in chapter 3. The final assembly language, \mathfrak{B} , is similar to K , but of course includes many PA-RISC specific peculiarities (see section 9.2).

1.8 Notation

Assume \mathcal{M} , \mathcal{N} and \mathcal{W} are sets. Then \mathcal{M}^* is the set of finite sequences (tuples) of elements of \mathcal{M} ; $\mathcal{P}\mathcal{M}$ is the set of subsets of \mathcal{M} ; $\mathcal{M} \setminus \mathcal{N}$ is the set of elements in \mathcal{M} that are not in \mathcal{N} . The empty set is \emptyset . Cartesian products of a set are written like this: $\mathcal{M}^4 = \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$.

$\mathcal{M} \rightarrow \mathcal{N}$ is the set of functions from \mathcal{M} to \mathcal{N} , and $\mathcal{M} \xrightarrow{\perp} \mathcal{N}$ is the set of partial functions from \mathcal{M} to \mathcal{N} , i.e., $\mathcal{M} \xrightarrow{\perp} \mathcal{N} = \bigcup \{\mathcal{W} \rightarrow \mathcal{N} \mid \mathcal{W} \subseteq \mathcal{M}\}$.

A λ -abstraction $\lambda a. \mathcal{B}$ with formal argument a and body \mathcal{B} denotes a nameless function; e.g., $\lambda a. 2 + a$ is the function that adds 2 to its argument. The body of a λ -abstraction extends as far to the right as possible.

Function application is denoted by juxtaposition: fg denotes f applied to g ; parentheses are only used for grouping.

We regard a function $f \in \mathcal{M} \rightarrow \mathcal{N}$ as a relation $f \subseteq \mathcal{M} \times \mathcal{N}$: $f a = b$ iff $(a, b) \in f$, and use $a \mapsto b$ as another notation for the pair (a, b) . If \mathcal{M} is a relation, \mathcal{M}^* is the reflexive, transitive closure of \mathcal{M} .

Function application has the highest precedence and associates left. Correspondingly, \rightarrow and $\xrightarrow{\perp}$ associate right. Other operators associate right. \times takes precedence over \rightarrow . Thus $\mathcal{M} \rightarrow \mathcal{N} \times f. \mathcal{M} \mathcal{N}$ means $\mathcal{M} \rightarrow (\mathcal{N} \times ((f(\mathcal{M}))\mathcal{N}))$.

Assume f and g are functions. Then $\text{Dm } f$ denotes the domain of f , and $f + g$ is the function defined by $f + g = \lambda a. \text{if } a \in \text{Dm } g \text{ then } g a \text{ else } f a$. E.g., $\{1 \mapsto 11, 2 \mapsto 0\} + \{2 \mapsto 12, 3 \mapsto 13\}$ is $\{(1, 11), (2, 12), (3, 13)\}$, i.e. the function with domain $\{1, 2, 3\}$ that maps a to $a + 10$. $f \setminus \mathcal{W}$ is the restriction of f to the domain $\text{Dm } f \setminus \mathcal{W}$. \circ is function composition: $f \circ g = \lambda a. f(g a)$.

We abbreviate “if and only if” by “iff”.

2 Source language

Our source language E is the call-by-value λ -calculus, augmented with various features such as references (updateable variables), exceptions, condition constructs, simple and constructed values, primitive operators, and region annotations. This chapter presents E briefly.

It is implicit in the name of a meta-variable which set it ranges over: $e \in E$, $e_1 \in E$, $x \in X$, $\dot{a} \in \dot{A}$, $\vec{\rho} \in \vec{P}$, etc. The only not obvious below are perhaps: $\rho \in P$, and $\varrho \in R$.

We use P for the set of *region variables*, and Y for the set of *λ -bound variables*. The core of E is the call-by-value λ -calculus:

$$E ::= Y \mid E E \mid \lambda Y. E \text{ at } P$$

The λ -abstraction $\lambda y. e_0 \text{ at } \rho$ with *formal argument* y and *body* e_0 evaluates to a nameless function. SML syntax for it is `fn y => e0`. We will say that a sub-expression e' of e is *directly within* e if it is within e but not within any function inside e . E.g., e' is directly within $\lambda y. e' \text{ at } r61$ but not directly within $\lambda y. \lambda z. e' \text{ at } r16 \text{ at } r60$. In the *application* $e_1 e_2$, the function e_1 is called with e_2 as argument. Call by value means that e_2 is evaluated *before* the function is called. This is an important characteristic of the semantics of E —especially for the implementor, as the implementation technology for call-by-need languages (like Haskell or Miranda) is very different from that for call-by-value languages (see, e.g., (Plasmeijer and van Eekelen, 1993)).

As an additional way to define variables, we introduce the `let`-construct. We use X for the set of *let-bound variables*.

$$E ::= X \mid \text{let } X = E \text{ in } E$$

The expression `let $x = e_1$ in e_2` is evaluated by evaluating e_1 and binding the result to x when evaluating e_2 . This can be expressed with λ -abstraction and application (as $(\lambda x. e_2 \text{ at } \rho) e_1$). But that is inefficient, so we treat `let $X = E$ in E` as an independent construction.

We use I for the set of *source language integers*, and O for the set of *binary operators*. The language includes these integer constructs:

$$\begin{aligned} E & ::= I \mid E O E \\ O & ::= + \mid - \end{aligned}$$

Next, we introduce constructs to build and consume *tuples*. We use U for the set of *unary operators*.

$$\begin{aligned} E & ::= (E, \dots, E) \text{ at } P \mid U E \\ U & ::= \#I \end{aligned}$$

The expression (e_1, \dots, e_n) **at** ρ is evaluated by evaluating the sub-expressions and building a tuple of the resulting values. Correspondingly, $\#i$ e_2 selects the i^{th} component of the tuple e_2 evaluates to. Components of a tuple are numbered from zero (SML numbers from one). The front end (figure 1) has compiled the more general *record* of SML to a tuple by deciding an order for its fields. E.g., $\{\mathbf{2=1}, \mathbf{flag=true}, \mathbf{no=3}\}$ may compile to $(1, \mathbf{true}, 3)$ **at** $\mathbf{r4}$, and then $\#\mathbf{flag}$ will compile to $\#\mathbf{1}$, and $\#\mathbf{2}$ to $\#\mathbf{0}$.

We use C for the set of *constructors*, which is split into the *nullary* constructors $\overset{\circ}{C}$ and the *unary* constructors \dot{C} . Here are the extensions to E for constructors:

$$\begin{aligned} C &::= \overset{\circ}{C} \mid \dot{C} \\ E &::= \overset{\circ}{C} \text{ at } P \mid \dot{C} E \text{ at } P \\ &\quad \mid \text{ case } E \text{ of } C \Rightarrow E \mid \dots \mid C \Rightarrow E \mid _ \Rightarrow E \\ U &::= \downarrow \end{aligned}$$

A unary constructor \dot{c} takes one argument as, e.g., **Some**, assuming the following (SML) declaration:

```
datatype 'a option = Some of 'a  
                  | None.
```

A *nullary* constructor $\overset{\circ}{c}$ takes no argument (as **None**).

The syntax of E enforces that unary constructors only appear in expressions of the form $\dot{c}_1 e_2$ **at** ρ , which build a *unary constructed value* consisting of the constructor \dot{c} and the value e_2 evaluates to. Analogously, a nullary constructor $\overset{\circ}{c}$ can only occur in expressions of the form $\overset{\circ}{c}$ **at** ρ , which build a *nullary constructed value*.

This is different in SML, where constructors can be used as function values:

```
fun pmap f (y1,y2) = (f y1,f y2)  
val it = pmap Some (1,2).
```

Here **Some** is used as a function value. To express the equivalent of that SML-program our front end would η -expand **Some**:

```
pmap ( $\lambda y.(\mathbf{Some} \ y \text{ at } \mathbf{r1})$  at  $\mathbf{r2}$ ) (1,2) at  $\mathbf{r3}$ .
```

Assume e_0 evaluates to a constructed value. Then

```
case  $e_0$  of  $c_1 \Rightarrow e_1 \mid \dots \mid c_n \Rightarrow e_n \mid \_ \Rightarrow e_{n+1}$ 
```

inspects the constructor of it and evaluates e_i if the constructor is c_i , or the default expression e_{n+1} if none of the constructors c_1, \dots, c_n match. The argument of the constructed value can be accessed with $\downarrow e_0$: $\downarrow(\mathbf{Kloer} \ 5 \ \text{at } \rho)$ will evaluate to 5.

Note that the `case`-construct always has a default expression e_{n+1} . This is no restriction; a `case`-expression without a default expression can always be converted to one with a default expression.

In SML, pattern matching is used to check what the constructor of a constructed value is. The pattern match compiler (figure 1) has compiled patterns to `case`- and \downarrow -expressions. For instance, the patterns in the SML fragment

```
datatype 'a t = Iron of 'a | Maiden of int
case t of Iron a    => f a
        | Maiden i => g i,
```

may be compiled to

```
case t of Iron => f ( $\downarrow$  t)
        | _    => g ( $\downarrow$  t).
```

Boolean expressions and conditional expressions have the syntax

```
E ::= T | not E | if E then E else E
T ::= true | false
O ::= = | <=
```

Unlike the definition of SML, we do not treat the `if`-construct as a derived form of the `case`-construct, because we want to deal with Boolean expressions specially (section 8.8). The binary operator `=` is only allowed on integers, Booleans, and references, not, e.g., on tuples.

It is an important characteristic of E that it has side effects. This is incorporated by having a special type of value called a reference, which references a memory cell that can be updated.

```
E ::= ref E at P    U ::= !    O ::= :=
```

The expression `ref e_1 at ρ` creates a new reference. If e_1 evaluates to a reference of a memory cell, `! e_1` evaluates to the value in that memory cell, and the memory cell can be updated to the value e_2 evaluates to with `$e_1 := e_2$` .

With side effects, the order of evaluation must be specified to completely determine the semantics of E . We specify that the evaluation order is left to right, e.g., in an application, e_1 must be evaluated before e_2 . Together with call by value this completely determines the evaluation order.

Because exceptions give interesting control flow, we also want them in E . Analogous to constructors, the set A of *exception constructors* is split into *nullary* and *unary* exception constructors. We extend E :

```
A ::=  $\overset{\circ}{A}$  |  $\dot{A}$ 
E ::=  $\overset{\circ}{A}$  at P |  $\dot{A}$  E at P
    | exception A in E | raise E | E handle A => E
```

The expression `exception a in e2` binds a fresh *exception name* to *a* in *e₂*. (It must be a fresh one each time the `exception` expression is evaluated; see section 4.8, p. 50.) The expression `ā at ρ` evaluates to a *nullary exception value* containing the exception name bound to *ā*, and `ā e2 at ρ` evaluates to a *unary exception value* consisting of the exception name bound to *ā* and the value that *e₂* evaluates to.

A *raise* `raise e1`, makes control flow to the nearest enclosing `handle`-expression which handles the exception value that *e₁* evaluates to.

The expression `e1 handle a => e2` handles a raised exception value, if the exception name of the exception value is the same as the exception name bound to *a*. For reasons of presentation, the `handle`-construct here is simpler than SML's. To express the equivalent of the SML fragment

$$e_0 \text{ handle } \mathbf{A1} => e_1 \mid \mathbf{A2} => e_2 \mid \mathbf{A3} => e_3,$$

use

$$((e_0 \text{ handle } \mathbf{A1} => e_1) \text{ handle } \mathbf{A2} => e_2) \text{ handle } \mathbf{A3} => e_3.$$

There are only “`at ρ`”-annotations on expressions that build new values. These annotations are necessary, because they tell in which region memory should be allocated for the new value: to evaluate `λy.e0 at ρ`, it is necessary to allocate memory for the resulting function, and the “`at ρ`”-annotation shows that this will be done in region *ρ*; to evaluate `(e1, ..., en) at ρ`, memory must be allocated for the tuple that is built; to evaluate `ĉ1 e2 at ρ`, memory must be allocated for the constructed value; etc.

Expressions that do not build new values have no “`at ρ`”-annotation. For instance in

$$\text{let } \mathbf{k} = \lambda y.y+1 \text{ at } r13 \text{ in } (\mathbf{k}, \mathbf{k}) \text{ at } r14,$$

the two occurrences of *k* in `(k,k)` have no “`at ρ`”-annotation, for although they evaluate to functions, they do not *build* new functions for which memory must be allocated; they simply reference an already built function.

Boolean and integer constants create new values just as, e.g., a tuple does, but these values will fit in a machine word and are therefore best represented directly instead of as a pointer into a region. Hence, memory is not allocated for the value created by these types of expressions, and consequently, they have no “`at ρ`”-annotations. (This is technical, but should be clear after chapter 4.)

The `letregion`-construct declares *letregion-bound region variables* *P*:

$$E ::= \text{letregion } \acute{P} \text{ in } E$$

It introduces a new region, binds it to the region variable, evaluates the sub-expression, and then discharges the region.

This is not the only way to declare region variables: *E* also has *region polymorphic functions* that take regions as arguments, allowing different calls to the same function to use different regions.

We use \dot{P} for the set of *formal region variables* thus declared.
The set B of (bindings of) region polymorphic functions is given by

$$\begin{aligned} B &::= F\vec{P}Y = E \\ \vec{P} &::= [\dot{P}, \dots, \dot{P}] \mid \epsilon \end{aligned}$$

where \vec{P} is the set of tuples of formal region variables, and F is the set of *names* of region polymorphic functions.

Finally, we introduce

$$\begin{aligned} E &::= \mathbf{letrec} B \cdots B \mathbf{at} P \mathbf{in} E \mid F\vec{P}E \\ \vec{P} &::= [P, \dots, P] \mid \epsilon \end{aligned}$$

This **letrec**-construct serves two purposes. First, it is used for declaring region polymorphic functions. Second, it allows (mutually) recursive functions to be defined: the region polymorphic functions (or: *letrec-functions*) b_1, \dots, b_m declared in **letrec** $b_1 \cdots b_m$ **at** ρ **in** e_{m+1} can call each other. A (*region polymorphic*) *application* $f [\rho_1, \dots, \rho_k] e_2$ applies the **letrec**-function named f to the *actual region arguments* ρ_1, \dots, ρ_k and (*normal*) *argument* e_2 .

A region variable is either **letregion**-bound or a formal region variable; i.e., $P ::= \dot{P} \mid \dot{P}$. These sets have the form:

$$\begin{aligned} \dot{P} &::= R:I \mid R:? \\ \dot{P} &::= R:\Psi \mid R:? \end{aligned}$$

We define that ρ has *known size* iff ρ has the form $\rho:i$ iff it is known at compile-time that all regions bound to ρ will have known size, i.e., iff the number of words needed for each region is known to be i ; ρ has *unknown size* iff ρ has the form $\rho:?$, i.e., iff the number of words needed for the regions bound to ρ is not known; and ρ has *variable size* iff ρ has the form $\rho:\psi$ iff ρ may be bound to both regions with known size and regions with unknown size at run-time. Only formal region variables can have variable size. The ψ is a variable that can be bound to either some i or $?$ at run-time according to the kind of region ρ is bound to.

For instance, in

```
letrec f0[r0:p0]y0 = e0
      f1[r1:?,r2:?]y1 = e1 at r3
in e3
```

the application $f1[r0:?,r0:?]y0$ may be a sub-expression of e_0 . Likewise, $f0$ may be called from e_1 , or (recursively) from e_0 . The formal (region) variables are not in scope in the other function: $r0$ or $y0$ cannot be used in e_1 or e_3 . While $f0$ may be applied to regions with either size because the

size annotation on `r0` is `:p0`, `f1` may only be applied to region variables with unknown size because `r1` and `r2` have `:?` size annotations.

Notice the restriction on region polymorphic function application $f \vec{\rho} e_2$: The syntax ensures that f is always fully applied (to the actual region arguments and the argument). E.g., `(f0,f1) at r1` is not a possible expression. To use a `letrec`-bound function as a value (i.e. to apply it partially), it must be η -expanded: `($\lambda y.f0[r3:?]y, \lambda y.f1[r3:?,r3:?]y$) at r1`. (But notice that these functions are not region polymorphic. For type-checking reasons, region polymorphism is only allowed for `letrec`-functions.)

It is convenient to define the set of *variables*:

$$Z ::= X \mid Y \mid F \mid P \mid A$$

In particular, exception constructors A are regarded as variables.

We assume all variables that occur in a binding position (i.e. y in `$\lambda y.e_0$ at ρ` , x in `let $x = e_1$ in e_2` , $\dot{\rho}$ in `letregion $\dot{\rho}$ in e_1` , $f, \dot{\rho}_1, \dots, \dot{\rho}_k$, and y in b , and a in `exception a in e_2`) are distinct.

Expressions must ML-typecheck (Milner, 1978, Damas and Milner, 1982).

For brevity, we omit many constructs in the report: the rest of the primitive integer operators (`~`, `abs`, `*`, `mod`, `div`, `<`, `>`, `>=`), other primitive operators (`output`, `std_in`, `std_out`, `open_in`, `open_out`, `input`, `lookahead`, `close_in`, `close_out`, `end_of_stream`), strings (`implode`, `explode`, `size`, `chr`, `ord`), case-expressions on integers, strings and exception constructors. These are all implemented. In the report, `=` is allowed on basic types only, but polymorphic equality has been implemented. See chapter 10 concerning discrepancies between the implementation and the report.

Reals have not been implemented, and neither has Modules.

3 Intermediate language

The intermediate language, K , is the language of a RISC (Hennessy and Patterson, 1990), augmented with facilities for handling regions.

3.1 RISC

We write ι for a *word*, I for the set of words, and Φ for the finite (small) set of *registers*, which contain words. We write κ for an *instruction*, and K for the set of instructions.

The intermediate language is a simple three-address language. There are instructions for putting a constant or the contents of a register in a register:

$$\begin{aligned} K & ::= \Phi := \Upsilon \\ \Upsilon & ::= \Phi \mid I, \end{aligned}$$

and for operations on registers:

$$\begin{aligned} K & ::= \Phi := \Phi \pm \Upsilon \\ \pm & ::= + \mid - \end{aligned}$$

E.g., $\phi_1 := \phi_2 - \iota$ subtracts ι from ϕ_2 and puts the result in ϕ_1 .

There is an infinite set of *memory cells*, which contain words, and is indexed by the set of words; i.e., there is a *memory* which is a map from I to I . The only way to access memory cells is through *load* and *store* instructions:

$$K ::= \Phi := m[\Phi \pm I] \mid m[\Phi \pm I] := \Phi$$

The load $\phi_2 := m[\phi_1 + \iota_1]$ changes ϕ_2 to be the word in the memory cell with index $\phi_1 + \iota_1$. Analogously with the store.

We introduce $;$ for sequencing instructions, and the instruction ϵ that does nothing:

$$K ::= K ; K \mid \epsilon$$

Assume $;$ is associative.

There are also instructions for (conditional) control flow:

$$K ::= I : K \mid \text{goto } \Upsilon \mid \text{if } X \text{ then } I \text{ else } I,$$

where the set of conditions is

$$X ::= \Phi \leq \Upsilon \mid \Phi = \Upsilon \mid \Phi.I \mid \neg X.$$

The label of κ is ι if $\iota : \kappa$ occurs in the program, and then the effect of the *jump goto* ι is to execute κ . Because labels are words, one can jump to the contents of a register. The *conditional jump* *if* χ *then* ι *else* $\bar{\iota}$ jumps to ι if the condition χ is true; otherwise to $\bar{\iota}$. $\neg\chi$ is true iff χ is false. The condition $\phi.\iota$ tests whether bit number ι in the word in ϕ is 0 or 1.

The language described so far constitutes the basic intermediate language. All features we add in the following can be described in terms of this language.

We need a stack, which we will call the *K stack* to distinguish it from other stacks. Assume a specific register, ϕ_{sp} , is reserved for a stack pointer. It is convenient with some abbreviations: `pop ϕ` pops a word and puts it in ϕ ; `pop` also pops a word, but does not put it in a register; `push ϕ` pushes the word in ϕ . These abbreviations are defined

$$\begin{aligned} \text{pop} &= \phi_{\text{sp}} := \phi_{\text{sp}} - 1 \\ \text{pop } \phi &= \text{pop} ; \phi := \text{m}[\phi_{\text{sp}} + 0] \\ \text{push } \phi &= \text{m}[\phi_{\text{sp}} + 0] := \phi ; \phi_{\text{sp}} := \phi_{\text{sp}} + 1. \end{aligned}$$

With these definitions, the stack grows upwards in memory, and ϕ_{sp} points to the first free word on the stack.

3.2 Administrating regions

It will be necessary to introduce and discharge regions, and allocate memory in them. We introduce the instructions

$$\phi := \text{letregion} \quad \text{endregion} \quad \phi_1 := \text{at } \phi_2 : \iota \quad \text{endregions } \phi.$$

A new region is introduced with `$\phi := \text{letregion}$` , which assigns to ϕ an identifier of the region.

If ϕ_1 identifies a region, `$\phi_2 := \text{at } \phi_1 : \iota$` allocates ι words in that region and sets ϕ_2 to point to them.

The instruction `endregion` discharges the most recently introduced region, i.e., it deallocates all memory allocated in that region.

As `endregions ϕ` is needed for a specific reason when raising an exception, it will be explained when we discuss how to implement exceptions in section 4.8.

Implementing the region instructions

Now we explain the semantics of the region instructions in terms of simpler *K* instructions. You can skip this if you are satisfied with the less detailed explanation of these instructions.

A region is represented by a list of fixed-size *chunks* of memory. They are kept in a stack:

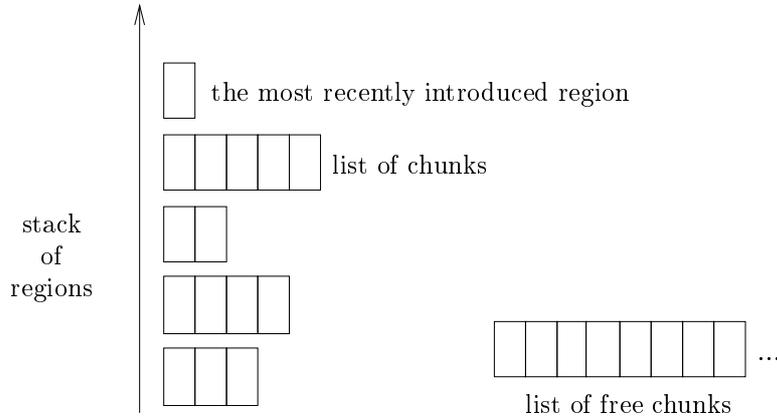


Fig. 4. *The conceptual representation of regions: a stack of lists of chunks of memory.*

The instruction $\phi := \text{letregion}$ pushes a chunk list with one new chunk on this stack, and puts an identifier of the region into ϕ . The instruction $\phi_1 := \text{at } \phi_2 : \iota$ tries to acquire ι words in the last chunk in the region identified by ϕ_2 . If there is not ι words free in this chunk, $\phi_1 := \text{at } \phi_2 : \iota$ puts a new chunk at the end of the list of chunks, and acquires its ι words there. The instruction endregion pops the topmost list of chunks, and they can then be reused. Thus, there is also a list of *free* chunks.

These lists and the stack are implemented as illustrated in this figure:

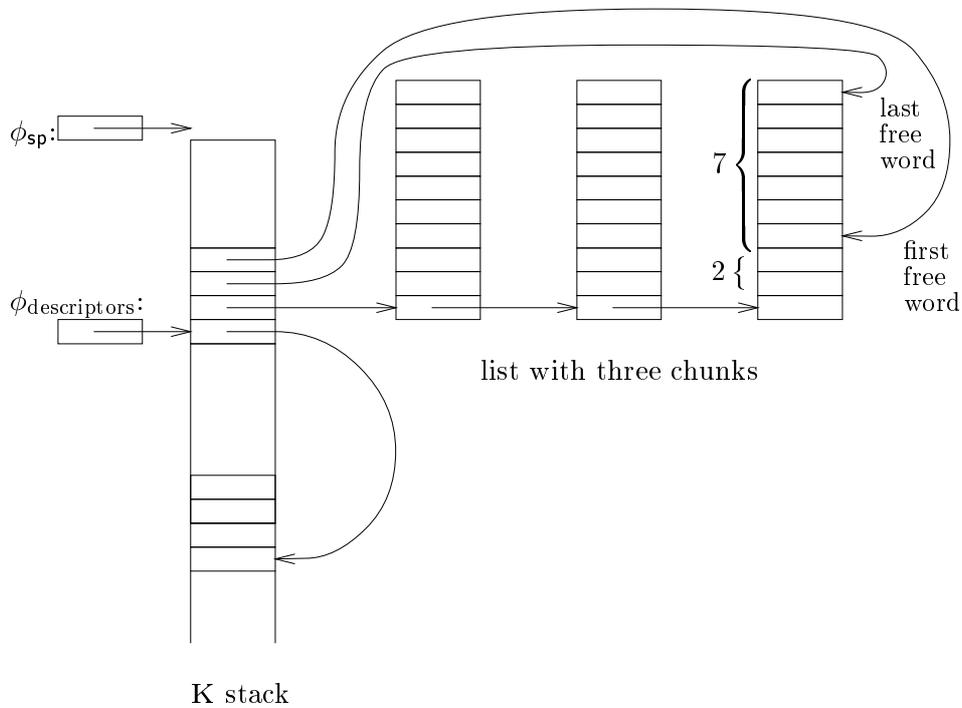


Fig. 5. *The concrete representation of regions.* A box  depicts a word. In this illustration, 9 words can be allocated in each chunk.

The lists of chunks are implemented as linked lists: each chunk has a pointer to the next chunk. The stack of chunk lists is implemented as a linked list of *region descriptors*: each region descriptor has a pointer to the next region descriptor and a pointer to its chunk list. Furthermore, because we need to know how much free memory is left in the last chunk in the list, each region descriptor also has pointers to the first and last free words in the last chunk. In figure 5, two words have been allocated in the last chunk in the topmost region; seven words are free. The region descriptors are placed in the K stack, and we have assumed a fixed register $\phi_{\text{descriptors}}$ points to the topmost region descriptor.

With this data structure, the K code to implement the region instructions is straightforward, though technical; we give an implementation of $\phi := \text{letregion}$. Assume a register ϕ_{free} is reserved to point to the first chunk in the list of free chunks.

$\phi := \text{letregion} =$	$\phi := \phi_{\text{sp}} ;$ $\text{push } \phi_{\text{descriptors}} ; \phi_{\text{descriptors}} := \phi$ $\text{push } \phi_{\text{free}} ;$ $\phi_{\text{tmp}} := \phi_{\text{free}} + 9 ; \text{push } \phi_{\text{tmp}} ;$ $\phi_{\text{tmp}} := \phi_{\text{free}} + 1 ; \text{push } \phi_{\text{tmp}} ;$ $\phi_{\text{free}} := \text{m}[\phi_{\text{free}} + 0]$	pointer to region descriptor push region descriptor set up pointer to chunk list push pointer to last free word push pointer to first free word remove chunk from free list.
------------------------------	---	---

Instead of expanding the region instructions to simple K instructions, the compiler may choose to implement them as sub-routines or calls to a run-time system. In any case, they will be implemented as pieces of code. One aspect of this is important to the translation presented in this report: It must know which registers are destroyed by these pieces of code. So we let $\hat{\phi}_{\text{letregion}}$ denote the set of registers that are destroyed, when a $\phi := \text{letregion}$ -instruction is executed, and let $\hat{\phi}_{\text{endregion}}$, $\hat{\phi}_{\text{endregions}}$, and $\hat{\phi}_{\text{at}}$ denote similar sets for the other instructions. With the implementation of $\phi := \text{letregion}$ above, $\hat{\phi}_{\text{letregion}} = \{\phi_{\text{tmp}}\}$.

Here we have reserved specific registers $\phi_{\text{descriptors}}$ and ϕ_{free} ; the compiler may choose to use memory cells.

4 Intermediate code generation

In this chapter we will discuss how each kind of expression in the source language is translated to the intermediate language. We will not worry about how registers are chosen for values, but merely assume that a register will always be available to hold the result of a computation. Register allocation is discussed in the following chapters which rely on the design decisions made in this chapter.

The constructs of our source language E may be split into two groups: constructs that *build* values, and constructs that *consume* values. For each kind of value, there are constructs to build that kind of value, and constructs to consume it. For example, a tuple is built by (e_1, \dots, e_n) **at** ρ , and consumed by $\#i$ e_2 . A normal function value is built by $\lambda y. e_0$ **at** ρ , and consumed by a (normal) application, e_1 e_2 . **letrec**-function values are built by **letrec** $b_1 \dots b_m$ **at** ρ **in** e_{m+1} , and consumed by $f \vec{\rho} e_2$. Constructed values are built by \hat{c} **at** ρ and \hat{c}_1 e_2 **at** ρ , and they may be consumed by both $\downarrow e_2$ and **case** e_0 **of** $c_1 \Rightarrow e_1 \mid \dots \mid c_n \Rightarrow e_n \mid _ \Rightarrow e_{n+1}$. The remaining constructs may also be split into constructs that build and constructs that consume.

Before we can decide what code should be generated for a given construct, we must decide what information the value manipulated by this construct should comprise at run-time. The general strategy for doing this is as follows: For each kind of value we start by discussing what the constructs that consume this kind of value should do. This discussion reveals what information values of this kind must comprise at run-time (e.g. a function value must contain the code for the function). After having discussed how the values of this kind should be represented at run-time, we can describe what code should be generated for the corresponding constructs (e_1 e_2 and $\lambda y. e_0$ **at** ρ in the case of function values).

Since we focus on register allocation, the guiding principle in this chapter is to choose a simple solution to problems, and we have made the same design decisions as in the existing intermediate-code generator COMPILER-LAMBDA. The representation of regions is also as in COMPILER-LAMBDA. It is described in (Birkedal et al., 1996). The exception is exceptions, which we implement differently from COMPILER-LAMBDA.

We start with some general reflections on how to represent data at run-time.

4.1 Representing data at run-time

The *atomic* values are the values that can be represented in one word: integers, I , Booleans, T , and constructors, C . We assume there are functions to convert atomic values to their representation as words, $\iota \in I$:

$$\begin{aligned} \llbracket \cdot \rrbracket_{I \rightarrow I} &\in I \rightarrow I \\ \llbracket \cdot \rrbracket_{T \rightarrow I} &\in T \rightarrow I \\ \llbracket \cdot \rrbracket_{C \rightarrow I} &\in C \rightarrow I. \end{aligned}$$

We require that $\llbracket \cdot \rrbracket_{C \rightarrow I}$ map different constructors from the same **datatype**-declaration in the SML source program to different representations, but constructors from different **datatype**-declarations can have the same representation; strong typing ensures that there will never be an opportunity to mistake one for the other. (Our source language does not have a **datatype**-declaration. It just has one set, C , of constructors. The constructors originate from **datatype**-declarations in the original SML program, but this is of no concern at this point in the translation; all we need to know is the representation of each individual constructor, and this is provided by $\llbracket \cdot \rrbracket_{C \rightarrow I}$.)

One way to naturally represent *composite* (i.e. non-atomic) values, such as tuples, is as consecutive words representing the atomic values that constitute the composite value. For example, the tuple value $(1, (\mathbf{true}, 3), 4)$ is made up of the atomic values 1, **true**, 3, and 4, and it can be represented by the words:

$$\boxed{\llbracket 1 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket \mathbf{true} \rrbracket_{T \rightarrow I}} \quad \boxed{\llbracket 3 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 4 \rrbracket_{I \rightarrow I}}$$

Here a box denotes a word, and juxtaposition of boxes means that the denoted words are consecutive.

This *flat representation* will not work in general, however, because the source language, E , is type polymorphic. The problem is illustrated by the program

```
let p = λy.(y,y) at r2 at r1 in
  let d = λyy.#1 yy at r7 in
    d(d(p(p 2))).
```

The function \mathbf{p} makes a pair of its input y . Since y may have any type, \mathbf{p} is a type polymorphic function. If we do not want to make specialised versions of the code for \mathbf{p} , the same code must be able to handle y 's of any type, e.g. the value 2 and the value $(2,2)$. Using flat representation, the same code for \mathbf{p} should transform the word $\boxed{\llbracket 2 \rrbracket_{I \rightarrow I}}$ to the consecutive words $\boxed{\llbracket 2 \rrbracket_{I \rightarrow I}}$ $\boxed{\llbracket 2 \rrbracket_{I \rightarrow I}}$, and those it should transform to

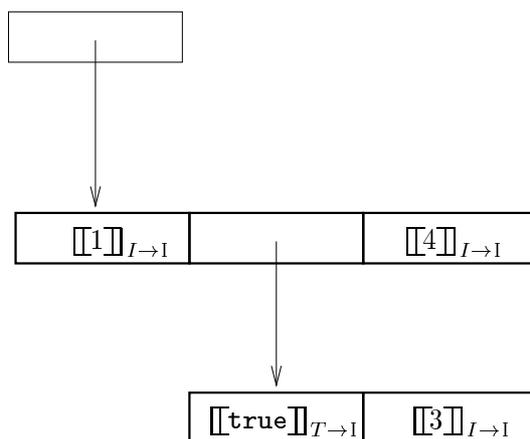
$$\boxed{\llbracket 2 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 2 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 2 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 2 \rrbracket_{I \rightarrow I}},$$

when \mathbf{p} is called the second time. To do this, \mathbf{p} must know the size of the flat representation of its input.

The function \mathbf{d} takes the second component of a pair of values of any type. From the words $\boxed{\llbracket 2 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 2 \rrbracket_{I \rightarrow I}}$, \mathbf{d} should extract the second word, while it should extract the third and fourth words, when given the representation $\boxed{\llbracket 2 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 2 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 2 \rrbracket_{I \rightarrow I}} \quad \boxed{\llbracket 2 \rrbracket_{I \rightarrow I}}$ of $((2,2),(2,2))$. Obviously, \mathbf{d} must know the layout of the pair, if we use flat representation.

The solution normally applied in implementations of type polymorphic languages is to use a uniformly sized representation of all types of values. This representation is called *uniform representation*. The size used for all types of values is usually one word. Composite values are then represented as a pointer (which *will* fit in a word). The pointer points to the “actual

representation” of the composite value, which is like its flat representation, except that the values that constitute the composite value are themselves represented in uniform representation. For instance, the uniform representation of $(1, (\mathbf{true}, 3), 4)$ is



Here fat-edged boxes denote memory cells; other boxes denote words, as before. (A memory cell is not the same as a word: a memory cell *contains* a word. A word may reside in either a memory cell or a register.) The part of the representation of a value that *must* be in memory (the part in the fat-edged boxes) we will call the *actual representation* of the value.

With this representation, \mathbf{p} knows that the size of its input is always one word, and \mathbf{d} knows that it can always find the second component of a pair as the second word of the actual representation of the pair. Another advantage of uniform representation is that it saves memory in cases where values are duplicated: When \mathbf{p} is applied to a value, the memory consumption is the two words for the pair, regardless of the size of the flat representation of the value. With flat representation, the memory needed for the pair will be the double of the size of the value.

Uniform representation is inefficient for two reasons: It is necessary to dereference a pointer whenever a composite value is accessed (e.g. when an element of a tuple is accessed), and the actual representation must reside in memory (one cannot have a pointer to a register). The latter restriction makes it impossible to allocate composite values to registers.

One remedy to the inefficiencies introduced by uniform representation uses that it is only necessary to use uniform representation for a value when it is passed as an argument to type polymorphic functions (as \mathbf{p} or \mathbf{d}). With this method, *representation* or *boxing analysis* decides when values must be in uniform representation and when flat representation can be used (Leroy, 1992, Henglein and Jørgensen, 1994, Jørgensen, 1995).

A completely different way of handling data representation in the presence of polymorphism is the *intensional type analysis* of (Harper and Morrisett, 1995) used in the ML compiler TIL (Tarditi et al., 1996). In that approach,

the type of the argument is passed to type polymorphic functions at run-time, such that a flat representation can be used instead of the uniform representation. In practice, TIL will often specialise type polymorphic functions and avoid passing types at run-time.

These sophisticated solutions are beyond the scope of this project, however; we shall always use uniform representation, although we realise that it will be a significant limitation on the register allocation that we are not able to allocate a composite value to a collection of registers.

Concerning polymorphic equality and tags, see chapter 10. In this presentation, integers are not tagged, so we may use the function $\llbracket \cdot \rrbracket_{I \rightarrow I}$ not only to generate the run-time representation of source language integers, but also for, e.g., index numbers.

4.2 Tuples

We have already indicated in the previous section how tuples are represented. As an n -tuple will not generally fit in one word, it is represented as a pointer to an actual representation, which is n consecutive words holding the uniform representation of the n values that constitute the tuple.

The tuples in the source language, E , represent both tuples and records in SML. This also holds for a record with only one field: The record $\{\mathbf{a}=7\}$ will have been translated to the 1-tuple E -expression $(7) \text{ at } \rho$. The uniform representation of a 1-tuple could be one word holding the representation of the single value of the tuple, but treating 1-tuples specially would give more trouble than benefit.

The 0-tuple (the result of $() \text{ at } \rho$) deserves special mentioning, as it is used often in SML: Expressions that are evaluated for their side-effects ($e_1 := e_2$) evaluate to the 0-tuple, for any expression must evaluate to something. The 0-tuple, which has type **unit**, need not be represented explicitly, because there is no built-in operation with **unit** in its input type. (Remember we have assumed equality is not defined on tuples in E , p. 16.) Of course, we must build a pair, when evaluating the expression $(e_1 := e_2, e_3 := e_4) \text{ at } r5$, but the exact contents of the two words of the actual representation of the pair is of no concern: although they can be retrieved again from the pair, they cannot be used for anything. We decide to represent $()$ as any arbitrarily chosen value, for then the code for $e_1 := e_2$ does not have to bother with putting some specific value representing $()$ in the destination register; it can simply leave any arbitrary value in the register. The code for $e_1 := e_2$ is developed in section 4.5.

The code to build a tuple, i.e. the code for the construct $(e_1, \dots, e_n) \text{ at } \rho$, must allocate memory for a tuple in region ρ , evaluate each sub-expression, and store the result into the tuple. We use $\phi_1 := \boxed{\text{code to evaluate } e_1}$ as a shorthand for “code to evaluate e_1 and put the resulting value in ϕ_1 ”.

$$\begin{aligned}
\phi &:= \boxed{\text{code to evaluate } (e_1, \dots, e_n) \text{ at } \rho} \\
&= \phi_t := \boxed{\text{the address of } n \text{ new cells in } \rho} ; \\
\phi_1 &:= \boxed{\text{code to evaluate } e_1} ; m[\phi_t + \llbracket 0 \rrbracket_{I \rightarrow I}] := \phi_1 ; \\
&\quad \vdots \\
\phi_n &:= \boxed{\text{code to evaluate } e_n} ; m[\phi_t + \llbracket n - 1 \rrbracket_{I \rightarrow I}] := \phi_n ; \\
\phi &:= \phi_t.
\end{aligned}$$

The code for the construct that consumes tuple values is:

$$\begin{aligned}
\phi := \boxed{\text{code to evaluate } \#i \ e_2} &= \phi_2 := \boxed{\text{code to evaluate } e_2} ; \\
&\phi := m[\phi_2 + \llbracket i \rrbracket_{I \rightarrow I}].
\end{aligned}$$

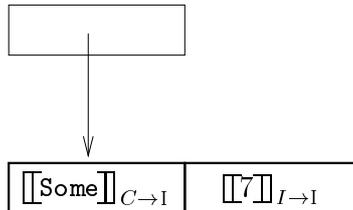
(Remember that $\#0$ extracts the first component which resides at offset 0 in the tuple.)

4.3 Constructed values

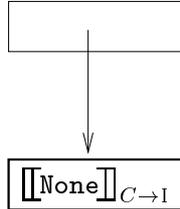
Two constructs consume constructed values: With

$$\text{case } e_0 \text{ of } c_1 \Rightarrow e_1 \mid \dots \mid c_n \Rightarrow e_n \mid _ \Rightarrow e_{n+1}$$

the constructor of a constructed value can be inspected, and, according to what it is, the proper sub-expression is evaluated. The argument of a *unary* constructed value can be extracted with $\downarrow e_2$. This implies that the run-time representation of a unary constructed value must comprise both the constructor and the argument. In other words, a unary constructed value is a pair, and its representation is like that of a 2-tuple: The uniform representation of a unary constructed value is a pointer to two consecutive words in memory. The first word contains the representation of the constructor, and the second contains the representation of the argument. The value `Some 7` will be represented



The *nullary* constructed value that \hat{c} at ρ evaluates to is conceptually a 1-tuple, and it is represented as a 1-tuple, i.e., as a pointer to one word in memory that contains the representation of the constructor:



Here the indirection is necessary, not to achieve uniform representation (the nullary constructor will fit in a word), but rather to ensure that the constructor of both unary and nullary constructed values can always be accessed the same way, namely by dereferencing a pointer. This is needed by the code for

```

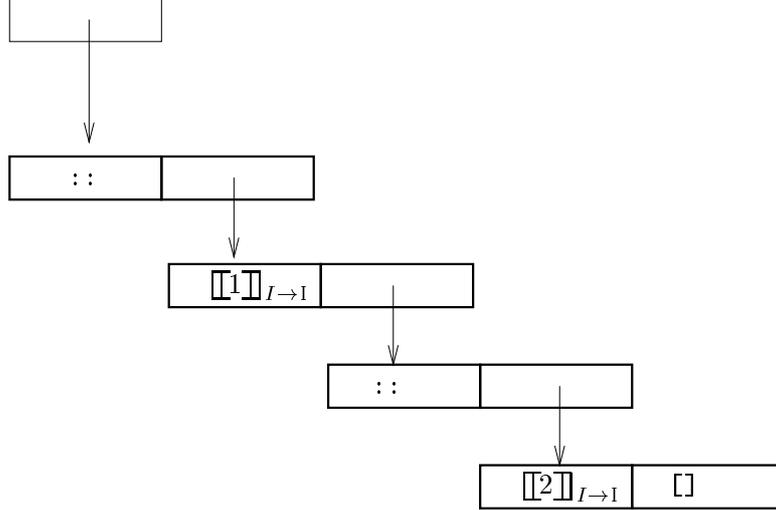
case e0 of Some x => e1
          | - => e2,

```

as it is not possible to determine at compile-time whether e_0 evaluates to a unary or a nullary constructed value.

(Notice the distinction between constructors and constructed values: The representation of the nullary constructor `None` is the word $[[\text{None}]]_{C \rightarrow I}$; the representation of the *constructed value* that to which `None at ρ` evaluates is a pointer to a word in memory containing $[[\text{None}]]_{C \rightarrow I}$.)

We choose this simple representation of constructed values to limit our job. Constructors can be represented more efficiently by using specialised representations for special situations: The indirection to the constructed value can be avoided if the **datatype**-declaration only contains nullary constructors. If pointers can be distinguished at run-time from integers representing constructors, all nullary constructed values can be represented without the extra indirection. The illusion of constructors with more than one argument is obtained by putting the arguments in a tuple and then applying the constructor to this tuple; e.g., the representation of `::(1, ::(2, []))` is



This gives another indirection: the pointer to the tuple. This indirection can be eliminated by having a special representation of constructed values that contain tuples. There are many more special cases that allow for more efficient representations. Some are petty optimisations, but, for instance, the one last mentioned implies that lists, which are used extensively in functional languages, will be represented significantly better. Cardelli (1984) implemented specialised representations of constructors in his ML compiler.

The code for the constructs that build constructed values is quite like the code for the 2- and 1-tuples (p. 28):

$$\begin{aligned}
 \phi := \boxed{\text{code to evaluate } \hat{c}_1 \ e_2 \ \text{at } \rho} &= \\
 \phi_t := \boxed{\text{the address of 2 new cells in } \rho} &; \\
 \phi_1 := \boxed{[[\hat{c}]]}_{C \rightarrow I} ; m[\phi_t + 0] := \phi_1 &; \\
 \phi_2 := \boxed{\text{code to evaluate } e_2} ; m[\phi_t + 1] := \phi_2 &; \\
 \phi := \phi_t, &
 \end{aligned}$$

and

$$\begin{aligned}
 \phi := \boxed{\text{code to evaluate } \hat{c} \ \text{at } \rho} &= \phi_t := \boxed{\text{the address of 1 new cell in } \rho} ; \\
 \phi_1 := \boxed{[[\hat{c}]]}_{C \rightarrow I} ; m[\phi_t + 0] := \phi_1 &; \\
 \phi := \phi_t. &
 \end{aligned}$$

The code for the consumer constructs of constructed values inspects the first and second component, respectively: The code for

$$\text{case } e_0 \text{ of } c_1 \Rightarrow e_1 \mid \dots \mid c_n \Rightarrow e_n \mid _ \Rightarrow e_{n+1}$$

jumps to the code for the proper sub-expression. For instance:

$$\begin{aligned}
\phi := \boxed{\begin{array}{l} \text{code to evaluate} \\ \text{case } e_0 \text{ of } \mathbf{Sex} \Rightarrow e_1 \\ \quad \quad \quad | \mathbf{Drugs} \Rightarrow e_2 \\ \quad \quad \quad | _ \Rightarrow e_3 \end{array}} &= \phi_0 := \boxed{\text{code to evaluate } e_0} ; \\
&\phi_? := m[\phi_0 + 0] ; \quad \text{fetch constructor} \\
&\text{if } \phi_? = \llbracket \mathbf{Sex} \rrbracket_{C \rightarrow I} \text{ then } \iota_1 \text{ else } \bar{\iota}_1 ; \\
&\bar{\iota}_1 : \text{if } \phi_? = \llbracket \mathbf{Drugs} \rrbracket_{C \rightarrow I} \text{ then } \iota_2 \text{ else } \bar{\iota}_2 ; \\
&\iota_1 : \phi := \boxed{\text{code to evaluate } e_1} ; \text{goto } \check{\iota} ; \\
&\iota_2 : \phi := \boxed{\text{code to evaluate } e_2} ; \text{goto } \check{\iota} ; \\
&\bar{\iota}_2 : \phi := \boxed{\text{code to evaluate } e_3} ; \text{goto } \check{\iota} ; \\
&\check{\iota} : \epsilon .
\end{aligned}$$

The code to extract the argument from a (unary) constructed value is exactly the code for **#1** e_2 :

$$\begin{aligned}
\phi := \boxed{\text{code to evaluate } \downarrow e_2} &= \phi_2 := \boxed{\text{code to evaluate } e_2} ; \\
&\phi := m[\phi_2 + 1] \quad \text{fetch argument.}
\end{aligned}$$

4.4 Regions

This section discusses what code to generate to introduce a region, discharge a region, put data in a region, and how to implement region variables and region size variables at run-time. We follow (Birkedal et al., 1996) in all.

Introducing and discharging regions

The code for **letregion** ρ **in** e_1 must introduce a new region before the code for e_1 , and discharge the region again afterwards.

Although regions are introduced and discharged in a last-in-first-out order, they cannot all be allocated on the stack, because the size of some regions cannot be determined at compile-time. This is dealt with by only allocating memory on the stack for regions with *known* size; for regions with *unknown* size, memory is allocated in the heap.

1. If ρ has *unknown* size, i.e., it has the form $\rho:?$, a heap region must be created and bound to it. The $\phi_\rho := \text{letregion}$ -instruction creates a new region in the heap and assigns to ϕ_ρ a pointer to a *region descriptor*, the data structure necessary for administrating allocations in a region in the heap. This pointer uniquely identifies the region. The *endregion*-instruction is used to deallocate all memory allocated for data in the most recently created region in the heap. Thus the code for **letregion** $\rho:?$ **in** e_1 is:

$$\begin{aligned}
\phi := \boxed{\text{code to evaluate } \text{letregion } \rho: ? \text{ in } e_1} \\
&= \phi_\rho := \text{letregion} ; \\
&\quad \phi := \boxed{\text{code to evaluate } e_1} ; \\
&\quad \text{endregion.}
\end{aligned}$$

2. If ρ has *known* size, i.e., it has the form $\rho:i$, a region with known size must be allocated. The memory needed for this region (i words) can be allocated on the stack when the region is introduced:

$$\begin{aligned} \phi := & \boxed{\text{code to evaluate } \text{letregion } \rho:i \text{ in } e_1} \\ = & \phi_{\text{sp}} := \phi_{\text{sp}} + \llbracket i \rrbracket_{I \rightarrow I} ; \\ & \phi := \boxed{\text{code to evaluate } e_1} ; \\ & \phi_{\text{sp}} := \phi_{\text{sp}} - \llbracket i \rrbracket_{I \rightarrow I}. \end{aligned}$$

Region variables and putting data in regions

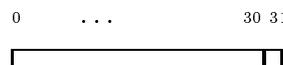
This section discusses how to identify regions at run-time, what the code to put data in a region should be, and how to decide whether a given region has known or unknown size.

To uniquely identify regions at run-time, a *region name* is assigned to each region: 1. The region name of a region with unknown size is the address of its region descriptor. 2. The region name of a region with known size is the address of the memory allocated for it on the stack.

A *put point* is an expression that has an “**at** ρ ”-annotation. The code for this expression will put some data in the region bound to ρ . To put ι words in a region, a register, ϕ , must first be set to point to an area in memory where ι words can be stored. Suppose ϕ_ρ contains the region name of the region. There are two cases: 1. If the region has unknown size, ϕ_ρ points to its region descriptor, and then ι words can be allocated with the instruction $\phi := \text{at } \phi_\rho : \iota$. 2. If the region has known size, ϕ_ρ points to memory that has already been allocated for it on the stack, i.e., ϕ can be set to point to the ι words with the instruction $\phi := \phi_\rho$.

How is it decided what the size of the region bound to ρ is at a specific put point? 1. If ρ has the form $\rho:?$, the regions bound to it will always have unknown size. 2. If ρ has the form $\rho:i$, the regions bound to it will always have known size. 3. If ρ has the form $\rho:\psi$, it can be bound to both regions with known size and regions with unknown size; it must be checked at run-time what the size of the region is. Thus, the region name of a region does not, in general, provide enough information to put data in the region; the region size (“known” or “unknown”) will also be necessary, if the region can be bound to a region variable that has the form $\rho:\psi$.

The information necessary at run-time to put data in a region is thus a pair consisting of the region name and the region size. How is this pair represented at run-time? A region size can be represented in one bit. A region name is a word-aligned address (i.e. an integer divisible by 4 on the PA-RISC), hence not all bits of a word are necessary to hold a region name. Therefore, the region-name-and-size pair can be squeezed into one word:



The region size is held in the least significant bit, and the region name is

held in the most significant bits.

If the representation of the pair is in ϕ_ρ , the size of the region can be checked with the instruction `if $\phi_\rho.lsb$ then ι_{unknown} else ι_{known}` , where *lsb* is the number of the least significant bit (31 on a PA-RISC). Extracting the region name from ϕ_ρ amounts to setting the least significant bit in ϕ_ρ to zero.

Thus, the code to put ι words into ρ , according to the form of ρ , is:

1. $\phi := \boxed{\text{the address of } \iota \text{ new cells in } \rho: ?}$ = $\phi_\rho := \boxed{\text{code to access } \rho: ?}$;
 $\phi := \text{at } \phi_\rho : \iota$
2. $\phi := \boxed{\text{the address of } \iota \text{ new cells in } \rho: i}$ = $\phi := \boxed{\text{code to access } \rho: i}$.
3. $\phi := \boxed{\text{the address of } \iota \text{ new cells in } \rho: \psi}$ =
 $\phi_\rho := \boxed{\text{code to access } \rho: \psi}$; if $\phi_\rho.lsb$ then ι_{unknown} else ι_{known} ;
 $\iota_{\text{known}} : \phi := \phi_\rho$; goto \checkmark ;
 $\iota_{\text{unknown}} : \phi := \text{at } \phi_\rho : \iota$; goto \checkmark ; $\checkmark : \epsilon$.

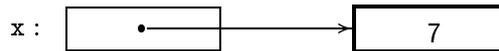
Here it is assumed $\phi := \text{at } \phi_\rho : \iota$ extracts the region name from the word in ϕ_ρ , i.e., it is not necessary to generate code to explicitly set the least significant bit in ϕ_ρ to zero. Furthermore, it is assumed that $\phi_\rho := \text{letregion}$ will set the region size bit appropriately in the word it returns in ϕ_ρ .

4.5 References

The construct `ref e_1 at ρ` creates a reference to the value that e_1 evaluates to. We represent a reference at run-time as the address of a memory cell. Thus, in the declaration

```
let x = ref 7 at r1
in e0,
```

the value bound to **x** is the address of a memory cell where the representation of 7 is stored:



Recall that a fat-edged box is a memory cell, while a box with thin edges is a word. The representation of the reference above can be thought of as being “the left-hand box and the arrow”. Since an address is one word, this representation of references is uniform.

Thus, the code for `ref e1 at ρ` must allocate a new memory cell in ρ , and return the address of that memory cell. It is exactly the code for the expression `(e1) at ρ` (p. 28):

$$\begin{aligned} \phi := \boxed{\text{code to evaluate ref } e_1 \text{ at } \rho} &= \\ &\phi_t := \boxed{\text{the address of 1 new cell in } \rho}; \\ &\phi_1 := \boxed{\text{code to evaluate } e_1}; \\ &m[\phi_t + 0] := \phi_1; \\ &\phi := \phi_t. \end{aligned}$$

Notice that only one memory cell is allocated: the values a reference can point to use uniform representation.

A reference may be *dereferenced* using the `!`-operator: evaluating `!x` yields 7. In general, the code for `!e2` return the word in the memory cell at the address that `e2` evaluates to. It is the same as the code for `#0 e2` (p. 28):

$$\begin{aligned} \phi := \boxed{\text{code to evaluate } !e_2} &= \phi_2 := \boxed{\text{code to evaluate } e_2}; \\ &\phi := m[\phi_2 + 0]. \end{aligned}$$

The memory cell addressed by a reference can be *updated*: evaluating `x:=42` has the side effect that the memory cell addressed by the reference now contains the representation of 42:



The reference itself never changes—`x` is bound to the exact same value; it is the word in the memory cell that changes.

In general, `e1 := e2` is evaluated by evaluating `e1` to an address of a memory cell and then updating this memory cell to the value to which `e2` evaluates:

$$\begin{aligned} \phi := \boxed{\text{code to evaluate } e_1 := e_2} &= \phi_1 := \boxed{\text{code to evaluate } e_1}; \\ &\phi_2 := \boxed{\text{code to evaluate } e_2}; \\ &m[\phi_1 + 0] := \phi_2. \end{aligned}$$

This code should return the 0-tuple, which is represented by any word (p. 27). Therefore it is not necessary to put any specific value in ϕ .

4.6 Functions

In this section we discuss functions that are built by λ -abstractions, `λy.e0 at ρ`, and the corresponding (normal) application construct, `e1 e2`. `letrec`-functions is the subject of the next section.

In a higher-order functional language, a function is a value that must be represented at run-time. Below, we discuss what information a *function*

value must contain and how it is represented, and then we can develop the code for application and λ -abstraction. (We use the term “function value” to hint at the similarities with other kinds of values. A function value is often called a *closure*.)

What information must a function value contain?

It is not generally known at compile-time which function will be applied at a given application. For instance, two different functions may be called by the code for the application

```
(if  $e_0$  then  $\lambda y_1.y_1+v$  at  $r_2$  else  $\lambda y_2.y_2-v-w$  at  $r_2$ ) 7.
```

Therefore, a function value must in some cases contain the code for the function.

Moreover, having functions as values means that a function may be applied in another environment than the one it is created in. In

```
let f = let a = 97 in
        let b = 98 in
        let g =  $\lambda y.a+y$  at r103
        in g
in f 1,
```

the function $\lambda_y = \lambda y.a+y$ at $r103$ is created in an environment that binds a to 97 and b to 98, but it is applied in an environment, where a and b are not defined. In this situation, λ_y is said to *escape*. Therefore, a function value must comprise not just the code for the function but also the values of the free variables of the function, a in this case (Landin, 1964).

Thus, a function value is a tuple where the first component is the code for the function and the remaining components are the values of the free variables of the function. E.g., the first λ -abstraction in the *if*-expression above would evaluate to the function value (“code for $\lambda y_1.y_1+v$ at r_2 ”, “value of v ”), while the second might evaluate to the function value (“code for $\lambda y_2.y_2-v-w$ at r_2 ”, “value of v ”, “value of w ”). (The order of the values of the free variables might be different.)

More efficient implementations of functions

This expensive way of implementing functions is necessary in some cases, because functions are values and might escape. In the average higher-order functional program, though, many (most?) of the functions are not passed around as values; they are just called like their counterparts in, e.g., Pascal (Welsh and Elder, 1982). When a function is only used “as a Pascal function”, it can be implemented in a cheaper fashion, because the function value need

not be represented explicitly. If, for instance, the example above read

```
let a = 97 in
let b = 98 in
let h =  $\lambda y.a+y$  at r103
in h 1,
```

λy would not escape, i.e., all applications of λy would be in the scope of its declaration, so λy would only be applied in an environment where all its free variables are available, and the pointer to the code for λy would be known at compile-time, since it is the only function that can be applied at the application. Thus, it would not be necessary at all to have an explicit function value for λy at run-time.

Different definitions of what it means that a function is only used “as a Pascal function” can be given. One such is: the identifier to which the function is bound may only appear as e_1 in applications $e_1 e_2$ (or f in $f \vec{e}_2$), only in the scope of the free variables of the function, and not under any λ . In this case, neither a code pointer, nor the free variables of the function are necessary, and therefore it is unnecessary to build a closure.

More sophisticated, in

```
let c = 99 in
let k = let d = 100 in
        let k =  $\lambda y.c+y+d$  at r103
        in k
in k 1,
```

λy is passed as a value and applied in an environment where its free variable d is not defined. Thus, we must have a function value at run-time, but it need not contain the free variable c , as it is available when λy is applied, and it need not contain the code pointer, as λy is the only function that can be applied.

The ORBIT Scheme compiler (Kranz et al., 1986) pioneered the specialised implementation of different kinds of functions. The closure representation analysis of Shao and Appel (1994) can “allocate closures in registers”, and functions can share closures. On average, the code generated by SML/NJ is 17% faster with this closure representation. Although both ORBIT and SML/NJ are continuation-based compilers, and this work therefore is not immediately applicable in our translation, it should be possible to adapt their methods.

Wand and Steckler (1994) present a transformation that transforms a λ -expression to *closure-passing style*, a λ -expression where closures are manipulated explicitly. Function application is replaced by closure application: the first component of the closure (a function—the “code pointer”) is invoked on the actual argument and the closure itself. They call their closure conversion *selective*, because not all applications are converted to closure applications; for instance, h above would not be represented by a closure in their

scheme, because it does not escape. Some of their closures are *lightweight*: they do not contain all the free variables of the function; for instance, the closure for k above would not contain c , because it is available at all the places where the closure might be invoked. They have not implemented the closure conversion, and it is unclear to us how fast it is (they do not explicitly give an algorithm).

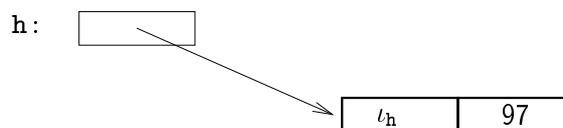
Closure representation analysis is a subject in itself and beyond the scope of this project. We will treat all functions in a uniform manner and always build closures for functions. This decision has serious implications for the register allocation: there will be no allocation of variables to registers across functions, because the free variables of a function must always be fetched from its closure in memory. This removes a whole layer of complexity from the register allocation, for now the free variables of the function can be regarded as *local* values that are fetched from the closure, and then we only have to worry about putting values local to a function in registers. This limitation thus reduces the complexity of doing inter-procedural register allocation by forcing the register allocator to be less inter-procedural.

Representing function values at run-time

Function values must use uniform representation at run-time, and we represent a function value like we represent other tuples (section 4.2): as the address of the actual representation of the tuple.

The constituents of the function value must themselves use uniform representation. The uniform representation of the code for the function is a label (which will fit in one word), and the actual representation is then the code labelled with this label.

The representation of h will look like this, assuming the label of its code is ι_h :



The code for an application

Now we sketch the code for an application, which we will call the *linking code*. At first, assume the application is not a tail call. The left-to-right evaluation order and call-by-value semantics of SML specifies that the code for an application $e_1 e_2$ must first evaluate e_1 and then e_2 . The former evaluates to a function value. The latter evaluates to the argument, which must be passed to the code for the function.

When the code for the function is executed, it must have access to the values of the free variables, which are recorded in the function value. Therefore,

we must also pass these to the code for the function.

Furthermore, we must pass to the function a label to which it should return.

Thus, an application can be viewed as a “goto with three parameters”: argument, tuple of values of free variables, and return label. (We call them *parameters* to avoid confusion with *argument*. The parameters of $\lambda y. \mathbf{a} + \mathbf{b} - y$ at $\mathbf{r1}$ are: its *argument* (the value to be bound to y), a tuple of the values of its free variables (the values of \mathbf{a} and \mathbf{b}), and a return label.) The linking code sets up the parameters by putting them on the stack or in appropriate registers:

1. set up closure
2. set up argument
3. set up return label
4. jump
5. return code.

We will not discuss now how to decide whether a parameter is passed on the stack or in a register and in which register, and when registers must be saved and restored. This is part of the discussion of register allocation in the following chapters.

Notice that by having the code in the first component of function values, the code part of a function value can always be accessed in the same manner, and that is necessary because the code for an application must call the code for different function values. For instance, the code for the application (if \dots) 7 above does not know whether it is a two- or a three-tuple that is applied; it simply extracts the code as the first component of the tuple.

To describe the linking code more specifically, assume that the parameters are passed in three different registers: the argument in $\phi_{\text{arg.}}$, the closure in $\phi_{\text{clos.}}$, and the return label in $\phi_{\text{ret.}}$, and that the result is passed in $\phi_{\text{res.}}$. The linking code is:

1. $\phi_{\text{clos.}} := \boxed{\text{code to evaluate } e_1}$;
2. $\phi_{\text{arg.}} := \boxed{\text{code to evaluate } e_2}$;
3. $\phi_{\text{ret.}} := \iota$;
4. $\phi_t := \mathbf{m}[\phi_{\text{clos.}} + 0]$; goto ϕ_t ;
5. $\iota : \phi := \phi_{\text{res.}}$,

where ϕ_t is some temporary register; ι is the label of the code the function should return to; and ϕ is the register for the result.

Some comments on this linking code: We simply pass the closure to the code for the function body and not just the sub-tuple containing the values of the free variables. It could be done simply by adding one to $\phi_{\text{clos.}}$, but that would cost an extra instruction and there is no reason to do it.

It is best to have part 1 and 2 of the linking code before 3, for if 3 was before 1 and 2, the register $\phi_{\text{ret.}}$ could not be used for other values in the code for e_1 and e_2 .

Like a function call can be viewed as a “goto with three parameters”, the return can be viewed as a “goto with one parameter”. In this sense, a return is similar to a call. This similarity is explicit when using continuation-passing-style as SML/NJ does (Appel, 1992). An application of a continuation is a “goto with parameters”.

Tail calls

In case the application is a tail call, there is no return code (5), and the code to set up the return label (3) is instead

$$\phi_{\text{ret.}} := \boxed{\text{code to access the return label of the current function}}.$$

Space forbids a discussion of what conditions make an application qualify as a tail call. Only, note that an application which is a tail call in the original SML program need not be a tail call in our source language. For instance, the application in `fn y => e1e2` is a tail call, but the region analyses may transform it to `λy. letregion r1 in e'1e'2 at r2`, in which the application would not immediately be considered a tail call because `r1` must be discharged *after* the application.

Functions of several arguments

In SML, functions always have one argument. Functions that appear to take several arguments may be implemented as functions that take a tuple as argument. Consider the SML function with “two arguments”:

```
fun sumacc (0,n) = n
  | sumacc (m,n) = sumacc (m-1, m+n).
```

Using the general method of generating code for tuples (explained in section 4.2), `sumacc` builds a tuple before each recursive call, and consumes the tuple again when performing the pattern match. Only the components of the tuple are used; the tuple itself is never needed. The generated code would be more efficient if we could somehow avoid building the tuple, and instead pass the individual components of the tuple to `sumacc`: Therefore, we try to convert a function that takes a tuple as its argument to a *function of several arguments*. Instead of having *exactly* one argument, functions then have *at least* one argument. This gives the following general code to set up the arguments for a function application:

$$\begin{aligned} \phi_{\text{arg-1}} &:= \boxed{\text{code to evaluate the first argument}} \\ &\vdots \\ \phi_{\text{arg-n}} &:= \boxed{\text{code to evaluate the } n\text{'th argument}} \end{aligned}$$

Section 7.5 discusses when a tuple function can be converted to a function of several arguments.

It is not always possible to pass all arguments in registers, for the set of registers is finite while there is no bound on the number of elements in a tuple. How to handle that is discussed in greater detail when we discuss register allocation of function application in section 8.9.

The code for a λ -abstraction

Now that we have decided that a function value must contain the label of the code and the value of the free variables, we can present the code for building a closure. A closure is a tuple, so the code to build it is much like the code for $(e_1, \dots, e_n) \text{ at } \rho$. Assume the free variables of $\lambda y. e_0 \text{ at } \rho$ are v_1, \dots, v_n :

$$\begin{aligned} \phi := & \boxed{\text{code to evaluate } \lambda y. e_0 \text{ at } \rho} = \\ & \phi_t := \boxed{\text{the address of } n + 1 \text{ new cells in } \rho} ; \\ & \phi_{\text{label}} := \iota_\lambda ; \quad \mathfrak{m}[\phi_t + 0] := \phi_{\text{label}} ; \\ & \phi_{v_1} := \boxed{\text{code to access } v_1} ; \quad \mathfrak{m}[\phi_t + 1] := \phi_{v_1} ; \\ & \quad \vdots \\ & \phi_{v_n} := \boxed{\text{code to access } v_n} ; \quad \mathfrak{m}[\phi_t + \llbracket n \rrbracket_{I \rightarrow I}] := \phi_{v_n} ; \\ & \phi := \phi_t, \end{aligned}$$

where ι_λ is the label of the code for (the body of) $\lambda y. e_0 \text{ at } \rho$.

4.7 Recursive and region polymorphic functions

Region polymorphic functions (or, synonymously, **letrec**-functions) are much like the functions discussed in the previous section. Our line of attack is to isolate the differences. The strategy is still to keep things as simple as possible. Thus, we will build closures for **letrec**-functions, for exploiting that a **letrec**-function never can be applied in an environment where its free variables are not available would complicate our job, as we could not simply fetch free variables from a closure. We start with region polymorphic application, and then discuss the representation of **letrec**-functions.

Differences between region polymorphic and normal application

Region polymorphic application $f \vec{\rho} e_2$ differs from normal application in three respects. Consider the linking code (1–5) (p. 38).

First, the code to set up the closure (1) is different. For a normal application $e_1 e_2$, it is $\phi_{\text{clos.}} := \boxed{\text{code to evaluate } e_1}$. In a region polymorphic application, there is no e_1 that evaluates to a closure; the closure we want to pass to the function is the closure for the **letrec**-function named f . So the code is “ $\phi_{\text{clos.}} := \text{somehow get the closure for the function named } f$ ”. What that is more exactly, we will discuss below.

Second, there are more arguments. For a **letrec**-function there are also the arguments ρ_1, \dots, ρ_k , which too must be passed to the function. Say

this is to be done in the registers $\phi_{\rho_1}, \dots, \phi_{\rho_k}$, then the code to set up the arguments is

$$\begin{aligned} \phi_{\text{arg-1}} &:= \boxed{\text{code to evaluate the first argument}} ; \\ &\vdots \\ \phi_{\text{arg-}n} &:= \boxed{\text{code to evaluate the } n\text{'th argument}} ; \\ \phi_{\rho_1} &:= \boxed{\text{code to access } \rho_1} ; \\ &\vdots \\ \phi_{\rho_k} &:= \boxed{\text{code to access } \rho_k} \end{aligned}$$

Third, the code to jump to the code for the applied function (4) is different. At a normal application, the label of the code for the body of the function must be extracted from the closure, because it is not known at compile-time what function will be applied, i.e., the code is

$$\phi_t := \text{m}[\phi_{\text{clos.}} + 0] ; \text{goto } \phi_t.$$

At a region polymorphic application, it is known at compile-time what function is applied—namely the one named f —and hence we know what label must be jumped to. Assuming this label is ι_f , the code to jump is `goto ι_f` .

For both kinds of applications, the code to set up the return label (3) is the same ($\phi_{\text{ret.}} := \iota$) and so is the return code (5) ($\iota : \phi := \phi_{\text{res.}}$). Tail calls are dealt with as they are dealt with at normal applications.

The first difference, how to pass the closure of the `letrec`-function, touches upon the issue of how to represent `letrec`-functions at run-time, which is treated now.

Representing `letrec`-functions at run-time

The relevant situations are captured by this example:

```
letrec f1 y1 = f1 v + f2 y1 + w
      f2 y2 = f1 y2 + f2 u + w at r17
in f1 1 + f2 2
```

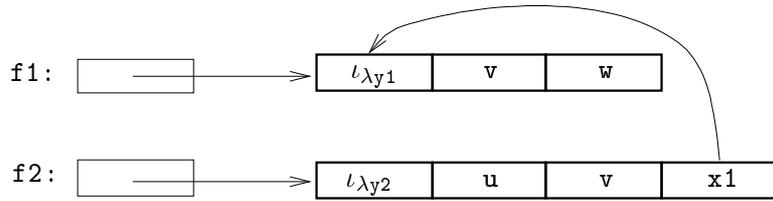
Two `letrec`-functions are declared. Each calls itself and the other recursively.

To see how to represent `letrec`-functions at run-time, consider a similar example with λ -abstractions instead of `letrec`-functions:

```
let x1 =  $\lambda y1.v+y1+w$  at r17 in
let x2 =  $\lambda y2.x1 y2 + u+w$  at r17
in x1 1 + x2 2
```

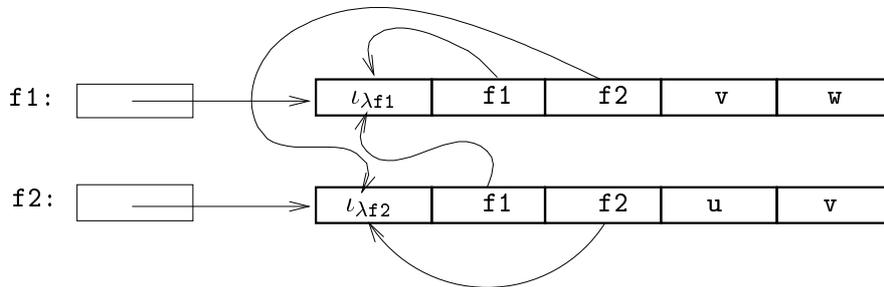
In the following, λ_{f_1} and λ_{f_2} denote the two `letrec`-functions (e.g., $\lambda_{f_1} = f1 y1 = f1 v + f2 y1 + w$), and λ_{x_1} and λ_{x_2} denote the λ -abstractions (e.g., $\lambda_{x_1} = \lambda y1.v+y1+w$ at r17). With `let`-expressions there can be no

recursion, because the x bound in `let $x = e_1$ in e_2` cannot be referenced in e_1 . Therefore, it is only λ_{x_2} that applies λ_{x_1} in the second example. According to the discussion about λ -abstractions above, the expression is evaluated thus: First a closure is built for λ_{x_1} and bound to \mathbf{x}_1 ; then a closure is built for λ_{x_2} and bound to \mathbf{x}_2 . As λ_{x_1} is applied in λ_{x_2} , \mathbf{x}_1 is a free variable in λ_{x_2} , and therefore the closure for λ_{x_2} contains the closure for λ_{x_1} . This will be represented this way:



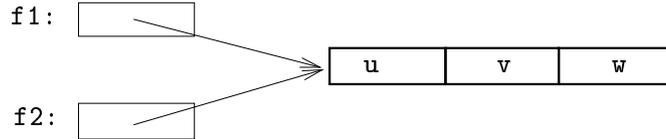
The representation of the second closure has a link to the representation of the first closure, reflecting that the second function intends to apply the first function. At the application `\mathbf{x}_2 \mathbf{y}_2` the closure that is passed is the one bound to \mathbf{x}_2 .

This suggests the following way of implementing the `letrec`-functions above. Treat `f1` and `f2` as *variables* that are bound to the closure for the function they refer to; e.g., `f1` is bound to the closure for λ_{f_1} . Getting the closure for the function named f when a region polymorphic function is applied is then an access to the variable f ; i.e., number 1 in the linking code above is `$\phi_{\text{clos.}} := \text{code to access } f$` . Then `f1` and `f2` are free variables of both λ_{f_1} and λ_{f_2} . Thus, both closures should contain the values bound to `f1` and `f2`; e.g., the closure for λ_{f_1} should contain itself (the value bound to `f1`) and the closure for λ_{f_2} (the value bound to `f2`). The representation would look like:



The circularity in the representation corresponds to the recursion in the program. Each closure points to itself reflecting that each function might apply itself, and each closure points to the other reflecting that each function might apply the other.

The code for `letrec $b_1 \dots b_m$ at ρ in e_{m+1}` should build the closures for the `letrec`-functions and bind them to the `letrec`-bound variables, and then it should evaluate e_{m+1} . As the closures are all built at the same time, (Birkedal, 1994) suggests building one *shared closure* for all functions in the `letrec`-expression. The representation, which is explained below, will be



Now `f1` and `f2` will both be bound to the shared closure. Their values are omitted from the shared closure, because there is no point in saving a couple of pointers to the shared closure in the same shared closure; i.e., we no longer consider `f1` and `f2` free variables of λ_{f1} and λ_{f2} .

As region polymorphic applications know the label of the code for the applied function, it is not necessary to have code pointers in the shared closure. E.g., at the region polymorphic application `f1 v`, it will always be λ_{f1} that is applied. The shared closure could never flow to a (normal) application, and thus make it impossible to syntactically identify what function is being applied, because names f of `letrec`-functions can only appear in the construct $f \vec{\rho} e_2$. An example like

```
let x = letrec f y = e_f at r119
      in f
in x 120
```

where it is actually `f` that is applied at `x 120` is not possible.

Note also that the common free variable `w` of λ_{f1} and λ_{f2} is now stored in memory only once.

This shared closure representation of `letrec`-functions saves space and simplifies things. For instance, it is easier to build a shared closure than the two circularly linked closures above.

The code for `letrec $b_1 \dots b_m$ at ρ in e_{m+1}` is thus

```
code to build the shared closure for  $b_1, \dots, b_m$  ;
 $\phi :=$  code to evaluate  $e_{m+1}$ ,
```

where the shared closure for b_1, \dots, b_m is simply a tuple containing all the free variables of the functions $b_1 \dots b_m$ (excluding the names of the `letrec`-functions $b_1 \dots b_m$).

We decided to treat `letrec`-function names f as variables. How are these variables accessed? We say f_1, \dots, f_m are *siblings*, if they are bound in the same `letrec`-expression:

$$\begin{aligned} \text{letrec } f_1 \vec{\rho}_1 y_1 = e_1 \\ \vdots \\ f_m \vec{\rho}_m y_m = e_m \text{ at } \rho \text{ in } e_{m+1} \end{aligned}$$

Accesses to any of f_1, \dots, f_m within e_1, \dots, e_{m+1} yield the same value, viz. the shared closure for the **letrec**-functions with the names f_1, \dots, f_m . Therefore, there is no reason to distinguish between accesses to, e.g., f_1 and f_m .

Assume $f_i \in \{f_1, \dots, f_m\}$. Consider a region polymorphic application $f_i \vec{\rho} e'$ directly within e_{m+1} (i.e. within e_{m+1} , but not within the body of any function in e_{m+1}). The code for this application will access f_i , and this access to f_i should yield the shared closure that has been built by the code preceding the code for e_{m+1} .

Now consider a region polymorphic application $f_i \vec{\rho} e'$ directly within $e_j \in \{e_1, \dots, e_m\}$. We call such an occurrence of f_i a *sibling access*. This access will occur in the code for the **letrec**-function named f_j , and while executing that code, the current closure will be the same shared closure that the access to f_i should yield, i.e., within the code for the **letrec**-function named f_j , the code $\phi := \boxed{\text{code to access } f_i}$ is simply $\phi :=$ the current shared closure.

Finally, consider occurrences of f_i that are within, but not directly within, one of e_1, \dots, e_m , i.e. within another function within one of e_1, \dots, e_m . In **letrec** $\mathbf{f} \mathbf{y} = \lambda \mathbf{y}' . \mathbf{f} \mathbf{y}'$ **at** $\mathbf{r1}$ **at** $\mathbf{r2}$ **in** e_2 , \mathbf{f} is a free variable of $\lambda \mathbf{y}' . \dots$, i.e., the application $\mathbf{f} \mathbf{y}'$ is not a sibling access, \mathbf{f} must be fetched from the closure. The code for the function named \mathbf{f} must access \mathbf{f} to put it into this closure. In that sense, there is a sibling access to \mathbf{f} *directly within* the body for the function named \mathbf{f} .

4.8 Exceptions

We call the run-time object that handles an exception a *handler*, a concept we shall postpone defining until we have discussed how **raise** e_1 and e_1 **handle** $a \Rightarrow e_2$ work.

The stack of active handlers

A handler belonging to e_1 **handle** $a \Rightarrow e_2$ should only handle exceptions raised while evaluating e_1 . E.g., the handler in

```
(17 handle a => 7) + raise a at r119
```

should not handle anything; it is only active while 17 is evaluated and not when \mathbf{a} **at** $\mathbf{r119}$ is raised. At run-time, we must maintain a set of active handlers. The code for e_1 **handle** $a \Rightarrow e_2$ must install in the set of active handlers a new handler that will handle a -exceptions. This handler should be installed before e_1 is evaluated, and it should be discarded if evaluating e_1 did not activate it. Thus, the active handlers can be kept in a stack, which we will call the *stack of active handlers*. (Do not confuse this stack with the K stack (the stack in the language K), or with the *stack of regions*.)

Using $a \Rightarrow e_2$ to denote the handler belonging to e_1 **handle** $a \Rightarrow e_2$,

$$\phi := \boxed{\text{code to evaluate } e_1 \text{ handle } a \Rightarrow e_2} = \text{handler-push } a \Rightarrow e_2 ;$$

$$\phi := \boxed{\text{code to evaluate } e_1} ;$$

$$\text{handler-pop.}$$

Raising an exception and propagating a raised exception

At **raise** e_1 , the exception value that e_1 evaluates to must be passed to the topmost handler on the stack of active handlers that can handle the exception value. Thus,

`((raise a' at r119) handle a=>1) handle a'=>2) handle a'=>3`

should evaluate to 2, for when **a' at r119** is raised, the stack of active handlers will be (starting with the topmost handler):

a=>1
a'=>2
a'=>3

and the topmost handler capable of handling **a' at r119** is **a'=>2**. In other words, when an exception is raised, handlers are popped from the stack of active handlers until a handler capable of handling the exception is found. We decide that the responsibility of finding the proper handler lies with the code for the handlers: **raise** e_1 simply pops the topmost handler and passes it the exception value. This handler is then expected to *deal with* the exception value. A handler *deals with* an exception value that it cannot *handle* by raising it again; this way the responsibility of dealing with an exception value propagates to the next handler on the stack of active handlers. This strategy can be illustrated in SML for the example above:

```
exception A and A';
(((raise A') handle A=>1
  | x =>raise x) handle A'=>2
  | x =>raise x) handle A'=>3
  | x =>raise x.
```

Here, each handler deals with any exception it does not “really” want to handle by explicitly handling it and raising it again.

Raised exceptions that are handled by no handler should be reported as uncaught exceptions and make the program terminate. This is done by having a top-level handler that handles any exception. We will not describe this part of the implementation of exceptions further.

As it is not known at compile-time which handler will handle a raised exception value, all handlers and code to raise an exception value must agree

on how this exception value is passed. We designate some register ϕ_{raised} to pass the value in. The code to raise an exception will have the form:

$$\phi := \boxed{\text{code to evaluate raise } e_1} = \phi_{\text{raised}} := \boxed{\text{code to evaluate } e_1};$$

handler-pop h ;
jump to the code for h .

In addition, two things must be done when an exception is raised. Consider

```
(letregion r17 in
  (let insignif = (b,b) at r17 in
    (if b then raise a at r119 else 1)+5)) handle a => 6.
```

The region introduced by the `letregion`-expression must be discharged again. If `a at r119` is raised, control will flow directly to the handler, skipping the rest of the code for the `letregion`-expression, including the code to discharge that region. Therefore, the code for `raise a at r119` must explicitly discharge the region. In general, after `e_1 handle $a \Rightarrow e_2$` has been evaluated, the stack of regions must be as it was right before `e_1 handle $a \Rightarrow e_2$` was evaluated, regardless of whether the handler introduced by `e_1 handle $a \Rightarrow e_2$` was activated or not. Therefore, `raise e_3` must reset the stack of regions to its state before the topmost handler was pushed, i.e., it must discharge regions introduced since the topmost handler was pushed.

Also the K stack must be reset: above, if the variable `insignif` was pushed on the K stack, it must be popped when `a at r119` is raised.

Thus, the code to raise an exception must be

$$\phi := \boxed{\text{code to evaluate raise } e_1} =$$

$\phi_{\text{raised}} := \boxed{\text{code to evaluate } e_1};$
 handler-pop h ;
 reset the stack of regions and the K stack
 to what they were before h was pushed ;
 jump to the code for h .

Note that e_1 must be evaluated before the handler is popped from the stack of active handlers, for e_1 might itself raise an exception which should be handled by the handler on the top of the stack of active handlers (as in `raise (raise x)`), and e_1 must also be evaluated before the stacks are reset, because the evaluation might need some of the values that will be deallocated.

Resetting stacks at a raise

It is easy to reset the K stack at a raise: Simply reset ϕ_{sp} to the value it had before the last handler was pushed on the stack of active handlers:

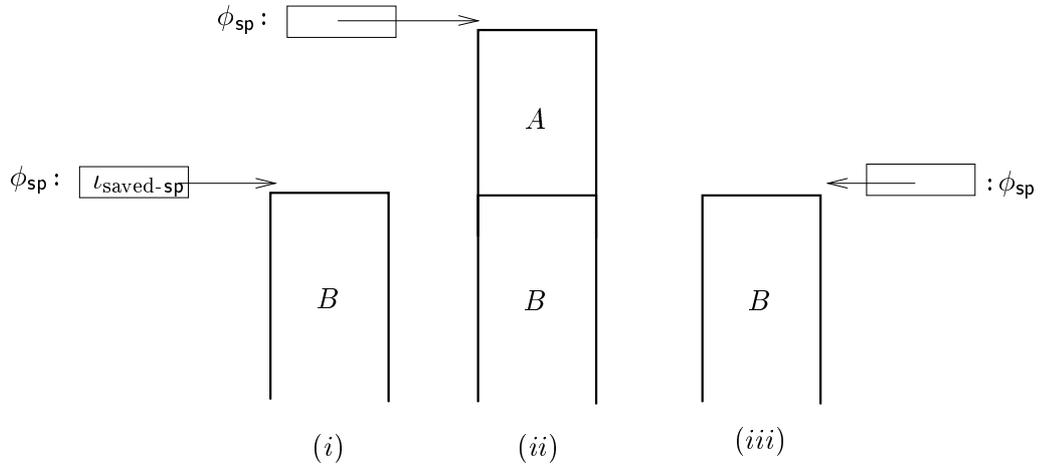


Fig. 14. *The K stack.* ϕ_{sp} always points to the next *free* word. (i) just before the topmost handler is pushed; (ii) just before an exception is raised; (iii) just after the exception is raised.

This will also discharge any regions of *known* size, since they are allocated on the K stack.

Then remain the regions of *unknown* size, which are allocated in the heap. These regions have *region descriptors* allocated in the *A*-part of the K stack. Assuming all region descriptors are in a linked list, we can run through this list and discharge (using *endregion*) all regions whose region descriptor is in the *A*-part of the stack, i.e., those whose region descriptor is above $\iota_{\text{saved-sp}}$. Therefore, it is possible to detect what heap regions should be discharged at a raise given $\iota_{\text{saved-sp}}$ of the handler on the top of the stack of active handlers. We will not discuss in further detail how this operation can be implemented; it is provided by our intermediate language in form of the instruction *endregions* ϕ , where ϕ must contain $\iota_{\text{saved-sp}}$.

Summing up, the only information necessary to reset the (K and region) stacks appropriately when an exception is raised is the contents, $\iota_{\text{saved-sp}}$, of ϕ_{sp} before the handler on top of the stack of active handlers was pushed. If $\phi = \iota_{\text{saved-sp}}$, the K stack can be reset and the appropriate known-size regions be discharged with the instruction $\phi_{sp} := \phi$, and the appropriate regions of unknown size can be discharged with *endregions* ϕ .

The concept handler

Now we can define the concept handler. A handler is the information needed by the code for **raise** e_1 to do the actual handling and reset the stacks, i.e., it is a pair $(\iota_{\text{saved-sp}}, \iota_{\text{handler-code}})$, where $\iota_{\text{saved-sp}}$ is the contents of ϕ_{sp} when the handler on top of the stack of active handlers was pushed, and $\iota_{\text{handler-code}}$

is the label of some code that deals with an exception. Thus, the code for `e_1 handle $a \Rightarrow e_2$` is

$$\begin{aligned} \phi := & \boxed{\text{code to evaluate } e_1 \text{ handle } a \Rightarrow e_2} = \\ & \text{handler-push } (\phi_{\text{sp}}, \iota_{a \Rightarrow e_2}) ; \\ & \phi := \boxed{\text{code to evaluate } e_1} ; \\ & \text{handler-pop,} \end{aligned}$$

where $\iota_{a \Rightarrow e_2}$ is the label of the code for the handler `$a \Rightarrow e_2$` , and the code for `raise e_1` is:

$$\begin{aligned} \phi := & \boxed{\text{code to evaluate raise } e_1} = \\ & \phi_{\text{raised}} := \boxed{\text{code to evaluate } e_1} ; \\ & \text{handler-pop } (\phi_1, \phi_2) ; \\ & \text{endregions } \phi_1 ; \\ & \phi_{\text{sp}} := \phi_1 ; \\ & \text{goto } \phi_2. \end{aligned}$$

Implementing the stack of active handlers

A handler is represented by a *handler element*, which contains $\iota_{\text{saved-sp}}$ and $\iota_{\text{handler-code}}$ for the handler it represents. We implement the stack of active handlers as a linked list of handler elements. A global variable, h , points to the topmost handler element.

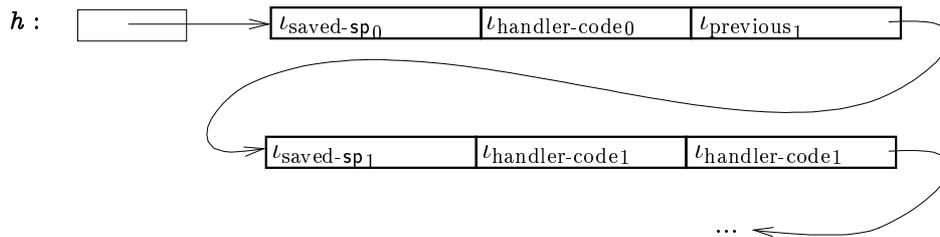


Fig. 15. *The list of active handlers.*

Memory for this linked list can be allocated in the K stack, if the pushes and pops of handlers are interleaved with other K stack pushes and pops in such a way that the stack discipline will be upheld. This turns out to be the case, as will be apparent after reading this chapter.

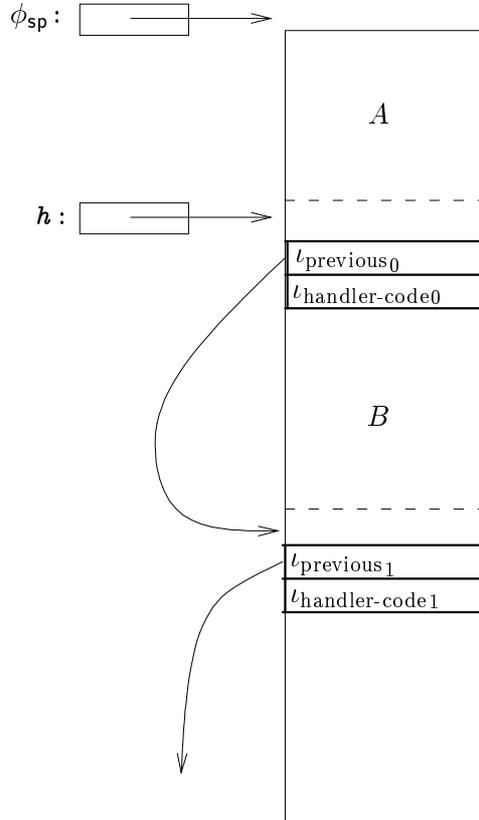


Fig. 16. *The stack of active handlers allocated on the K stack.*

Notice that $l_{\text{saved-sp}}$ is not in the handler elements in figure 16: The position on the K stack of the handler element itself indicates what $l_{\text{saved-sp}}$ is. That $l_{\text{saved-sp}}$ need not be explicitly saved is an extra bonus from allocating memory for the stack of active handlers in the K stack

With this implementation of the stack of active handlers, we can give a more specific description of the code for `raise e_1` . If an exception is raised with the stack situation in figure 16, h indicates how much must be peeled off the stack. The heap regions with region descriptors above h are discharged with the `endregions`-instruction, and ϕ_{sp} is reset to h , effectively discarding the A -part of the K stack. Then, h is reset to l_{previous_0} , thereby popping the handler on the top of the stack of active handlers. After that, control flows

to the code labelled $\iota_{\text{handler-code0}}$:

$$\begin{aligned} \phi := & \boxed{\text{code to evaluate } \mathbf{raise} \ e_1} = \\ & \begin{aligned} & \phi_{\text{raised}} := \boxed{\text{code to evaluate } e_1} ; \\ & \text{endregions } \mathbf{h} ; \\ & \phi_{\text{sp}} := \mathbf{h} ; \\ & \text{pop } \mathbf{h} ; \\ & \text{pop } \phi' ; \\ & \text{goto } \phi'. \end{aligned} \end{aligned}$$

$\left. \begin{array}{l} \\ \\ \\ \end{array} \right\}$

$\left. \begin{array}{l} \text{reset stacks} \\ \\ \end{array} \right\}$

$\left. \begin{array}{l} \\ \\ \end{array} \right\}$

$\left. \begin{array}{l} \text{handler-pop} \\ \end{array} \right\}$

The code for $e_1 \mathbf{handle} \ a \Rightarrow e_2$ creates a new handler element on the K stack, links it to the list of handler elements, evaluates e_1 , and then it takes the handler element out of the linked list and removes it from the K stack:

$$\begin{aligned} \phi := & \boxed{\text{code to evaluate } e_1 \ \mathbf{handle} \ a \Rightarrow e_2} = \\ & \begin{aligned} & \text{push } \iota_{a \Rightarrow e_2} ; \\ & \text{push } \mathbf{h} ; \\ & \mathbf{h} := \phi_{\text{sp}} ; \\ & \phi := \boxed{\text{code to evaluate } e_1} ; \\ & \text{pop } \mathbf{h} ; \\ & \text{pop}. \end{aligned} \end{aligned}$$

$\left. \begin{array}{l} \\ \\ \\ \end{array} \right\}$

$\left. \begin{array}{l} \text{handler-push} \\ \\ \end{array} \right\}$

$\left. \begin{array}{l} \\ \\ \end{array} \right\}$

$\left. \begin{array}{l} \text{handler-pop} \\ \end{array} \right\}$

(Exactly what the code is depends on how we implement the global variable \mathbf{h} , which is discussed when we discuss register allocation (section 8.10). For instance, “push \mathbf{h} ” may not correspond exactly to a push-instruction, which is the reason we use “push” instead of “push”.)

Exceptions are generative. Exception names

Despite the name, exception constructors are more like **let**-bound variables than like constructors. At run-time, an exception constructor is a *variable* bound to an *exception name*, whereas a constructor c is a *constant*, viz. the word $\llbracket c \rrbracket_{C \rightarrow I}$. As with constructors, notice the distinction between an *exception name* (the value bound to \mathring{a}) and a *nullary exception value* (the value that \mathring{a} **at** ρ evaluates to).

The expression **exception** a **in** e_2 is akin to a **let**-expression: each time it is evaluated, a *fresh* exception name is bound to a . The following example (stolen from (Birkedal and Welinder, 1993)) serves to illustrate the behaviour

of `exception a in e2`:

```
letrec strange y =
  exception a in
  if y=0 then raise a at r0
    else ((strange (y-1)) handle a => 0)
  at r0
in strange 2.
```

The exception raised in the recursive call `strange (y-1)` will never be handled, because the exception name of the handler is of a newer vintage than the exception value that is raised: In each recursive call of `strange`, the exception constructor `a` is declared anew, and bound to a different exception name each time.

This perhaps strange semantics of exceptions was designed to ensure that different exception declarations that accidentally use the same identifier for the exception constructor will indeed give rise to different exceptions (Milner and Tofte, 1991, p. 19). It has the by-effect that we must bind a different exception name to the same exception constructor every time it is declared, although it is declared by the same declaration every time.

We represent an exception name as a word, and keep track of what exception names have been used with a global variable `n`. Here is a sketch of the code for `exception a in e2`:

$$\begin{aligned} \phi := & \boxed{\text{code to evaluate exception } a \text{ in } e_2} = \\ & n := n + 1 ; \\ & \text{bind } a \text{ to } n \text{ in the environment ;} \\ & \phi := \boxed{\text{code to evaluate } e_2}. \end{aligned}$$

Representing exception values

Exception values are much like constructed values as illustrated below. Among other things, we must use a uniform representation of them, as they can be used like any other values.

$$\begin{aligned}
\phi := & \boxed{\text{code to evaluate } \overset{\circ}{a} \text{ at } \rho} = \\
& \phi_t := \boxed{\text{the address of 1 new cells in } \rho} ; \\
& \phi_{\delta} := \boxed{\text{code to access } \overset{\circ}{a}} ; m[\phi_t + 0] := \phi_{\delta} ; \\
& \phi := \phi_t.
\end{aligned}$$

The code for a handler

The code for the handler $a \Rightarrow e_2$ must decide whether it can handle the raised exception. If it can, it evaluates e_2 ; otherwise it raises the exception again:

code for the handler $a \Rightarrow e_2$ =

$$\begin{aligned}
\iota_{a \Rightarrow e_2} : \phi_a := & \boxed{\text{code to access } a} ; \\
& \phi_{a?} := m[\phi_{\text{raised}} + 0] ; \\
& \text{if } \phi_a = \phi_{a?} \text{ then } \iota \text{ else } \bar{\iota} ; \iota : \phi := \boxed{\text{code to evaluate } e_2} ; \text{goto } \check{\iota} ; \\
& \bar{\iota} : \text{code to raise the exception again.}
\end{aligned}$$

The handler puts the exception name bound to a in ϕ_a and the exception name of the raised exception into $\phi_{a?}$. Remember that the code for a **raise**-expression puts the raised exception value in ϕ_{raised} (p. 46).

If the handler can handle the raised exception, it evaluates e_2 , and then it jumps to $\check{\iota}$, the code for the expression e_1 **handle** $a \Rightarrow e_2$ that it belongs to. Thus, the code for e_1 **handle** $a \Rightarrow e_2$ (p. 50) must end with the label $\check{\iota}$.

After e_1 **handle** $a \Rightarrow e_2$ has been evaluated, the exception a may or may not have been raised, and correspondingly, the result of the expression may stem from e_1 or from e_2 , and thus, the code for e_1 and e_2 must use the same destination register, i.e., ϕ above must be the same as the ϕ in the code for e_1 **handle** $a \Rightarrow e_2$ (p. 50).

The “code to raise the exception again” is the same as the code for a **raise**-expression except that ϕ_{raised} already contains the raised exception value.

Comparison with other approaches

A raise is a transfer of control and of a value, i.e., it is a “goto with argument”, i.e., an application of a continuation. Therefore, it is particularly nice to implement exceptions in a continuation-passing style compiler such as SML/NJ (Appel, 1992). There is always a *current continuation* that should be applied to the result of the expression being computed. The current handler (the topmost on the stack of active handlers) is implemented as a *handler continuation*, and all raise does is to apply the handler continuation instead of the current continuation to the exception value.

The existing intermediate-code generator, COMPILE-LAMBDA, implements handlers much like SML/NJ's handler continuation: A handler is implemented as a *handler function* that takes the raised exception as argument. The code for e_1 **handle** $a \Rightarrow e_2$ builds a closure for the handler function but also saves on the stack the label that the handler function should return to. The code for **raise** e_1 "applies" the current handler function to the value e_1 evaluates to, except that the return label it gives the handler function is the label that was saved when the handler function was created.

A raise is like a function application in that it is a transfer of control and of a value, but it is unlike a function application in that it does not return. Therefore it is the creator of the handler function (the code for e_1 **handle** $a \Rightarrow e_2$), and not its caller (the code for **raise** e_1), which knows the return label.

Our approach is better for two reasons: First, COMPILE-LAMBDA's handler function will always be applied in an environment where its free variables are available, and thus it is unnecessary to build a closure for a handler function. Second, it is unnecessary to explicitly manipulate a return label, as the code for a handler always "returns" to the same program point, namely the program point just after the expression e_1 **handle** $a \Rightarrow e_2$ it belongs to.

These improvements may have no significance in practice: the overhead in COMPILE-LAMBDA's explicit manipulation of return labels should be small; and although we avoid building a closure for a handler function, the variables used in the handler will often have to be saved on the stack anyway.

A region inference based compiler has to control deallocation of memory, and a raise may make it necessary to deallocate some memory, while a garbage collection based compiler (as SML/NJ) can simply rely on the garbage collector to clean up. The method we use to deallocate at a raise (including the idea of putting region descriptors on the K stack and having the *endregions* instruction) is due to Lars Birkedal and Mads Tofte, who used it in COMPILE-LAMBDA.

5 Inter-procedural register allocation

Our inter-procedural register allocation can be divided in two: the inter-procedural strategy (for processing the functions of the program), and the per-function strategy. This chapter discusses the former; the latter is discussed in the next chapter.

We discuss the purpose of register allocation (section 5.1), in particular the goals of making it inter-procedural (section 5.2), our approach to inter-procedural register allocation (section 5.3), how to exploit inter-procedural information (sections 5.4 and 5.5). Then we discuss the problems in implementing our inter-procedural strategy (sections 5.6–): building the call graph (section 5.7), how to have individual linking conventions for functions (section 5.8), and deal with recursion (sections 5.9 and 5.10). Then the overall algorithm for the inter-procedural strategy is summarized (section 5.11). We conclude with a comparison with other approaches to inter-procedural register allocation (section 5.12).

The inter-procedural part of our algorithm, discussed in this chapter, is developed in detail in chapter 7.

5.1 Why register allocation?

Register allocation is probably the single most important thing when compiling to a RISC. Its purpose is to reduce the number of load, store and move instructions.

If the time to execute an operation on the RISC is one unit, each operand that is not in a register and has to be loaded from memory before performing the operation will add at least one unit to the execution time. If, furthermore, the result of the operation has to be stored in memory, the execution time will be increased by at least one unit more. Register allocation will not in general be able to eliminate all memory traffic, but even a simple register allocation will cut down memory traffic significantly. In the existing back end, KAM, for the ML Kit, there is a near 50% reduction in execution time when comparing code that has been register allocated with code that loads the operands from memory before each operation and stores the result afterwards (Elsman and Hallenberg, 1995, p. 40). This is even without retaining values in registers across basic block boundaries.

We will try to reduce the number of register-to-register moves by trying to make the producer of a value place the value in the same register as the consumer wants the value in. It is difficult to say much about the relative importance of reducing the number of loads and stores, and eliminating register-to-register moves. For instance, on the PA-RISC, a load takes one clock cycle to execute plus one cycle to transfer the value from the cache to a register (assuming a cache hit, for now). If the instruction immediately after the load needs the loaded value, it must wait for one cycle until the value arrives. Thus, a load takes one or two clock cycles, where a move always takes only one. So generally, eliminating a load should be preferred to

eliminating a move, but the importance of eliminating moves should not be underestimated. George and Appel (1995) report a surprisingly big speedup of 4.4% solely from eliminating moves, and this is even an improvement over an algorithm that already tries to eliminate moves.

Traditionally, to simplify the intermediate code generation phase of compilers, this phase has been allowed to generate many move instructions, relying on the ensuing register allocation to eliminate them, thus making it necessary for this phase to deal with moves. Alternatively, the code generation can be made smarter (and more complex) and produce fewer moves. In any case, making code that has as few moves as possible is desirable.

The considerations above assume that the loaded value is in the cache. If there is a cache miss, the load will take around 20 clock cycles instead of one to two (Andersen, 1995). Considering the negligible price of a load when there is a cache hit compared to the penalty for a cache miss, should not the goal of register allocation be to reduce the number of cache misses rather than the number of load and move instructions? We believe not. Since there is more room for data in the cache than in the registers, it should not be possible to reduce the number of cache misses using the registers: any value that is used often enough to be eligible for allocating to a register will also be in the cache, assuming the cache is completely associative. Of course, caches are never completely associative, and hence, values that are used often may accidentally be thrown out, but predicting this at compile-time is out of the question. The conclusion is that it is not the job of the register allocator to avoid cache misses: when doing register allocation, we assume that all values are in the cache, and that the price for a load hence is one or two clock cycles. This assumption increases the relative importance of eliminating moves compared to that of eliminating loads.

5.2 Why inter-procedural?

In this section we discuss the merits of inter-procedural register allocation and the benefits we expect from it.

We assume functions are smaller in programs in functional languages than in imperative ones, and function calls are more frequent at run-time. This is partly a programming style imposed on the programmer by the language (e.g., to loop, one must make a recursive function), and partly, it is our experience that programmers tend to program this way in functional languages.

This implies that it is more important that function calls are implemented efficiently in a functional language. One way to do that is with inter-procedural register allocation, for it allows individual function calls to be implemented in specialised, efficient ways, rather than in the same general, inefficient way.

The second assumption—that functions are generally smaller—also renders inter-procedural register allocation more important: If the code for each function in many cases only needs a small part of the available registers, an intra-procedural register allocation algorithm will not be able to exploit the registers fully.

5.3 Our approach to inter-procedural register allocation

We see two approaches to inter-procedural register allocation: the *truly inter-procedural* approach allocates registers for the whole program at once, whereas the *per-function inter-procedural* approach extends a basically intra-procedural register allocation to an inter-procedural one by doing the register allocation on one function at a time but exploiting inter-procedural information while doing so. The following considerations lead us to choose the second approach.

Clearly, a truly inter-procedural algorithm could give a better result than algorithms using the more restricted, per-function approach, but it is not clear how such an algorithm could be devised.

Moreover, a truly inter-procedural algorithm might easily be computationally costly for big programs. It is, e.g., out of the question to build one big interference graph for the whole program and do graph colouring on it (Steenkiste, 1991, pp. 41–42).

If we wanted to allocate free variables of functions to registers across functions, a truly inter-procedural approach would be necessary. But recall from section 4.6 that we have confined ourselves from doing this by deciding that the free variables of a function are passed to it in a tuple in memory every time the function is applied. Consequently, when a function body is evaluated, the free variables are fetched from this tuple and not from the current environment. This isolation of functions from the environment they are applied in harmonises with the per-function approach.

Allocating free variables of functions to registers across functions is easier in languages that do not have functions as values. As discussed in section 4.6, in Pascal, the free variables of a function need not be kept in a closure, for they will always be available in the environment where the function is applied. This is because each free variable of a function f is a *local* variable in some other function that has been called and has not returned yet, and therefore, the free variable is accessible to f . Hence, in Pascal, allocating the free variables of f to registers amounts to allocating local variables of other functions to registers across the calls to f . In C (Kernighan and Ritchie, 1988), the problem is conceptually even simpler: a variable is either free in all functions (global) or local to a single function.

Furthermore, truly inter-procedural register allocation may be problematic with separate compilation, because not all functions are available. With the per-function approach, functions are processed one at a time, and functions in other modules will be no problem except that there is less inter-procedural information about them than about the functions inside the module. (Currently, our compiler has no separate compilation.)

In the following sections, we will discuss how a per-function register allocation can use knowledge about other functions. The basic idea in the algorithm is from (Steenkiste and Hennessy, 1989).

5.4 Exploiting information about registers destroyed by a function

One useful type of inter-procedural information is what registers are destroyed when a given function is applied. Consider the call graph

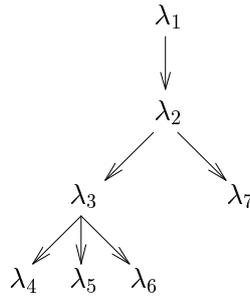


Fig. 19. A call graph. The λ 's are functions; λ_2 calls λ_3 and λ_7 , etc.

Suppose we are doing register allocation for λ_3 , and suppose we know what registers are destroyed by the calls in λ_3 to λ_4 , λ_5 , and λ_6 . Figure 20 illustrates how this knowledge can be used when allocating registers for the values used in λ_3 .

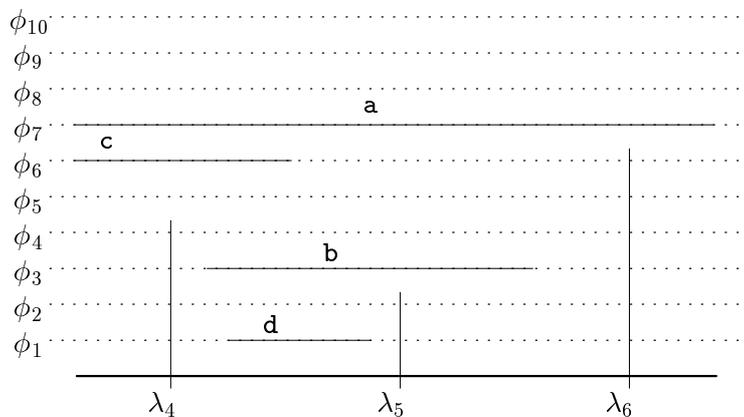
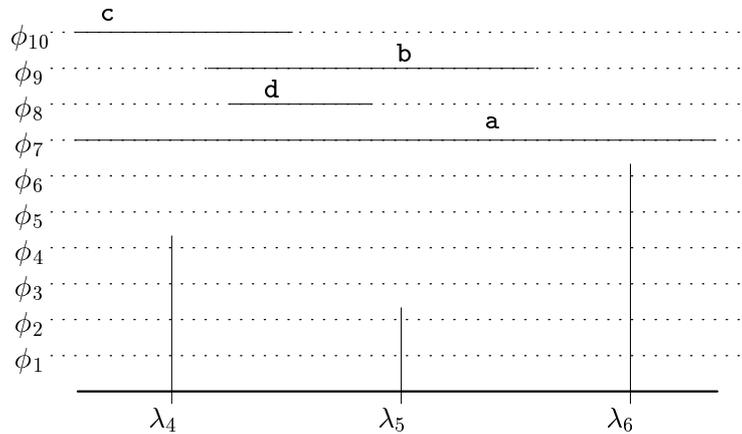


Fig. 20. Register allocation for λ_3 . The bottom line symbolises the code for λ_3 (we assume there are no jump instructions). The dotted lines represent registers ϕ_1 through ϕ_{10} . The vertical lines indicate the points in the code for λ_3 where λ_4 , λ_5 , and λ_6 , respectively, are called, and which registers will be destroyed by the call. E.g., λ_4 destroys registers ϕ_1 through ϕ_4 . The horizontal lines **a**, **b**, **c**, and **d** indicate live ranges of values.

We have tried fitting values **a**, **b**, **c**, and **d** into registers in a good way. For instance, it is better to put **b** in any of ϕ_3 through ϕ_{10} than in ϕ_1 or ϕ_2 , because the latter are destroyed while **b** is still live by the call to λ_5 , and then **b** would have to be saved elsewhere across that call.

The figure shows that λ_3 will destroy ϕ_1 through ϕ_7 , when it is called. The values could have been placed like this:



but then a call to λ_3 would destroy 10 registers, instead of 7. To minimise the total number of registers destroyed when λ_3 is called, it is best to use ϕ_1 through ϕ_6 in the body of λ_3 , because these registers will be destroyed anyway when λ_3 is applied.

The example shows that we should pursue two goals when choosing registers for values in a function:

- (i) If possible, put a value in a register that will not be destroyed by an application while the value is live, for then the register need not be saved across that application.
- (ii) When possible, use the register with the smallest number that will be destroyed anyway, thereby trying to minimise the total number of registers used by the function, and thus leaving more registers for the register allocation of any caller of this function. Using the register with the lowest number will “push the values into the clefts” between function calls. For example, in figure 20, **d** could have been placed in ϕ_4 , but this would not be as good a choice.

These goals are in order of priority, for if the priority was opposite, the register allocation would be too eager to reuse registers; it would never use more than one register.

Contrast this with what an intra-procedural register allocation will do. A uniform convention for all functions will tell what registers are destroyed by a call. For instance, the convention might be that ϕ_1 through ϕ_5 are destroyed when any function is called. Then the picture would be

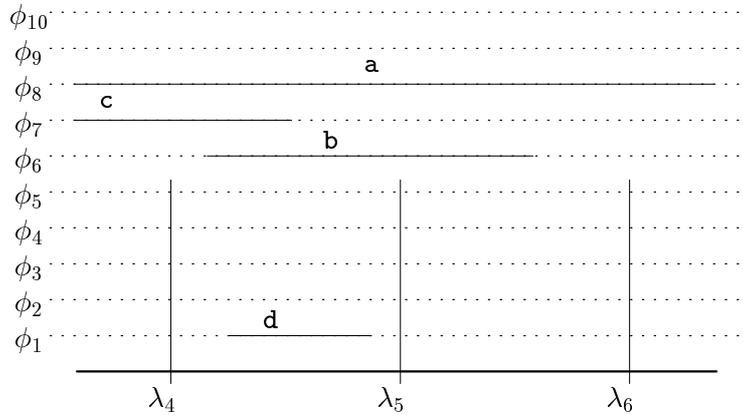


Fig. 21. Register allocation for λ_3 when a uniform convention for which registers are destroyed at a function call is assumed.

A function that destroys registers other than ϕ_1 through ϕ_5 (e.g. λ_6) must save them. This more rigid scheme implies worse register allocation: λ_6 must save ϕ_6 , although it is not used by the caller, λ_3 . The value **b** cannot reside in ϕ_3 across the call to λ_5 , because the uniform convention says λ_5 destroys that register, although it does not. We must either save **b** in memory across the call to λ_5 or put it in one of the caller-saves registers, which must then in turn be preserved for λ_3 to comply to the convention.

5.5 Exploiting information about parameter registers

Another way inter-procedural information can be utilised is when parameters are passed to functions, and when functions return their result. By a *linking convention* for a function, we mean the information necessary to generate code to call that function, i.e., how should the argument be passed, etc. With individual linking conventions, different functions can receive their parameters and return their result in different registers. Thus, a function can receive its argument and return its result, etc. in the registers that suit it best, instead of having to move values to and from fixed registers. If, e.g., **c** in figure 20, p. 58, indicates the live range of the argument to λ_3 , it is best for λ_3 to receive its argument in another register than ϕ_1 through ϕ_4 , because these are destroyed by the call to λ_4 while **c** is still live.

Perhaps an even greater advantage with individual linking conventions is achieved alone from using *different* registers at different function calls, instead of always the same: Assume we use a uniform linking convention for all functions such that the closure is passed in $\phi_{clos.}$, the argument in $\phi_{arg.}$, the return label in $\phi_{ret.}$, and the result is returned in $\phi_{res.}$. The code for the expression $f(g\ 1) + h\ 2$ is in figure 22. The code to save and restore in that figure is necessary with a uniform linking convention. With individual linking conventions, it may sometimes be avoidable: Because the result from a function is always returned in the same register, the result of $f(g\ 1)$ must be saved elsewhere while $h\ 2$ is computed. Using individual linking

conventions, \mathbf{f} and \mathbf{h} may return their results in different registers, and the result of $\mathbf{f}(\mathbf{g} \ 1)$ can remain in its register, while $\mathbf{h} \ 2$ is computed (assuming that \mathbf{h} does not destroy that register). Likewise, with the uniform linking convention, \mathbf{f} and \mathbf{g} will both use $\phi_{clos.}$ to pass the closure in, and then the closure for \mathbf{f} must be saved elsewhere while \mathbf{g} is evaluated.

$\phi_{clos.} :=$	code to access \mathbf{f}	;	
	\langle save $\phi_{clos.}$ for \mathbf{f} somewhere		
	code to access \mathbf{g}	;	}
$\phi_{arg.} :=$	1	;	
$\phi_{ret.} :=$	l_1	;	
	code to jump to \mathbf{g}	;	
$l_1 :$	$\phi_{arg.} :=$	$\phi_{res.}$	
	$\phi_{ret.} :=$	l_2	
	\langle restore $\phi_{clos.}$ for \mathbf{f}		
	code to jump to \mathbf{f}	;	
$l_2 :$	\langle save $\phi_{res.}$ from \mathbf{f}		
	$\phi_{clos.} :=$	code to access \mathbf{h}	}
	$\phi_{arg.} :=$	2	
	$\phi_{ret.} :=$	l_3	
	code to jump to \mathbf{h}	;	
$l_3 :$	$\phi :=$	\langle restore $\phi_{res.}$ from call to \mathbf{f}	
	$\phi :=$	$\phi + \phi_{res.}$	

Fig. 22. A disadvantage with using a fixed linking convention. The code for $\mathbf{f}(\mathbf{g} \ 1) + \mathbf{h} \ 2$ assuming the closure is always passed in $\phi_{clos.}$, the argument in $\phi_{arg.}$, the return label in $\phi_{ret.}$, and the result is returned in $\phi_{res.}$. There is saving and restoring of values (the bracketed code), because the registers of the linking convention are “crowded”.

5.6 Design decisions conclusion

To conclude, we expect to profit on inter-procedural register allocation mainly through the information it gives about which registers are destroyed by the functions called. This will allow us to avoid unnecessary saving of registers across function calls, and it will enable us to reduce the total number of registers a function destroys (by trying to “put live ranges into the clefts (of figure 20) between function calls”). It is not clear that a uniform caller-save and callee-save convention will not be almost as good as inter-procedural register allocation in this respect. On the other hand, it is very cheap in space and time to collect and use this inter-procedural information with the method we propose. The strongest argument against making the register allocation inter-procedural is that it complicates the algorithm. We also expect to gain something from implementing function calls in individual ways, and not always use the same dedicated linking registers. In this respect, a uniform convention cannot compete.

The following sections explain in greater detail how the inter-procedural strategy presented above can be implemented, and discuss some problems.

This ends in a sketch of the overall algorithm for the inter-procedural strategy for translating E to K .

Generally, the functions we need inter-procedural information about when doing register allocation of a function are the functions it might call, i.e. its children in the call graph. So, if we do per-function register allocation of the children of a function before we do it for the function itself, the information will be available. For instance, with the call graph of figure 19, we would process (do register allocation and generate code for) λ_3 and λ_7 before λ_2 . In other words, we process the nodes in the call graph in bottom-up order. In the following we discuss how to build and process the call graph. As will be evident, the two main problems are recursion and the fact that functions are values.

5.7 Call graph

Functions appear in the program in two forms: as λ -abstractions, $\lambda y. e_0$ at ρ , and as **letrec**-functions, $f \rho y = e_0$. Therefore, we define the *set of functions*

$$\Lambda ::= \lambda Y. E \text{ at } P \mid F \vec{P} Y = E.$$

A call graph for a program is then a directed graph of functions $\lambda \in \Lambda$. To build a call graph, we need information about what functions might be applied at a given application. Obtaining this is more difficult in a higher-order functional language than in other languages because functions are values that flow in the program like any other types of values. For instance, in the program

```
let a =  $\lambda f. (\lambda x. f \ x \text{ at } r6)$  at r5
in
  a ( $\lambda y. y \text{ at } r71$ ) 23,
```

it is actually $\lambda y. y \text{ at } r71$ that is applied at the application $f \ x$.

To establish which functions may be applied at an application, a data flow analysis, called *closure analysis*, is necessary. Formally, a closure analysis translates a program $e \in E$ to a *lambda-annotated* program $\hat{e} \in \hat{E}$, where \hat{E} is defined by the same grammar as E , except that all applications have been annotated with a set $\lambda \subseteq \Lambda$ of functions that can be applied at that application:

$$\hat{E} ::= \hat{E}_\Lambda \hat{E} \mid F_\Lambda \vec{P} \hat{E} \mid \dots$$

where $\Lambda = \mathcal{P}\Lambda$. The lambda-annotated version of the program above is

```
let a =  $\lambda f. (\lambda x. f_{\{\lambda_y\}} x \text{ at } r6)$  at r5
in
  a_{\{\lambda_f\}} (\lambda y. y \text{ at } r71)_{\{\lambda_x\}} 23,
```

using the abbreviations

$$\begin{aligned} \lambda_x &= \lambda x. f \ x \text{ at } r6 \\ \lambda_f &= \lambda f. (\lambda x. f \ x \text{ at } r6) \text{ at } r5 \\ \lambda_y &= \lambda y. y \text{ at } r71. \end{aligned}$$

Our closure analysis is described in section 7.2.

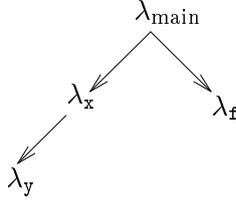
A *call graph* for a program, e , is a directed graph where the nodes are the functions of e and where there is an edge from one function to another, if the first might call the latter. In the program above, there is an edge from λ_x to λ_y . Function calls are not necessarily within a function, though. For instance, the application \mathbf{a} ($\lambda y.y$ at $\mathbf{r71}$) above, where λ_f may be applied, is not within any function. From which call graph node should the corresponding edge to λ_f then be? This problem is solved, when making the call graph, by pretending that the program is not simply e , but rather $\lambda_{\text{main}} = \lambda y_{\text{main}}.e$ at \mathbf{r}_{main} , where $y_{\text{main}} \in Y$ and $\mathbf{r}_{\text{main}} \in P$ are dummy identifiers that do not occur in e . Then, the applications that are not within any function in e will be inside λ_{main} ; e.g., the call graph for the example program will have an edge from λ_{main} to λ_f .

More formally, a call graph for e must be defined with respect to a closure analysis: It is a rooted, directed graph,

$$\gamma = (\mathcal{X}^{\text{cg}}, \mathcal{E}, \lambda_{\text{main}}) \in ? = \mathcal{P}\Lambda \times \mathcal{P}(\Lambda \times \Lambda) \times \Lambda,$$

whose nodes, \mathcal{X}^{cg} , are the functions of $\lambda_{\text{main}} = \lambda y_{\text{main}}.e$ at \mathbf{r}_{main} , and where there is an edge from λ_1 to λ_2 , i.e., $(\lambda_1, \lambda_2) \in \mathcal{E}$, iff the closure analysis says λ_1 might call λ_2 . The root node of the graph is λ_{main} .

It is trivial to build a call graph from a lambda-annotated program. The details can be found in section 7.6. A call graph for the example program is:



Finally, code is generated for the program by processing each λ of the call graph in bottom-up order; e.g., first λ_f , then λ_y , λ_x , and λ_{main} .

Here is a first try at the overall algorithm for translating an expression to intermediate code:

$$E \xrightarrow{\llbracket \cdot \rrbracket_{\text{ca}}} \hat{E} \xrightarrow{\llbracket \cdot \rrbracket_{\text{cg}}} ? \xrightarrow{\text{rdfs}} K.$$

The closure analysis, $\llbracket \cdot \rrbracket_{\text{ca}} \in E \rightarrow \hat{E}$, translates the program, e , to a lambda-annotated program, \hat{e} . From this, $\llbracket \cdot \rrbracket_{\text{cg}} \in \hat{E} \rightarrow ?$ builds the call graph. The nodes of the call graph are processed in *reverse depth-first search* order by $\text{rdfs} \in ? \rightarrow K$ to produce the code $\kappa \in K$ for the program. (For now, we assume there is no recursion in the program, and hence, the call graph will be acyclic, and can be processed in a reverse depth-first search. Section 5.9 explains how to deal with recursion.) In other words, the function to translate a program e to κ is

$$\llbracket \cdot \rrbracket_{\text{compile}} = \text{rdfs} \circ \llbracket \cdot \rrbracket_{\text{cg}} \circ \llbracket \cdot \rrbracket_{\text{ca}}.$$

During the bottom-up traversal, *rdfs* must remember inter-procedural information about the λ 's that have been processed thus far. For instance, after having processed λ_y it must be recorded which registers are destroyed by λ_y (as discussed in section 5.4 above), and what the linking convention for λ_y is (section 5.5). This is done in an environment, $\eta \in \mathbf{H}$ that maps λ 's to their inter-procedural information. What this environment exactly records is discussed below. The initial environment, η_0 , maps all λ 's to “unknown”. The function *rdfs* starts by processing λ_{main} , the root node of γ :

$$\text{rdfs } \gamma = \text{let } (\eta, \kappa) = \text{rdfs}_0 \lambda_{\text{main}} \gamma \eta_0 \text{ in } \kappa,$$

where *rdfs*₀ is a function of three arguments: λ_{cur} , the function currently being processed; γ , the call graph; and η , the current environment. *rdfs*₀ λ_{cur} processes the children $\{\lambda_1, \dots, \lambda_l\}$ of λ_{cur} before it processes λ_{cur} . Each λ is processed with $\llbracket \cdot \rrbracket_{\text{donode}}$, which takes λ_{cur} and the current environment, η , and returns the code, κ , for λ_{cur} and an environment, η' , updated with information for λ_{cur} :

$$\begin{aligned} \text{rdfs}_0 \lambda_{\text{cur}} \gamma \eta &= \text{let } \{\lambda_1, \dots, \lambda_l\} = \text{children } \lambda_{\text{cur}} \gamma \\ &\quad (\eta, \kappa_1) = \text{rdfs}_0 \lambda_1 \gamma \eta \\ &\quad \vdots \\ &\quad (\eta, \kappa_l) = \text{rdfs}_0 \lambda_l \gamma \eta \\ &\quad (\eta, \kappa) = \llbracket \lambda_{\text{cur}} \rrbracket_{\text{donode}} \eta \\ &\text{in } (\eta, \kappa ; \kappa_1 ; \dots ; \kappa_l). \end{aligned}$$

The environment passed to $\llbracket \lambda_{\text{cur}} \rrbracket_{\text{donode}}$ will contain the necessary inter-procedural information for the functions λ_{cur} might call, because they have been processed before λ_{cur} is processed.

In what follows, we shall modify and refine this sketch of the overall algorithm to take care of the problems encountered.

5.8 Linking convention

Different functions can be applied at the same application in a higher-order functional language. Consider the lambda-annotated expression

```
let a =  $\lambda \mathbf{f} . (\lambda \mathbf{x} . \mathbf{f}_{\{\lambda_{\mathbf{g}}, \lambda_{\mathbf{y}}\}} \mathbf{x}$  at r1) at r2 in
let b =  $\lambda \mathbf{g} . \mathbf{g}_{\{\lambda_{\mathbf{x}}, \lambda_{\mathbf{y}}\}} \mathbf{3}$  at r3 in
let i =  $\lambda \mathbf{y} . \mathbf{y}$  at r7
in ( $\mathbf{a}_{\{\lambda_{\mathbf{f}}\}} \mathbf{b}$ ,  $\mathbf{b}_{\{\lambda_{\mathbf{g}}\}} \mathbf{i}$ ,  $\mathbf{b}_{\{\lambda_{\mathbf{g}}\}} (\mathbf{a}_{\{\lambda_{\mathbf{f}}\}} \mathbf{i})$ ) at r5.
```

$\lambda_{\mathbf{g}}$ is shorthand for $\lambda \mathbf{g} . \mathbf{g} \ \mathbf{3}$ at r3, etc. At the application $\mathbf{f} \ \mathbf{x}$, both $\lambda_{\mathbf{y}}$ and $\lambda_{\mathbf{g}}$ may be applied. Since it is the same code that will be calling these functions (namely the code for $\mathbf{f} \ \mathbf{x}$), $\lambda_{\mathbf{y}}$ and $\lambda_{\mathbf{g}}$ must use the same linking convention. Since also $\lambda_{\mathbf{x}}$ and $\lambda_{\mathbf{y}}$ can be applied at the same application,

they too must use the same linking convention, and in effect, all three λ 's have to use the same linking convention.

Generally, if λ_1 and λ_2 may be applied at the same application, they must use the same linking convention. The relation “must use the same linking convention” on the set of λ 's in a program is an equivalence relation, and the set of λ 's in a program can be divided into *equivalence classes* of λ 's that must use the same linking convention. For the program above, there are two equivalence classes: $\{\lambda_f\}$ and $\{\lambda_x, \lambda_y, \lambda_g\}$.

When we process a function and decide the linking convention for it, we must ensure that all the functions in its equivalence class get the same linking convention. This implies that the linking convention for a function may be fixed before the function is processed. If, for instance, we process λ_y above first (and hence decide a linking convention for it), the linking convention for λ_x and λ_g will already be decided when we process them.

It follows that a linking convention is associated with an equivalence class of functions (and not with the individual function). The set of registers that will be destroyed when the code for a function is called, on the other hand, is individual to each function. Thus, the inter-procedural environment, η , consists of two maps: η^l maps an equivalence class of λ 's to its linking convention, and η^d maps an individual λ to the set of registers that will be destroyed when λ is called. Say λ_y and λ_x have been processed (in that order), and assume the linking convention decided for λ_y is l_y , that λ_y destroys the registers $\hat{\phi}_y$, and that λ_x destroys $\hat{\phi}_x$, then the inter-procedural environment will be $\eta = (\eta^l, \eta^d)$, where

$$\begin{aligned}\eta^l &= \{ \{\lambda_x, \lambda_y, \lambda_g\} \mapsto l_y, \quad \{\lambda_f\} \mapsto -lc \} \\ \eta^d &= \{ \lambda_x \mapsto \hat{\phi}_x, \quad \lambda_y \mapsto \hat{\phi}_y, \quad \lambda_f \mapsto \emptyset, \quad \lambda_g \mapsto \emptyset \},\end{aligned}$$

where “ $\lambda^{\equiv} \mapsto -lc$ ” means that the linking convention has not yet been decided for the equivalence class λ^{\equiv} , and “ $\lambda \mapsto \emptyset$ ” means that the set of registers that will be destroyed by a call to the code for λ is not yet known.

We modify *rdfs* from the previous section to find the set, $\lambda^{\equiv s} \in \mathcal{P}(\mathcal{P}\Lambda)$, of equivalence classes, $\lambda^{\equiv} \in \mathcal{P}\Lambda$, of functions in the program, and to set up the initial environment, η_0 , that is passed to *rdfs*₀ to map all equivalence classes to $-lc$ and all functions to \emptyset :

$$\eta_0 = (\{\lambda^{\equiv} \mapsto -lc \mid \lambda^{\equiv} \in \lambda^{\equiv s}\}, \{\lambda \mapsto \emptyset \mid \lambda \in \lambda^{cg}\}).$$

To find the set $\lambda^{\equiv s}$ of equivalence classes of functions, *rdfs* uses a union-find algorithm, $\llbracket \cdot \rrbracket_{uf} \in \hat{E} \rightarrow \mathcal{P}(\mathcal{P}\Lambda)$, described in section 7.9. Assuming γ has the form $(\lambda^{cg}, \mathcal{E}, \lambda_{main})$, *rdfs* becomes

$$\begin{aligned}rdfs \gamma &= \text{let } \lambda^{\equiv s} = \llbracket \lambda_{main} \rrbracket_{uf} \\ &\quad \eta_0 = (\{\lambda^{\equiv} \mapsto -lc \mid \lambda^{\equiv} \in \lambda^{\equiv s}\}, \{\lambda \mapsto \emptyset \mid \lambda \in \lambda^{cg}\}) \\ &\quad (\eta, \kappa) = rdfs_0 \gamma \lambda_{main} \eta_0 \\ &\text{in } \kappa.\end{aligned}$$

5.9 Dealing with recursion

If there is recursion (Koch and Olesen, 1996) in the program, there can be cycles in the call graph:

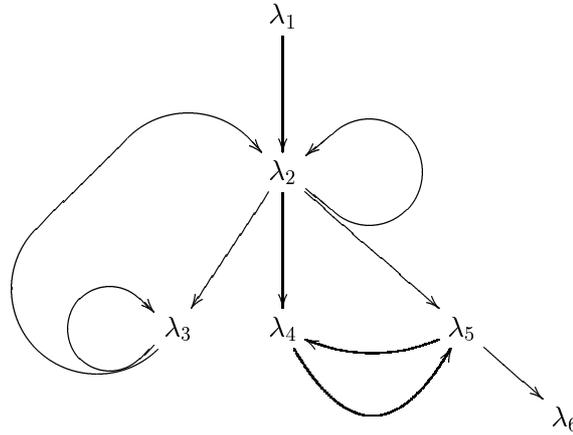


Fig. 23. A call graph with recursion. λ_4 and λ_5 are mutually recursive functions. λ_2 may call itself directly, and it may call λ_3 . Likewise, λ_3 may call itself directly, and it may call λ_2 .

A graph with cycles cannot be traversed bottom-up. We handle this by finding the *strongly connected components* of the call graph.¹ This gives us another graph in which the nodes are the strongly connected components of the call graph and there is an edge between two strongly connected components iff there is an edge in the call graph between a call graph node in the first strongly connected component and another one in the second:

¹A subset of the nodes of a graph is a strongly connected component iff it is the biggest subset such that there is a path from every node in the subset to every other node in the subset.

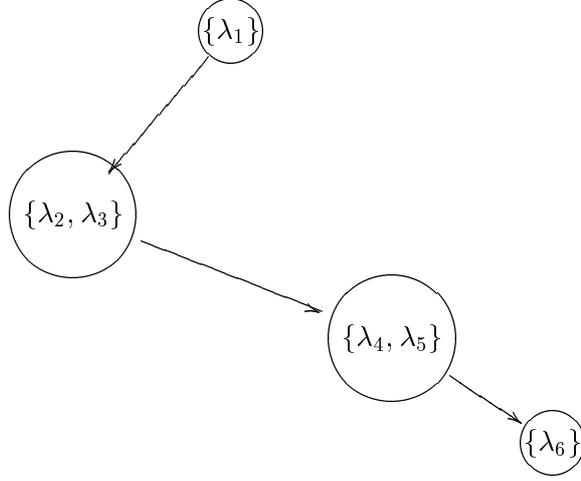


Fig. 24. The strongly connected components graph for the call graph in figure 23.

In contrast to the call graph, the strongly connected components graph will be acyclic, and thus, it can be processed bottom-up.

Like a call graph, a strongly connected components graph is a rooted, directed graph:

$$\gamma = (\boldsymbol{\lambda}^{\circ s}, \mathcal{S}, \boldsymbol{\lambda}_{\text{main}}^{\circ}) \in \Gamma = \mathcal{P}\mathbf{\Lambda} \times \mathcal{P}(\mathbf{\Lambda} \times \mathbf{\Lambda}) \times \mathbf{\Lambda},$$

where $\mathbf{\Lambda}$ is the set of strongly connected components, i.e., $\mathbf{\Lambda} = \mathcal{P}\mathbf{\Lambda}$. If the root node of the call graph is λ_{main} , the root node, $\boldsymbol{\lambda}_{\text{main}}^{\circ}$, of the corresponding strongly connected components graph will be $\{\lambda_{\text{main}}\}$.

We assume the function $sccs \in ? \rightarrow \Gamma$ will convert a graph γ to its strongly connected components graph γ , and we modify $rdfs$ to take a γ instead of a γ , such that

$$\llbracket \cdot \rrbracket_{\text{compile}} = rdfs \circ sccs \circ \llbracket \cdot \rrbracket_{\text{cg}} \circ \llbracket \cdot \rrbracket_{\text{ca}}.$$

Before, $rdfs$ processed each node λ of the call graph with $\llbracket \cdot \rrbracket_{\text{donode}} \in \mathbf{\Lambda} \rightarrow \mathbf{H} \rightarrow \mathbf{K} \times \mathbf{H}$ (p. 65); now it processes each strongly connected component $\boldsymbol{\lambda}^{\circ} \in \mathbf{\Lambda}$ with a similar function $do\text{-}scc \in \mathbf{\Lambda} \rightarrow \mathbf{H} \rightarrow \mathbf{K} \times \mathbf{H}$, which in turn uses $\llbracket \cdot \rrbracket_{\text{donode}}$ to process the λ 's of each strongly connected component.

5.10 Processing a strongly connected component

Recursion introduces other problems than cycles in the call graph. Consider the following figure:

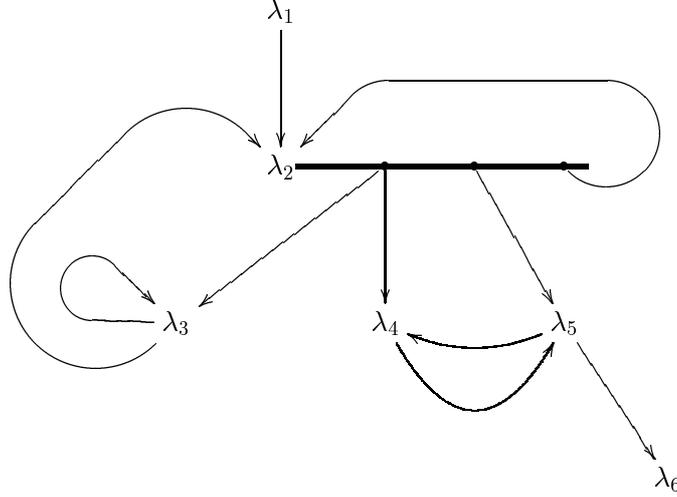


Fig. 25. *Zooming in on λ_2 from figure 23.* The fat line symbolises the code for λ_2 . The dots in the code are applications where λ_3 , λ_4 , λ_5 , and λ_2 are called. Here we have chosen that λ_3 and λ_4 may be applied at the same (first) application in λ_2 , while the application of λ_5 and the recursive application are each from their own application. The code for λ_2 could look differently given the call graph in figure 23.

In the following sections we shall assume we are processing λ_2 . We want to know which registers will be destroyed by the calls in λ_2 , i.e., we want to form a picture for λ_2 like that for λ_3 in figure 20 (p. 58).

Potentially recursive applications

We say an application $e_1 \lambda e_2$ or $f_\lambda \vec{p} e_2$ directly within a λ in a strongly connected component λ° is *potentially recursive* iff some λ' that may be applied at that application is in λ° , i.e., iff $\lambda \cap \lambda^\circ \neq \emptyset$. In figure 25, the first application (dot) in the code for λ_2 is potentially recursive, since a function (λ_3) that is in the same strongly connected component as the caller (λ_2) may be applied (i.e., $\{\lambda_2, \lambda_3\} \cap \{\lambda_3, \lambda_4\} \neq \emptyset$). The second application is not potentially recursive. The third is.

When processing applications in later phases, we will want to know whether they are potentially recursive or not. Therefore, we annotate each application with “ \circ ” or “ \emptyset ” according to whether it is potentially recursive or not. E.g., if $\text{fib}_\lambda(n-2)$ is a potentially recursive application, the annotated version is: $\text{fib}_\lambda^\circ(n-2)$. The application $(\lambda y.y \text{ at } r7)_{\{\lambda y.y \text{ at } r7\}} \text{fib}$, which is not potentially recursive, has this annotated version:

$$(\lambda y.y \text{ at } r7)_{\{\lambda y.y \text{ at } r7\}}^\emptyset \text{fib}.$$

This annotation of applications is done by the function $\llbracket \cdot \rrbracket_{\text{ar-}\Lambda}$. It translates a lambda-annotated function to a *recursiveness-annotated* function in which all applications are annotated with an $r \in R$ where $R ::= \circ \mid \emptyset$; i.e.,

after the translation, the expressions are

$$\overset{\circ}{E} ::= \overset{\circ}{E} \overset{R}{\Lambda} \overset{\circ}{E} \quad | \quad F \overset{\circ}{P} \overset{R}{\Lambda} \overset{\circ}{E} \quad | \quad \dots$$

The rest of the grammar is similar to that for \hat{E} .

Non-recursive applications

Consider the second dot in figure 25 above—the application where λ_5 may be applied. This application will destroy the registers that λ_5 destroys. But λ_5 may call λ_4 , and thus, calling λ_5 may also destroy the registers destroyed by λ_4 . In general, calling a function implies that all functions in its strongly connected component λ° may be called and thus that the set of registers that may be destroyed is the union of the sets of registers destroyed by the λ 's in λ° . Although λ_5 calls λ_4 , the set of registers destroyed by λ_4 is not necessarily a subset of the registers that are recorded to be destroyed by λ_5 , for λ_5 may have been processed before λ_4 , i.e., at a point where it was still undecided what registers would be destroyed by λ_4 . The set of registers destroyed by λ_6 , on the other hand, *will* be included in the set of registers recorded to be destroyed by λ_5 , because λ_6 is in another strongly connected component than λ_5 and therefore will have been processed when λ_5 is processed.

The essence of this is that the “registers destroyed by ...” concept should be associated with each strongly connected component rather than with each function; i.e., the η^d -component of the inter-procedural environment η maps strongly connected components λ° to the set of registers that may be destroyed when a $\lambda \in \lambda^\circ$ is called. For instance, after having processed the strongly connected components $\{\lambda_6\}$ and $\{\lambda_4, \lambda_5\}$ the environment is $\eta = (\eta^l, \eta^d)$, where η^l maps equivalence classes of functions to linking conventions as before, and η^d may be

$$\eta^d = \left\{ \begin{array}{l} \{\lambda_4, \lambda_5\} \mapsto \{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5\}, \\ \{\lambda_6\} \mapsto \{\phi_1, \phi_2, \phi_3\}, \quad \{\lambda_2, \lambda_3\} \mapsto \emptyset, \quad \{\lambda_1\} \mapsto \emptyset \end{array} \right\}.$$

assuming λ_4 and λ_5 together destroy ϕ_1 through ϕ_5 , etc., and assuming that we have yet to process the strongly connected components $\{\lambda_1\}$ and $\{\lambda_2, \lambda_3\}$. Notice that $\eta^d\{\lambda_6\} \subseteq \eta^d\{\lambda_4, \lambda_5\}$, as one would expect because λ_5 calls λ_6 .

Approximating the set of registers that will be destroyed by a function

There is one final aspect to processing a strongly connected component that we must consider before we give the final description of the algorithm. Remember the goals (i) and (ii) (p. 59), which should be pursued when choosing a register for a value. Some kind of preliminary analyses will be required to fulfil them.

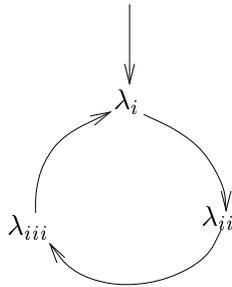
Goal (i) requires a liveness analysis on the λ currently being processed. This liveness analysis can be done on a per-function basis, as values are not

live inter-procedurally, for free variables of a function are fetched from its closure (p. 37). The liveness analysis will be discussed when we discuss how to process a λ (section 6.5).

Goal (ii) instructs us to put a value in a register that will be destroyed anyway. To decide which registers will be destroyed anyway, another analysis is needed.

At first, it may seem natural to do this analysis on a per-function basis too, i.e., to start the processing of each λ by finding out which registers will be destroyed by applications in that λ . E.g., start processing λ_3 of figure 20 (p. 58) by discovering that the registers ϕ_1 through ϕ_7 will be destroyed anyway because they are destroyed by applications in λ_3 .

But since calling a function may mean calling the other functions in its strongly connected component, it might be better to do this analysis on a strongly-connected-component basis. Consider the strongly connected component $\{\lambda_i, \lambda_{ii}, \lambda_{iii}\}$:



Assume a per-function approximation of which registers are destroyed anyway says λ_i , λ_{ii} , and λ_{iii} will destroy $\check{\phi}_i$, $\check{\phi}_{ii}$, and $\check{\phi}_{iii}$, respectively. Since an application of, e.g., λ_i may trigger applications of other λ 's in the strongly connected component, we might choose to consider the set of registers that are destroyed anyway by each λ as the set $\check{\phi}_i \cup \check{\phi}_{ii} \cup \check{\phi}_{iii}$.

For strongly connected components that represent loops in which some time will be spent at run-time, the latter approach is the most reasonable: Calling a function in the strongly connected component will make the program loop through all functions in the strongly connected component. Thus, the registers that will be destroyed anyway in each λ in the strongly connected component are the registers that will be destroyed by the loop; it is immaterial that the loop happens to be distributed over many functions.

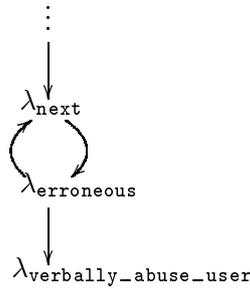
On the other hand, strongly connected components do not necessarily represent loops. Consider the situation where a λ only infrequently calls other functions in its strongly connected component. An example could be a function that calls an error handling function when it (very rarely) is applied to an invalid argument. Assume the error function rescues the situation by calling back its caller with a valid argument. Then the two functions will be in a strongly connected component together:

```

letrec
  next year                = if year>99 then erroneous year
                             else year+1
  erroneous year           = let x = verbally_abuse_user year
                             in next (year mod 100)
  verbally_abuse_user year = ⟨some expression whose code destroys
                             many registers⟩
  ⋮

```

This example gives rise to the following call graph fragment:



Here, it is inaccurate to approximate the set of registers that will be destroyed anyway by λ_{next} by the set of registers that is destroyed by all the λ 's in the strongly connected component $\{\lambda_{\text{next}}, \lambda_{\text{erroneous}}\}$, for it will destroy many registers because it contains $\lambda_{\text{erroneous}}$, which calls $\lambda_{\text{verbally_abuse_user}}$ that destroys many registers.

The two solutions, to approximate per function, and to do it per strongly connected component, are equally simple to implement, and it is possible to argue for both solutions. Perhaps the difference between them is small in practice. We choose the latter.

Roughly, the approximation of which registers are destroyed anyway by λ is the union of the sets of registers that are destroyed by the functions that may be applied by λ . These sets are known because the functions that may be applied by λ have already been processed (at least in the cases where there is no recursion). There are, however, other opportunities for predicting which specific registers will be destroyed by the code for a given expression. For instance, we know that the code for **letregion** $\varrho:?$ **in** e_1 creates a region in the heap with the instruction $\phi := \text{letregion}$ (section 4.4), and thus that it will destroy the set of registers destroyed by this specific instruction (viz. $\hat{\phi}_{\text{letregion}}$ —cf. chapter 3). We can define a function to approximate the set of registers that the code for a λ destroys.

Given a $\overset{\circ}{\lambda}$ and an inter-procedural environment $\eta \in \mathbf{H}$, the function

$$\llbracket \cdot \rrbracket_{\text{da-}\Lambda} \in \overset{\circ}{\Lambda} \rightarrow \mathbf{H} \rightarrow \mathcal{P}\Phi$$

returns an approximation of the set of registers that will be destroyed by the code for $\overset{\circ}{\lambda}$. The environment is needed for approximating which registers are destroyed by applications directly within $\overset{\circ}{\lambda}$.

The implementation of $\llbracket \cdot \rrbracket_{\text{da-}\Lambda}$ is explained in detail in section 7.11.

5.11 Revised overall algorithm

We have discussed some necessary modifications to the basic bottom-up overall algorithm. This section presents the revised overall algorithm.

$\text{rdfs } \gamma$ first finds the set $\mathbf{\lambda}^{\equiv\text{s}}$ of equivalence classes of functions that must use the same linking convention, then sets up the initial environment η_0 , and calls rdfs_0 to process the strongly connected components graph γ in reverse-depth-first search order. Assume γ has the form $(\mathbf{\lambda}^{\circ\text{s}}, \mathcal{S}, \mathbf{\lambda}_{\text{main}}^{\circ})$ and $\mathbf{\lambda}_{\text{main}}^{\circ}$ is $\{\lambda_{\text{main}}\}$:

$$\begin{aligned} \text{rdfs } \gamma &= \text{let } \mathbf{\lambda}^{\equiv\text{s}} = \llbracket \lambda_{\text{main}} \rrbracket_{\text{uf}} \\ &\quad \eta_0 = (\{ \mathbf{\lambda}^{\equiv} \mapsto -1_{\text{c}} \mid \mathbf{\lambda}^{\equiv} \in \mathbf{\lambda}^{\equiv\text{s}} \}, \\ &\quad \quad \{ \mathbf{\lambda}^{\circ} \mapsto \emptyset \mid \mathbf{\lambda}^{\circ} \in \mathbf{\lambda}^{\circ\text{s}} \}) \\ &\quad (\kappa, \eta) = \text{rdfs}_0 \gamma \mathbf{\lambda}_{\text{main}}^{\circ} \eta_0 \\ &\quad \text{in } \kappa. \end{aligned}$$

The reverse-depth-first traversal rdfs_0 is almost as before, except that it now works on the strongly connected components graph instead of on the call graph, and hence uses the function $\text{do-scc} \in \Lambda \rightarrow \mathbf{H} \rightarrow \mathbf{K} \times \mathbf{H}$ to process each node, instead of $\llbracket \cdot \rrbracket_{\text{donode}} \in \Lambda \rightarrow \mathbf{H} \rightarrow \mathbf{K} \times \mathbf{H}$:

$$\begin{aligned} \text{rdfs}_0 \gamma \mathbf{\lambda}_{\text{cur}}^{\circ} \eta &= \text{let } \{ \mathbf{\lambda}_1^{\circ}, \dots, \mathbf{\lambda}_l^{\circ} \} = \text{children } \gamma \mathbf{\lambda}_{\text{cur}}^{\circ} \\ &\quad (\kappa_1, \eta) = \text{rdfs}_0 \gamma \mathbf{\lambda}_1^{\circ} \eta \\ &\quad \quad \vdots \\ &\quad (\kappa_l, \eta) = \text{rdfs}_0 \gamma \mathbf{\lambda}_l^{\circ} \eta \\ &\quad (\kappa, \eta) = \text{do-scc } \mathbf{\lambda}_{\text{cur}}^{\circ} \eta \\ &\quad \text{in } (\kappa ; \kappa_1 ; \dots ; \kappa_l, \eta). \end{aligned}$$

Roughly, $\text{do-scc } \mathbf{\lambda}_{\text{cur}}^{\circ} \eta$ uses $\llbracket \cdot \rrbracket_{\text{donode}}$ on all λ 's in the current strongly connected component $\mathbf{\lambda}_{\text{cur}}^{\circ}$. See explanations below:

$$\begin{aligned} \text{do-scc } \mathbf{\lambda}_{\text{cur}}^{\circ} \eta &= \text{let } \{ \lambda_1^{\circ}, \dots, \lambda_j^{\circ} \} = \{ \llbracket \lambda \rrbracket_{\text{ar-}\Lambda} \mathbf{\lambda}_{\text{cur}}^{\circ} \mid \lambda \in \mathbf{\lambda}_{\text{cur}}^{\circ} \} \\ &\quad \check{\phi} = \llbracket \lambda_1^{\circ} \rrbracket_{\text{da-}\Lambda} \eta \cup \dots \cup \llbracket \lambda_j^{\circ} \rrbracket_{\text{da-}\Lambda} \eta \\ &\quad \nu = (\eta, \check{\phi}) \\ &\quad (\kappa_1, \nu) = \llbracket \lambda_1^{\circ} \rrbracket_{\text{donode}} \nu \\ &\quad \quad \vdots \\ &\quad (\kappa_j, \nu) = \llbracket \lambda_j^{\circ} \rrbracket_{\text{donode}} \nu \\ &\quad \eta = (\nu^1, \nu^d + \{ \mathbf{\lambda}_{\text{cur}}^{\circ} \mapsto \nu^{\check{\phi}} \}) \\ &\quad \text{in } (\kappa_1 ; \dots ; \kappa_j, \eta). \end{aligned}$$

The arguments of *do-scc* are the current strongly connected component $\lambda_{\text{cur}}^{\circ}$, and the current inter-procedural environment, η , and it returns the code for the functions in $\lambda_{\text{cur}}^{\circ}$, and an updated inter-procedural environment. Before the functions are processed, all applications in them are annotated by $\llbracket \cdot \rrbracket_{\text{ar-}\Lambda}$, yielding the recursiveness annotated functions $\{\lambda_1^{\circ}, \dots, \lambda_j^{\circ}\}$. Furthermore, the set, $\check{\phi}$, of registers that will be destroyed anyway by $\lambda_{\text{cur}}^{\circ}$, is approximated. The function $\llbracket \cdot \rrbracket_{\text{donode}}$ takes and returns a *strongly connected component environment* ν , which comprises the inter-procedural environment η and the approximation $\check{\phi}$: $\nu = (\eta, \check{\phi})$. We use ν^{η} to denote the η in ν , and $\nu^{\check{\phi}}$ to denote the $\check{\phi}$ in ν ; etc. There is no natural order in which to process the λ 's in a strongly connected component; *do-scc* simply processes them in arbitrary order.

While processing a function, $\llbracket \cdot \rrbracket_{\text{donode}}$ updates the $\check{\phi}$ in ν whenever a value is allocated to some register. Hence the $\check{\phi}$ in the ν returned by the last $\llbracket \cdot \rrbracket_{\text{donode}}$ tells which registers will be destroyed when a function in the strongly connected component $\lambda_{\text{cur}}^{\circ}$ is applied. The inter-procedural environment η returned by *do-scc* is updated to record this.

5.12 Relation to other approaches

This section relates our inter-procedural register allocation to others. See also (Steenkiste, 1991) for a discussion of inter-procedural register allocation schemes.

Per-function inter-procedural register allocation

We have mentioned two approaches to inter-procedural register allocation: truly inter-procedural and per-function inter-procedural (p. 57). All inter-procedural register allocators using the latter approach, including ours, have, to our knowledge, used Steenkiste and Hennessy's (1989) method of processing the call graph bottom-up.

Theirs is the only inter-procedural register allocation for a call-by-value functional language (Lisp) we have heard of.

The main differences between our and their inter-procedural strategies are: We use individual linking conventions for functions. This is more difficult in Lisp, especially because of the dynamic environment of Lisp. They only distinguish between applications where a single, known function may be applied and applications where an unknown function may be applied; we use a closure analysis to approximate the set of functions that can be applied.

Chow's (1988) approach to inter-procedural register allocation is also based on Steenkiste's idea. As the per-function part of his algorithm, Chow uses priority-based colouring (Chow and Hennessy, 1990). His source language is imperative. (The register allocator is implemented in a general back end used for a C and a Pascal compiler, among others.)

Chow deals in a uniform manner with the situations when the inter-procedural information is not available—at recursive calls, indirect calls and calls to functions in another module (separate compilation). In all these cases,

he uses a fixed linking convention with, among other things, a convention for which registers are caller- and callee-save registers. This implies that registers will be preserved properly across recursive calls. This way, he elegantly kills three birds with one stone. While one does not expect a compiler for an imperative language to implement recursion especially efficiently, it is crucial that it is in a functional language: we must do better than just use a fixed convention for recursive calls. Hence his solution is not applicable for us.

In our compiler, different functions may be called at the same application:

```
let i = if p then f else g in
    (f 7, i 9) at r13
```

At `i 9`, both `f` and `g` may be applied. This is not the case in Chow's compiler. Consider the C fragment

```
if (p) i=&f; else i=&g;
f(7); *i(9);
```

At an application, either one specific, named function is called (e.g. `f(7);`), or a function pointed to by some variable is called (e.g. `*i(9);`). The latter is called an *indirect call*, and Chow's compiler does not attempt to determine which functions might be called in that case. This is quite reasonable in an imperative language, where indirect calls are generally used less frequently than functions are used as values in higher-order functional languages. Furthermore, finding out which functions might actually be called at an indirect call requires an elaborate data-flow analysis (that would probably not give very accurate information anyway)—closure analysis is easier in functional languages.

Thus, in Chow's compiler, at an application, it is either completely unknown which function might be called, or it is known what single function will be applied. Therefore, unlike us, Chow does not have to worry about functions that may be applied at the same application and hence must use the same linking convention. Consequently, it is also less complicated for Chow than for us to allow different functions to have different linking conventions.

Like our algorithm, Chow's allows a function to pass parameters in more than one register.

Chow measures reductions in executed clock cycles of the generated code from -1% to 14% on 13 programs with a geometric mean of about 3% . We explain this slightly discouraging result as follows. First, his language is imperative and not functional. Second, he compares the inter-procedural register allocation with an intra-procedural one with four registers for passing parameters and conventions for which registers are caller-save and callee-save registers, and this gives many of the advantages that inter-procedural register allocation gives; i.e., his baseline is rather good. Third, he counts clock cycles instead of measuring actual execution time, so the interaction with the cache is not in the picture. Fourth, his benchmarks use library routines that do not participate in the inter-procedural register allocation.

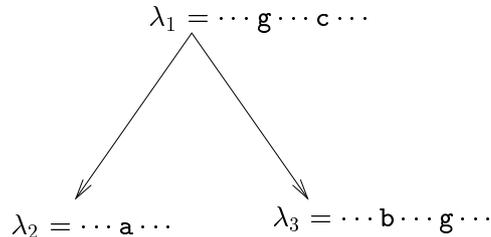
According to Chow, there is a clear tendency that the inter-procedural register allocation does better on smaller programs. He conjectures that this is because larger programs have deeper call graphs and hence the set of registers has been exhausted in the upper regions of the call graph.

Truly inter-procedural register allocation

The first inter-procedural register allocation that has employed the approach of processing the entire program (truly inter-procedural register allocation) is Wall's (1986).

We assume there is no separate compilation and thus that we have the whole program. Wall deals with separate compilation by postponing the register allocation to link time, when all modules are linked together to form one object code file.

Doing the register allocation program-wide excludes graph colouring as a realistic method; it will be too costly in time and space. Instead of building an interference graph, Wall groups together local variables that will never be live at the same time and thus can share the same register. Wall uses the call graph, in a way analogous to ours, to see which local variables will be live at the same time. If functions have local variables and call each other as indicated here,



a and **b** can be put in the same group, because they will never be live at the same time, while **a** and **c** cannot because they may be live simultaneously. For each global variable **g**, a singleton group $\{\mathbf{g}\}$ is created. Now, if there are 52 registers, Wall's algorithm picks the 52 groups that have highest priority, and assigns registers to the variables in them. The priority of a group is the sum of some estimate of the usage frequencies of the variables in the group. In comparison, Steenkiste's way of processing the call graph bottom-up will simply give highest priority to the functions in the lower parts of the call graph.

Wall allocates free variables to registers unlike us. If we wanted to allocate free variables to registers, we would have to change the decision to fetch all free variables from a closure (section 4.6). Allocating a free variable to a register is easier for Wall, because in his source language (C) a variable is either local to one specific function or free in all functions it occurs in, i.e., *global*. In SML, a variable can be local in one function and free in another.

At function calls, Wall allocates space on the stack for the parameters. The function entry code will load the parameters from the stack to their registers. Thus, functions have a uniform linking convention, which is necessary because they might be called indirectly. At direct calls, though, the uniform linking convention is not used; instead the parameters are moved directly to their registers and the function entry code that loads them from the stack is skipped.

Summing up, Wall's inter-procedural register allocation is fundamentally different from ours in that it is truly inter-procedural, but it uses inter-procedural information akin to our "destroys" information. Functions can use different linking conventions stating which parameters are passed on the stack, and which in registers, but not in which specific register each parameter is passed. Thus, inter-procedural information about *which* registers to pass parameters in, is not used in the register allocation of the function. Wall deals with indirect calls by, in principle, having two versions of each function: one conforming to a specific, uniform linking convention, and a specialised version that takes some of its parameters in registers.

The inter-procedural register allocation of Santhanam and Odnert (1990) must also be categorised as truly inter-procedural, because it can allocate a global variable (their source language is C) to a register over many functions. The main improvement over Wall's allocation of global variables to registers, is that the global variable does not have to be allocated to a register in all of the program, and the same global variable can even be allocated to different registers in different functions. Santhanam and Odnert do this by identifying *webs* in the call graph, where a particular global variable is used. A web is a global variable and a minimal sub-graph of the call graph such that the global variable is neither referenced in any ancestor node nor in any descendent node of the sub-graph. They build an *interference graph of webs*, where the nodes are the webs and two webs interfere, i.e., there is an edge between them, if they contain the same call graph node. Then graph colouring is used to assign registers to webs. If a web does not get a colour, the corresponding global variable is not allocated to a register across function boundaries in that web, but the global variable may be allocated to registers in other webs.

Inter-procedural register allocation for call-by-need languages

(Boquist, 1995) describes an inter-procedural register allocation for a higher-order call-by-need language. Note that the implementation technology for call-by-need languages is very different from that for call-by-value languages. Boquist's approach is truly inter-procedural: he uses graph colouring for the whole program. This may prove untractable for large programs. Like we do, Boquist allows individual linking conventions for functions. The node coalescing part of the graph-colouring register allocation algorithm (i.e., the elimination of register-to-register moves) decides the individual linking conventions. Boquist uses analyses similar to closure analysis to narrow in the possible control flow.

6 Per-function register allocation

In the previous chapter we explained how to translate a program e to intermediate code κ by translating the functions of the program one at a time. This chapter discusses how to translate each function. Since we have already discussed what intermediate code to generate for each construct of E (chapter 4), this chapter is mostly concerned with the register allocation aspects of the translation.

Our general approach is to allocate registers on the source language, *before* the intermediate-code generation, and then let the register allocation direct the intermediate-code generation. The customary approach is the opposite: first generate intermediate code, then allocate registers. We motivate our uncommon approach (section 6.1); then the rest of this chapter discusses how to implement it. This is done by developing the register allocation and subsequent intermediate-code generation (from now on: the *translation*) for the construct `let $x = e_1$ in e_2` .

Developing the translation implies discussing how the register allocation can direct the intermediate-code generation (section 6.2), what kinds of values there are (section 6.3), and how to allocate values to registers (section 6.4). To aid when choosing a register for a value, we use a standard liveness analysis tailored to our needs (section 6.5). The allocation of values to registers is simple-minded: simply allocate a value to a register when the value is used during the translation (section 6.6). A heuristic function is used to choose a register given the liveness information at a given program point (section 6.7). Before we can finish developing the translation of `let $x = e_1$ in e_2` , we must discuss what to do when not all values fit in registers: We discuss which values are *spilled*, i.e., kept in memory (section 6.8). We give a rudimentary framework for discussing where to place *spill code*, the code that moves values between memory and registers (section 6.9). Using this framework, we discuss where to place spill code within functions (section 6.10) and callee-save vs. caller-save registers (section 6.11). In the end, we content ourselves with a simple spill code placement strategy, and present the final translation for the `let`-construct (section 6.12). We conclude by comparing our per-function register allocation with other methods (section 6.13).

6.1 General approach

Graph colouring

Register allocation by graph colouring is done by building an *interference graph*; the nodes are live ranges of values, and there is an edge between two live ranges if they overlap, i.e., if they must not be allocated to the same register. Colouring this graph with at most k colours such that no two neighbouring nodes get the same colour corresponds to assigning the live ranges to at most k registers such that no two overlapping live ranges are put in the same register.

Since Chaitin et al. implemented the first register allocator to use graph colouring (Chaitin, 1982, Chaitin et al., 1981), almost all register allocators in the literature have been cast in this framework.

A strength of graph-colouring register allocation is that it creates a *global picture* (the interference graph) of the problem and converts the complicated optimisation problem of having as much data in registers as possible into a conceptually simpler one—that of colouring a graph. The problem of colouring a graph with as few colours as possible is simpler to understand, but not to solve, as it is NP-hard (Garey and Johnson, 1979). An interference graph gives a global picture of the *problem*, not a solution. A heuristic must be used to solve the problem.

A problem with basic graph-colouring register allocation is that a value is either allocated to a register for all of its live range or not at all; it cannot be put in memory in some parts of its live range and in a register in other parts, or be put in different registers in different parts of its live range. (Extending the basic graph-colouring register allocation to do live range splitting (Chow and Hennessy, 1990), may circumvent some of these problems.)

Graph colouring captures the problem of deciding which values should be kept in which registers. It does not easily address the problem of where to place spill code, because there is no connection between the program and the interference graph.

Furthermore, the interference graphs may become very large (Gupta et al., 1994).

One way of addressing these shortcomings of basic graph-colouring register allocation is to take the structure of the program into account, instead of only looking at the interference graph. Callahan and Koblenz (1991) do this: they do graph-colouring register allocation on the parts of the program instead of on the whole program. This means that a value can be in different registers, etc. in different parts of the program. It also reduces the problem with the size of interference graphs. When spill code is placed, the structure of the program is used. For instance, the area where a value is not in a register may be increased in order to move its spill code outside a loop. The results of the graph-colouring register allocation of the parts of the program are combined in a way that retains the global perspective. This way they get the global picture an interference graph for the whole program would give, while avoiding the unfelicities of basic graph-colouring register allocation.

We explore the idea of using the structure of the source language for register allocation in the next section.

Using the structure of the source program

Viewing register allocation as one of many transformations in the compiler, the question arises: What requirements should the input language to the register allocator satisfy?

1. It must be so close to the target machine language that it can be seen which registers are needed for the different operations. Our intermediate lan-

guage K has this property (almost): It is easy to see that, e.g., $\phi_1 := \phi_2 + \phi_3$ maps to a target machine instruction that will use three registers. (It is only almost, because for instance on the PA-RISC, the instruction $m[\phi_1 + \iota] := \phi_2$ will map to instructions that use an auxiliary register if the offset ι is sufficiently large.)

2. Uses of values should appear as such, because it is not known until after the register allocation what they should end up as in the target language. Whether a value must be loaded or not depends on whether it is allocated to a register or not. The form of the code to load the value depends on whether the value is, e.g., free, **let**-bound, or an argument to a function. Consequently, it should be possible to see whether a value is, e.g., free, **let**-bound, or an argument to a function. Our intermediate language does not accommodate this, but it could be extended with a “use value instruction”.

3. Function calls should be recognisable or else inter-procedural register allocation will not be possible. If we extend K with a “function call instruction”, it could satisfy this.

The usual manner of designing an input language to the register allocator would now be to take a language very close to the target machine language and extend it to satisfy the three requirements. But observe that the source language E obviously satisfies the last two requirements and actually also the first, for we know which registers are needed by the different constructs of the language, because we know what code should be generated for each construct. For instance, the code for $e_1 + e_2$ has the form

$$\phi_1 := \boxed{\text{code to evaluate } e_1} ; \phi_2 := \boxed{\text{code to evaluate } e_2} ; \phi := \phi_1 + \phi_2 ,$$

where ϕ_1 and ϕ_2 are registers needed temporarily, and ϕ is the register that should hold the result of the expression. Hence, we know that two registers are needed for the construct $e_1 + e_2$.

By taking the structure of the program into account, Callahan and Koblenz address some of the problems that basic graph-colouring register allocation does not consider. We want to experiment by going further and avoid graph colouring entirely by using source level information. At the same time, we want to explore how much can be retained of the global picture an interference graph would give. Some of this global perspective can be found in the structure of the source language: Above, the value that e_1 evaluates to is live while e_2 is evaluated, because it is defined by the code for e_1 and used after e_2 has been evaluated when the results of the two sub-expressions are added together. If we keep the value from e_1 in ϕ_1 , the code for e_2 must not change ϕ_1 , or alternatively, ϕ_1 must be preserved elsewhere across the code for e_2 .

With this approach, the result of the register allocation is (as normally) to decide which registers should be used and when values should be moved between memory and registers. A usual register allocator would modify the already generated code to incorporate this information, but our register allocator instead uses the information to direct the generation of code. E.g., for the expression above, the register allocation would result in a choice of ϕ_1

and ϕ_2 and maybe a decision to preserve ϕ_1 across the code for e_2 . The code for the expression would then be generated accordingly, *after* the register allocation.

It is well-known that it is possible to translate the source language directly to register allocated code, for this is done by syntax-directed compilers that keep track of the contents of the registers in a register descriptor (Waite, 1974). What we want to investigate is *using* source level information to make *good* register allocation.

Now we give a first try at developing the translation of `let $x = e_1$ in e_2` without worrying about exactly how the register allocation is done. We will then discuss how source-level register allocation of that expression may be done.

6.2 Translating an expression

The main difference between an expression-oriented language and three-address code is that in the latter, (sub)expressions must be designated a specific destination register (except when the expression is a constant). For instance, the expression

$$\text{let } x = (a+b)+c \text{ in } e_2$$

could be translated to these instructions:

$$\begin{aligned} \phi_1 &:= \phi_a + \phi_b ; \\ \phi_x &:= \phi_1 + \phi_c ; \\ \phi &:= \boxed{\text{code to evaluate } e_2}, \end{aligned}$$

where ϕ is the register for the result, ϕ_1 is a temporarily used register, and ϕ_a, ϕ_b, ϕ_c , and ϕ_x are the registers allocated to a, b, c , and x , respectively. (At first, register allocation will not be the issue; we assume all values are in registers.) We only used a temporary register for the sub-expression $a + b$. If we had introduced a new temporary register for every sub-expression, the instructions would have been:

$$\begin{aligned} \phi_1 &:= \phi_a ; \\ \phi_2 &:= \phi_b ; \\ \phi_3 &:= \phi_1 + \phi_2 ; \\ \phi_4 &:= \phi_c ; \\ \phi_5 &:= \phi_3 + \phi_4 ; \\ \phi_x &:= \phi_5 ; \\ \phi &:= \boxed{\text{code to evaluate } e_2}. \end{aligned}$$

The problem we consider in this section is how to decide when it is necessary to introduce a new temporary register.

If an expression “naturally provides a specific destination register”, it is not necessary to introduce a new temporary register. This applies to the sub-expressions **a**, **b**, and **c** above, whose natural destination registers are the registers they are allocated to— ϕ_a , ϕ_b , and ϕ_c , respectively. Therefore, ϕ_1 , ϕ_2 , and ϕ_4 are not needed. Conversely, ϕ_3 *is* necessary, because the expression **a + b** does not naturally provide a specific destination register.

Also the *context* of a sub-expression may naturally provide a specific destination register, and then a new temporary register for that sub-expression need not be introduced. For instance, the context **let x = □ in e₂** naturally provides a specific destination register, viz. the register allocated to **x**, ϕ_x , and therefore the temporary, ϕ_5 , for the sub-expression **(a + b) + c**, is not needed.

Summing up, a temporary is needed for a given sub-expression, iff neither the sub-expression, nor its context, naturally provide a specific destination register. Hence, we want information to flow both upwards (from sub-expression to context) and downwards (from context to sub-expression), when translating expressions.

Consider how to translate the expression **let x = e₁ in e₂**. Using ϕ_x for the register allocated to *x*, the code should be

$$\phi_x := \boxed{\text{code to evaluate } e_1} ; \phi := \boxed{\text{code to evaluate } e_2}.$$

When translating the **let**-expression, we do not know what the destination register, ϕ , for the whole **let**-expression is. To get information to flow downwards, we do not translate an expression to instructions, κ , but rather to a function, β , that will return instructions to evaluate the expression, when it is applied to the result register:

$$\beta = \lambda\phi. \phi_x := \boxed{\text{code to evaluate } e_1} ; \phi := \boxed{\text{code to evaluate } e_2}.$$

(The body of a λ -abstraction extends as far to the right as possible.) One can think of β as some code with a hole in it for the destination register. If the sub-expressions e_1 and e_2 are translated to β_1 and β_2 , respectively, the translation of **let x = e₁ in e₂** can be written as

$$\beta = \lambda\phi. \beta_1\phi_x ; \beta_2\phi.$$

The context of the **let**-expression will be able to decide which register the result of the expression should be placed in, by applying β to that register.

Now the function $\llbracket \cdot \rrbracket_{\text{ra}}$ that translates an expression can be defined for the **let**-construct:

$$\begin{aligned} \llbracket \cdot \rrbracket_{\text{ra}} &\in E \rightarrow B \\ \beta &\in B = \Phi \rightarrow K \\ \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} &= \text{let } \beta_1 = \llbracket e_1 \rrbracket_{\text{ra}} \\ &\quad \beta_2 = \llbracket e_2 \rrbracket_{\text{ra}} \\ &\quad \beta = \lambda\phi. \beta_1\phi_x ; \beta_2\phi \\ &\quad \text{in } \beta, \end{aligned}$$

where ϕ_x is the register that should contain x . Notice how information flows from the context to the sub-expression, when a β is applied to a ϕ .

We also wanted information to flow the other way—from sub-expression to context. This is achieved by modifying $\llbracket \cdot \rrbracket_{\text{ra}}$ to return not only a β , but also a *natural destination register*. Thus, now the result of $\llbracket e \rrbracket_{\text{ra}}$ is a pair $(\overset{\dagger}{\phi}, \beta)$, where $\overset{\dagger}{\phi}$ is the natural destination register for e . If e does not naturally have a specific destination register, we say its natural destination register, $\overset{\dagger}{\phi}$, is $-\text{register}$. Hence, $\llbracket \cdot \rrbracket_{\text{ra}} \in E \rightarrow (\Phi_{\perp} \times \text{B})$, where $\Phi_{\perp} = \Phi \cup \{-\text{register}\}$. $\overset{\dagger}{\phi}$ always ranges over Φ_{\perp} .

For **let** $x = e_1$ **in** e_2 , the natural destination register is the same as the natural destination register of the sub-expression e_2 :

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} &= \text{let } (\overset{\dagger}{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \\ &\quad (\overset{\dagger}{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \\ &\quad \beta = \lambda\phi. \beta_1\phi_x ; \beta_2\phi \\ &\quad \text{in } (\overset{\dagger}{\phi}_2, \beta), \end{aligned}$$

where ϕ_x is the register that should contain x .

If we extend $\llbracket \cdot \rrbracket_{\text{ra}}$ to also decide what ϕ_x should be, it would be a combined register allocation and code generation for the **let**-expression:

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} &= \text{let } (\overset{\dagger}{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \\ &\quad (\overset{\dagger}{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \\ &\quad \boxed{\begin{array}{l} \text{the register allocation part of the translation:} \\ \text{find a register, } \phi_x, \text{ to contain } x \end{array}} \\ &\quad \beta = \lambda\phi. \beta_1\phi_x ; \beta_2\phi \\ &\quad \text{in } (\overset{\dagger}{\phi}_2, \beta). \end{aligned}$$

Summarising, we intend to make the per-function part of the register allocator work on the source language representation of each function by incorporating the register allocation in the translation function $\llbracket \cdot \rrbracket_{\text{ra}}$. In the rest of this chapter, we develop $\llbracket \cdot \rrbracket_{\text{ra}}$ for the **let**-construct. The translation is developed for the other constructs in chapter 7.

6.3 What kinds of values can be allocated to registers

A *value* is data that may reside in a register at run-time. In this section, we analyse what kinds of values there are. We refer to the function currently being processed with λ_{cur} . The following contrived fragment serves to illustrate the different kinds of values.

```

letregion r5:? in
  let x1 = 5 in
    letrec f1 [r1:?] y1 =
      letregion r3:3 in
        letregion r4:? in
          (y1+x1) +
            let x2 = (c1 x1 at r5:?,
                    λy3.let x3 = 3
                    in f2 [r3:3] y3 at r1:?) at r3:3
          in 7
        f2 [r2:2] y2 = (y2,y2) at r2:p2
      at r5:?
    in f2 [r5:?] 0

```

In some respects, all values are treated uniformly. E.g., any value is eligible for allocation to a register but also for keeping in memory, albeit in different ways. The kinds of values with respect to λ_{cur} are:

1. The argument y of λ_{cur} . (Suppose in the following examples that λ_{cur} is λ_{f1} , i.e., the function named **f1** in the fragment above. The argument of λ_{f1} is $y1$.)
2. Region arguments ρ of λ_{cur} . (**r1:?**).
3. **let**-bound variables x that are bound by a **let**-expression directly within λ_{cur} . (i.e. bound in λ_{cur} , but not bound in any λ within λ_{cur} .) (**x2**, but not **x3**, because it is also bound in λ_{y3} inside λ_{cur} .)
4. known-size **letregion**-bound variables $\rho:i$ bound directly within λ_{cur} . (**r3:3**).
5. unknown-size **letregion**-bound variables $\rho:?$ bound directly within λ_{cur} . (**r4:?**).
6. **letrec**-function names f bound directly within λ_{cur} . (There are none in the body of λ_{f1} , but if we assume for a minute that we are processing a λ whose body is the whole fragment, the **f2** in the region polymorphic application **f2 [r5:?] 0** in the last line is an example of a use of a **letrec**-function name.)
7. Exception constructors a bound directly within λ_{cur} . As was discussed in section 4.8, an exception constructor is a variable because it may be bound to different exception names at run-time (unlike a constructor, which is a constant).
8. Free variables ζ of λ_{cur} . (**x1**, **f2**, and **r5:?** are free variables of λ_f). In contrast to the above-mentioned kinds of values, this includes free variables that do not occur *directly* within λ_{cur} . (e.g., **f2**). The free variables will appear as values of the above-mentioned kinds (including

free variables) when some other λ is λ_{cur} , and they will not be of the kinds mentioned below. E.g., $\mathbf{x1}$ is a **let**-bound and $\mathbf{r5:?}$ is a **letregion**-bound, in the λ containing the whole fragment.

9. Temporary values. In the fragment above, the result of the sub-expression $(\mathbf{y1+x1})$ is called a temporary value. It is needed to evaluate the expression $(\mathbf{y1+x1}) + \mathbf{let\ x2 = \dots}$ but only after the **let**-expression has been evaluated; yet $(\mathbf{y1+x1})$ must be evaluated before the **let**-expression is evaluated. Thus, the value it evaluates to must be remembered somewhere while the **let**-expression is evaluated.
10. The closure. The free variables of λ_{cur} must be fetched from the closure, which is one of the parameters that are passed to λ_{cur} when it is applied.
11. The return label is also a value that is passed to λ_{cur} when it is applied.

These values except 9, 10, and 11 are named; i.e., they are variables in the program. If we worked on some intermediate language (e.g., continuation-passing-closure-passing style (Appel, 1992), instead of directly on the source language, the latter kinds of values could also be explicit in the language.

We treat the closure like the other values, and it partakes in the register allocation on equal terms with any other value: it can be kept in memory in some parts of the function, and it can be put in different registers—even in the same function. A more usual practice would perhaps be the—on the face of it—simpler solution of dedicating a specific register to hold the closure throughout the body of the function. If the same dedicated register is used in all functions, this would make it a busy register (but, of course, different registers could be the dedicated closure register in different functions). Worse, having dedicated registers complicates the algorithm in different ways, for many of the things that must be taken care of for values in normal registers must also be explicitly taken care of for the dedicated register: it must be saved across code that destroys it; it must be recorded somewhere what the dedicated register is so that it can be accessed. (These things were very bothersome in earlier versions of this algorithm, where the closure was not treated as a value.) By regarding the closure as a value, all of this is taken care of uniformly.

The observations about the closure generally also apply to the return label. Often in compilers, the return label is pushed on the stack, which is a reasonable thing to do, for it is only used once and this is at the very end of the function; i.e., the return label is very “spill-worthy”. We could have chosen that too, but it seems nicer to also treat the return label like other values. Also, when λ_{cur} is small enough to allow the return label to stay in a register until it is needed, this will be more efficient than having the return label on the stack.

Recall that Z is the set of variables (p. 19). We define the set of *values*:

$$V ::= Z \mid \mathbf{clos} \mid \mathbf{ret}.$$

6.4 Register allocation strategy

Our strategy for deciding which values are allocated to registers and which registers they are allocated to is as follows.

Code is generated and values are allocated to registers in a forward traversal of each sub-expression of the program.

Whenever a value is used it is allocated to a register; we do not have a concept of values allocated to memory, i.e., values that reside in memory and are loaded every time they are needed. This is only a conceptual difference, for in our scheme, a value might also have to be loaded each time it is needed if it is used so seldom that it is always thrown out of its register before it is used the next time. Not having values allocated to memory allows accesses of values to be treated in a more uniform way. (If the target machine were a Complex Instruction Set Computer (CISC) instead of a RISC, there would be a real difference, for a CISC can access a value in memory directly without first loading it to a register, and then it might be worth having a concept of values allocated to memory.)

We keep track of which registers contain which values in a *descriptor* $\delta \in \Delta$. Consider figure 20 again:

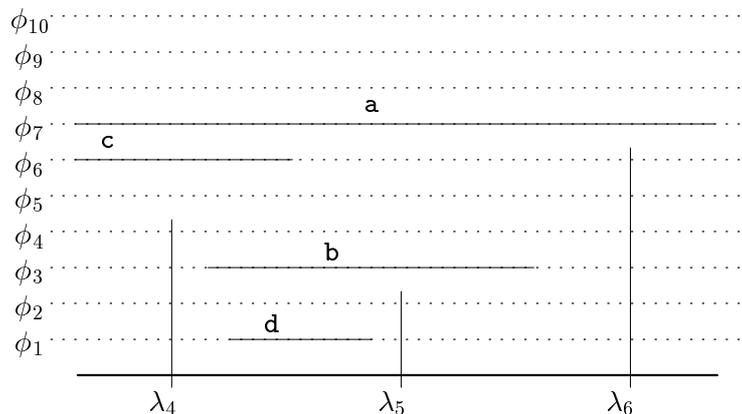


Fig. 20. from p. 58 once again

When a register is needed to hold a value v , we choose a register with these objectives in mind:

1. Preferably choose a register that v is naturally produced in. E.g., when choosing a register for x of `let $x = e_1$ in e_2` , preferably choose the natural destination register of e_1 (cf. section 6.2).
2. Avoid choosing registers that are known to be destroyed while v is live. E.g., when choosing a register for b in the figure, we want to avoid ϕ_1 and ϕ_2 , because they will be destroyed while b is live, viz. when λ_5 is called.

We say that v is *hostile to ϕ at a given program point* iff ϕ is known to be destroyed after that program point while v is live. E.g., \mathbf{b} is hostile to ϕ_1 and ϕ_2 at all program points before the call to λ_5 .

To know which registers v is hostile to at a given program point, we need a *hostility analysis*.

3. Avoid choosing registers that contain live values. E.g., when choosing a register for \mathbf{b} , just after the call to λ_4 , we want to avoid picking ϕ_7 and ϕ_6 , because they contain the live values \mathbf{a} and \mathbf{c} . The descriptor δ tells us that ϕ_7 and ϕ_6 contain \mathbf{a} and \mathbf{c} , respectively. To know that \mathbf{a} and \mathbf{c} are live, we need a *liveness analysis*. This analysis must decide, for each program point, which values will be needed further on.
4. Preferably choose the register with the smallest number that is known to be destroyed by the current strongly connected component (recall section 5.10). E.g., in figure 20, prefer ϕ_4 to ϕ_8 , because ϕ_4 will be destroyed anyway by λ_3 .

Objectives 2 and 4 are the two general goals (i) and (ii) from the preceding chapter (p. 59). Objectives 1 and 3 are more specific to the particular way we have chosen to do the per-function register allocation.

Before we discuss in greater detail how to choose registers, let us discuss the liveness and hostility analyses needed for objectives 2 and 3.

6.5 Liveness and hostility analyses

The purpose of the liveness analysis is to decide for each program point which values are live. The purpose of the hostility analysis is to decide for each program point the set of registers to which the values that are live at that program point are hostile. Both analyses are per-function analyses.

The two analyses can be done at the same time. Define ω -*information* to be a map,

$$\omega \in \Omega = V \xrightarrow{\perp} \mathcal{P}\Phi,$$

from the *live* values at a given program point to sets of registers that those values are *hostile* to. Hence, a value v is live at a program point with ω -information ω iff $v \in \text{Dm}\omega$, and in that case, ωv is the set of registers that v is hostile to.

The ω -*analysis* translates an expression to an ω -*annotated expression* $\check{e} \in \check{E}$, where

$$\begin{aligned} \check{E} ::= & \Omega X \Omega \\ & | \Omega \text{let } X = \check{E} \text{ in } \check{E} \Omega \\ & | \Omega \text{letregion } \acute{P} \text{ in } \check{E} \Omega \\ & \vdots \end{aligned}$$

and so on in the same fashion. Each construct has two ω 's annotated, which we will call the ω -information *before* and *after* the construct, respectively.

As an example, the expression

$$\text{let } \mathbf{x} = \overset{\circ}{e}_{\mathbf{x}} \text{ in } \mathbf{k}_{\lambda}^{\phi} 5 + \mathbf{x}$$

will be translated to an ω -annotated expression with the form

$$\omega_1 \left(\text{let } \mathbf{x} = \omega_2 \overset{\circ}{e}_{\mathbf{x}\omega_2}' \text{ in } \left(\omega_3 \left(\omega_4 \left(\mathbf{k}_{\lambda}^{\phi} \omega_5 5 \omega_5' \right) \omega_4' + \omega_6 \mathbf{x} \omega_6' \right) \omega_3' \right) \omega_1' \right)$$

Since \mathbf{x} is not live before or after the whole expression, $\mathbf{x} \notin \text{Dm } \omega_1$ and $\mathbf{x} \notin \text{Dm } \omega_1'$. Since \mathbf{x} is needed after the application of \mathbf{k} , it will be in, e.g., $\text{Dm } \omega_4$ and $\text{Dm } \omega_6$. Furthermore, if the application of \mathbf{k} destroys ϕ_1 through ϕ_7 , these ω 's will record that \mathbf{x} is hostile to these registers, i.e., $\{\phi_1, \dots, \phi_7\} \subseteq \omega_4 \mathbf{x}$, etc. One would expect that $\omega_6' = \omega_3' = \omega_1'$.

We shall always give ω -information after an expression a $'$. The implementation of the ω -analysis is explained in section 8.1.

6.6 Choosing a register for a value

Given an ω -annotated expression, we can now explain more specifically how a register is chosen with the four objectives in section 6.4 in mind. We do this by completing the definition of $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$; i.e., we explain how to “find a register, ϕ_x , to contain x ” (p. 82).

In the following, it is assumed that ω -information is annotated on all expressions, but to avoid cluttering the picture, the annotations will be omitted and only mentioned when needed.

Given the ω -information, ω , at a specific program point and a descriptor, δ , telling which registers contain which values at that program point, we can choose a register for a value while pursuing the objectives 1–4 above (p. 85). We use a heuristic function to pick a register: *choose* $\hat{\phi} \hat{\phi} \hat{\phi} \omega \delta$ yields a good choice of register, given (i) a set, $\hat{\phi}$, of registers that *must not* be chosen, (ii) a register, $\hat{\phi}$, we would *prefer* chosen, (iii) a set, $\hat{\phi}$, of registers we would *prefer not* chosen, (iv) ω -information ω , and (v) the current descriptor δ . Hence,

$$\text{choose} \in \mathcal{P}\Phi \rightarrow \Phi_{\perp} \rightarrow \mathcal{P}\Phi \rightarrow \Omega \rightarrow \Delta \rightarrow \Phi.$$

The usefulness of these parameters will become more apparent shortly. Typically, $\hat{\phi}$ is the natural destination register of an expression, while $\hat{\phi}$ will be used to tell *choose* which registers a given value is hostile to, and the set $\hat{\phi}$ is used when we want to prevent specific registers from being touched.

The heuristic *choose* uses $\hat{\phi}$ to live up to objective 1, $\hat{\phi}$ to live up to objective 2, δ and ω to live up to objective 3, and δ to live up to objective 4. The implementation of *choose* is described in the next section.

Using *choose* we can now define the register allocation for $\text{let } x = e_1 \text{ in } e_2$. While translating an expression, we must keep track of which registers contain which values in δ , i.e., the translation function $\llbracket e \rrbracket_{\text{ra}}$ is modified to

take an in-flowing δ —the descriptor describing the contents of the registers when *entering* the code to evaluate e —, and return an out-flowing δ —the descriptor at the *exit* of the same code:

$$(\delta_{\text{after}}, \hat{\phi}, \beta) = \llbracket e \rrbracket_{\text{ra}} \delta_{\text{before}}.$$

Thus, now $\llbracket \cdot \rrbracket_{\text{ra}} \in \overset{*}{E} \rightarrow \Delta \rightarrow \Delta \times \Phi_{\perp} \times \mathbf{B}$, and $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$ of p. 82 becomes

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} \delta_{\text{before } e} &= \\ \text{let } (\delta_{\text{after } e_1}, \hat{\phi}_1, \beta_1) &= \llbracket e_1 \rrbracket_{\text{ra}} \delta_{\text{before } e} \end{aligned}$$

the register allocation part of the translation:
 $\phi_x = \text{choose a register to contain } x$
 $\delta_{\text{before } e_2} = \delta_{\text{after } e_1}$ changed to record that ϕ_x contains x .

$$\begin{aligned} (\delta_{\text{after } e_2}, \hat{\phi}_2, \beta_2) &= \llbracket e_2 \rrbracket_{\text{ra}} \delta_{\text{before } e_2} \\ \beta &= \lambda\phi. \beta_1 \phi_x ; \beta_2 \phi \\ \text{in } (\delta_{\text{after } e_2}, \hat{\phi}_2, \beta). \end{aligned}$$

Notice how the flow of δ 's simulates (at compile-time) the run-time control flow.

How should we choose the register, ϕ_x , for x ? Preferably, we want to choose the natural destination register, $\hat{\phi}_1$, of e_1 , (and hence avoid a register-to-register move). We also want to avoid registers that are known to be destroyed while x is live, i.e. registers that x is hostile to according to the ω -information, ω_2 , before e_2 , i.e. the set $\omega_2 x$. Therefore, we pick ϕ_x with *choose* $\emptyset \hat{\phi}_1 (\omega_2 x) \omega_2 \delta$. The first argument is \emptyset , because there are no registers that must not be chosen.

Using $ra(\phi \mapsto v)\delta$ to denote δ updated to record that the value v has been allocated to the register ϕ , we can define the auxiliary function $\llbracket \cdot \rrbracket_{\text{def}}$ to do as prescribed in the box above:

$$\begin{aligned} \llbracket v \rrbracket_{\text{def}} \hat{\phi} \hat{\phi} \omega \delta &= \text{let } \phi_v = \text{choose } \hat{\phi} \hat{\phi} (\omega v) \omega \delta \\ &\quad \delta = ra(\phi_v \mapsto v) \delta \\ &\quad \text{in } (\delta, \phi_v). \end{aligned}$$

I.e., $\llbracket v \rrbracket_{\text{def}} \hat{\phi} \hat{\phi} \omega \delta$ chooses a register for v and updates δ accordingly (the arguments $\hat{\phi}$ and $\hat{\phi}$ are simply passed on to *choose*). Then,

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} \delta &= \text{let } (\delta, \hat{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \delta \\ &\quad (\delta, \phi_x) = \llbracket x \rrbracket_{\text{def}} \emptyset \hat{\phi}_1 \omega_2 \delta \\ &\quad (\delta, \hat{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \delta \\ &\quad \beta = \lambda\phi. \beta_1 \phi_x ; \beta_2 \phi \\ &\quad \text{in } (\delta, \hat{\phi}_2, \beta), \end{aligned}$$

where ω_2 is the ω -information before e_2 (i.e. annotated on the left side of e_2). (We have quit using qualifications on the δ 's, because a δ always refers to the last δ defined.) When reading this code, one can profitably ignore the δ 's that are passed around and regard the operations on the δ 's as side effects. Intuitively, $\llbracket x \rrbracket_{\text{def}} \dot{\phi} \delta \omega$ reads “find a register for x , given that a good candidate is $\dot{\phi}$, the descriptor is δ , and the information about the code following is ω .” The $\llbracket \cdot \rrbracket_{\text{def}}$ appears between $\llbracket e_1 \rrbracket_{\text{ra}}$ and $\llbracket e_2 \rrbracket_{\text{ra}}$ because x “becomes live” between the code for e_1 and that for e_2 .

Alternatively to viewing $\llbracket \cdot \rrbracket_{\text{ra}}$ as a function that translates an expression to some code (β) with a hole in it for the destination register, one can view $\llbracket \cdot \rrbracket_{\text{ra}}$ as a two-phase translation consisting of 1. register allocation, followed by 2. intermediate code generation. The first phase in the case for **let** $x = e_1$ **in** e_2 is when $\llbracket \cdot \rrbracket_{\text{ra}}$ traverses e_1 ; the register, ϕ_x , for x is chosen; and $\llbracket \cdot \rrbracket_{\text{ra}}$ traverses e_2 . The second phase is when all the β 's are applied. This is only after the whole of the current function has been traversed by $\llbracket \cdot \rrbracket_{\text{ra}}$. It is the “ $\lambda\phi$.” in front of the β 's that delays the actual generation of the code till the β 's are applied and thus splits the translation process into two stages. We could have formulated $\llbracket \cdot \rrbracket_{\text{ra}}$ explicitly as two phases: The result of the first phase, in the case of **let** $x = e_1$ **in** e_2 , would be to annotate the **let**-expression with ϕ_x ; and then the second phase would use this annotation to generate the code for the **let**-expression. The way we do it, ϕ_x is an “implicit annotation”: ϕ_x is a free variable in the β for the **let**-expression it “annotates”.

6.7 Heuristic for choosing a register

In this section, we describe how *choose* chooses a register for a value. The register returned by *choose* $\dot{\phi} \hat{\phi} \omega \delta$ must not be in $\hat{\phi}$, and, it should be chosen with the objectives from p. 85, which we recap here:

1. preferably choose $\dot{\phi}$,
2. preferably avoid choosing registers from $\hat{\phi}$,
3. preferably avoid choosing registers that contain live values according to ω and δ ,
4. preferably choose the register with the smallest number that is known to be destroyed by the current strongly connected component.

These objectives are put into effect in the following way:

1. Using a heuristic function, *heur*, (described below) we first obtain a register ϕ according to objectives 2–4. If $\dot{\phi}$ is not allowable (i.e., $\dot{\phi} \in \hat{\phi}$) or there is no register (i.e., $\dot{\phi} = -_{\text{register}}$), we choose ϕ and not $\dot{\phi}$. Otherwise, if $\dot{\phi}$ is not in $\hat{\phi}$, we choose $\dot{\phi}$, thereby satisfying both 1 and 2. If, on the other hand, $\dot{\phi}$ is in $\hat{\phi}$ while ϕ is not, we elect to satisfy 2 and choose ϕ . If both are in $\hat{\phi}$, we cannot satisfy 2 and might as well satisfy 1 by choosing $\dot{\phi}$. Thus, *choose* is defined:

$choose \hat{\phi} \hat{\phi} \hat{\phi} \omega \delta = \text{let } \phi = \text{heur } \hat{\phi} \hat{\phi} \omega \delta \text{ in}$
 $\text{if } \hat{\phi} \in \hat{\phi} \quad \vee$
 $\hat{\phi} = \text{-register} \quad \vee$
 $\hat{\phi} \in \hat{\phi} \wedge \phi \notin \hat{\phi} \quad \text{then } \phi \text{ else } \hat{\phi}.$

II. We make $\text{heur } \hat{\phi} \hat{\phi} \omega \delta$ yield a register that is not in $\hat{\phi}$ while trying to satisfy objectives 2–3 above as follows:

1° Remove the set of forbidden registers $\hat{\phi}$ from the set of candidates.

2° Divide the set of remaining registers into subsets (illustrated in figure 26) that correspond to objectives 2–3 and choose a register from the best non-empty subset in the following way:

(i) Aiming at objective 2, divide the registers into two subsets, according to whether they are in $\hat{\phi}$ or not (the horizontal line in the figure).

(ii) Aiming at objective 3, divide the registers into two subsets, according to whether they contain live values or not (the thick vertical line in the figure).

(iii) Aiming at objective 4, divide the registers that do not contain live values according to whether they will be destroyed anyway or not (ϕ_{dirty} and ϕ_{clean} , respectively, in the figure). (How to approximate the set of registers that will be destroyed anyway is described in section 5.10.)

(iv) Finally, observe that when we are forced to evict a live value from its register (because all registers contain live values), it is better to evict a value that is in a register it is hostile to, than one that is in a register which it is not hostile to. Therefore we divide the registers that contain live values according to whether they contain *malplaced* or *wellplaced* values (ϕ_{mal} and ϕ_{well} , respectively, in the figure): A value is malplaced iff it is in a register it is hostile to, and wellplaced otherwise.

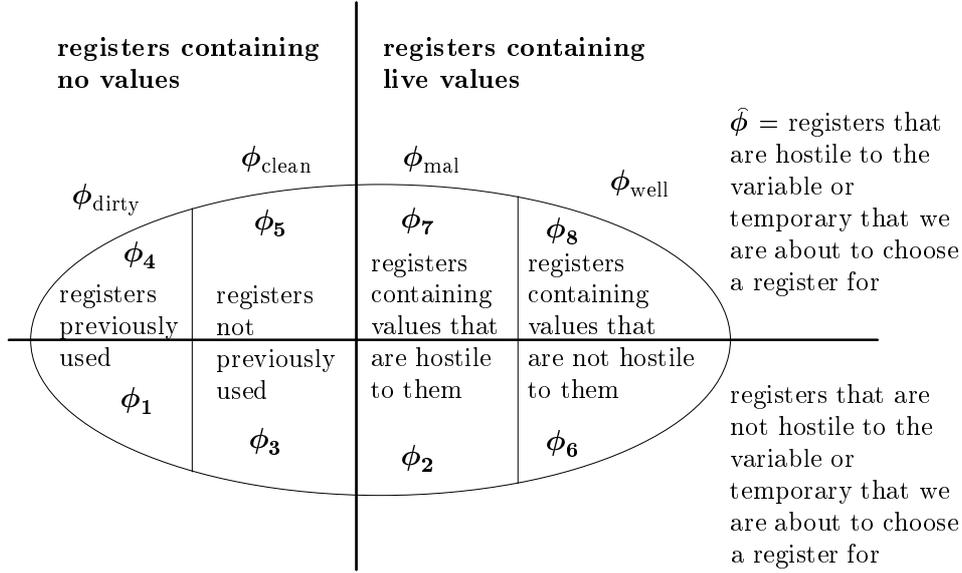


Fig. 26. *The division of registers.* The numbers of the subsets indicate the order in which registers are chosen, i.e., *heur* first tries to choose a register from ϕ_1 ($= \phi_{\text{dirty}} \setminus \hat{\phi}$); if this set is empty, it tries to choose a register from ϕ_2 ($= \phi_{\text{mal}} \setminus \hat{\phi}$); etc.

3° Now we have a division of the registers into pairwise disjoint subsets. In the figure these subsets are numbered in the order we prefer to choose registers from them. For instance, we prefer a register from ϕ_1 to one from ϕ_2 , because the registers in ϕ_1 do not contain live values, which the registers in ϕ_2 do. Choose the lowest numbered register from the lowest numbered, non-empty subset.

Some comments on our choice of ordering of the subsets:

Choosing a register in $\phi_1 = \phi_{\text{dirty}} \setminus \hat{\phi}$ re-uses a register in an attempt to minimise the total number of used registers.

If ϕ_1 is empty, a register in $\phi_2 = \phi_{\text{mal}} \setminus \hat{\phi}$ is chosen. The value in this register is going to be spilled anyway, and the value that is going to be placed in the register is not hostile to it, so it can profitably be replaced by another value.

Choosing a register in $\phi_3 = \phi_{\text{clean}} \setminus \hat{\phi}$ evicts no live value but uses a hitherto unused register.

The next two candidate sets are $\phi_4 = \phi_{\text{dirty}} \cap \hat{\phi}$, and $\phi_5 = \phi_{\text{clean}} \cap \hat{\phi}$. The registers in these sets will be destroyed at some later point in the program, but they do not contain live values.

Choosing a register in $\phi_6 = \phi_{\text{well}} \setminus \hat{\phi}$ will evict a value that already resides in a register, but at least the value that we are choosing a register for is not hostile to this register.

The heuristic presented here can be refined in numerous ways; for instance, it could take use counts into account.

6.8 Which values are spilled

When there are not enough registers for all values, we must *spill*, i.e., put some of them in memory. The three main questions are

- (a) which values are spilled?
- (b) where should we place the *store code* that stores a value in memory?
- (c) where should we place the *load code* that loads a value from memory?

This section discusses (a). The following discuss (b) and (c).

If we choose always to leave it to the function *choose*, described in the previous section, to choose a register, the answer to (a) is given. When *choose* is asked to find a register to hold a value v and it picks a register that holds some other value v' , it effectively evicts v' from its register. If v' is needed later, it must be loaded from memory. In other words, the combination of first evicting v' and later discovering that it is needed again is an implicit decision to spill v' .

This is a crude way to decide which values are spilled compared with the sophistication in other register allocators. Most are, however, based on graph colouring and the choice of which values to spill is intimately connected with how this graph colouring is done.

It is, however, perhaps of lesser importance in our register allocator than in others which values are spilled. To see this, consider the imperative language (Pascal) example

```
... b ...  
n := n - 117;  
while n < 86681582 do begin  
    n := n + a  
end;  
... b ...
```

A smart register allocator allocates the variables used in the loop to registers, i.e., n and a rather than b . In our register allocator, it is only possible to a lesser degree to be that smart, since loops are recursive functions and we do not allocate free variables to registers. In our source language, the loop above could be expressed

```
letrec while n = if n<86681582 then while (n+a)  
                else n  
at r119 in ... b ... while (n-117) ... b ... .
```

The one value used in the loop, a , is a free variable of **while** and hence has no chance of remaining in a register throughout the loop; it must be fetched from the closure in each iteration. The other value used in the loop, n , is the argument of **while** and will always be allocated to a register. In either case, our register allocator does not have an opportunity to choose otherwise.

Summing up, when deciding which values to spill, we make a locally optimal choice. This may result in choices that are not globally optimal, but this is perhaps less important in our register allocator, which has less freedom of choice anyway, because free variables are always fetched from the closure when first used within each function.

6.9 Placing the spill code

The subject of placing the spill code should not be neglected for the benefit of, e.g., the issues of packing values in as few registers as possible and choosing which values to spill. In this section, we first try to analyse how spill code should ideally be placed; then we discuss practical ways to do it.

In imperative languages, there is a concept of a current value of a variable. At each program point, the current value is kept either in a register or in memory. At transitions between a program point where the current value is in memory and a program point where it is in a register, the current value must be *loaded*. Conversely, at transitions between a program point where the current value is in a register and one where it is in memory, it must be *stored*.

This is different in a functional language: because variables cannot be updated, there is no concept of a current value that must be stored at every transition from a program point where it is in a register to a program point where it is not. The value need only be stored at most once: If there are sufficiently many registers the value is never stored in memory; otherwise, exactly one store is required. It must be loaded at all transitions from program points where it is not in a register to program points where it is. (Remember references are also simply values, and the memory cell the reference references is not eligible for allocation to a register.)

A framework for discussing spill code placement

One main goal when placing spill code is to place it such that it will be executed infrequently at run-time. We investigate how this is done by giving rules that describe the fundamental ways in which the execution frequency of spill code can be reduced by moving the spill code from one program point to another. The rules afford a framework in which to discuss how to place spill code; they do not provide an algorithm.

We will not discuss how to estimate, at compile-time, the execution frequency of a given program point. The spill code can also be placed with respect to other criteria than the execution frequency. It might, e.g., be profitable to place the load code as far from the use as possible because of load latency. This we will not do anything about (beyond hoping that instruction scheduling will help). Nor will we attempt placing spill code so as to free up registers; we assume the allocation of values to registers has been decided and will not be changed when the spill code is placed. A fourth discussion that we will not delve into is how the spill code placement influences the memory usage (moving spill code into a recursive function may make the

program use more memory). Fifth, the code size is affected by how much spill code is inserted; this is, however, clearly a problem of inferior importance compared with the importance of reducing the execution frequency of the spill code.

For brevity, we will only discuss where to place *store* code. The corresponding discussion for *load* code is similar in many aspects.

At control flow *forks*, this placement of store code

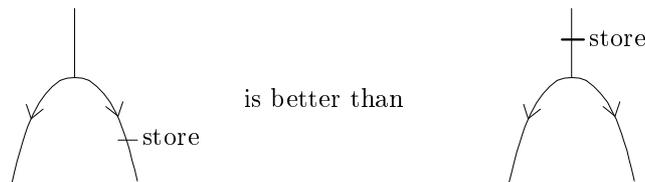


Fig. 27. *The fork rule.*

The first placement of the store code is better, because the store is only executed in the case control flows right, whereas it is executed in both cases with the second placement. Moving the store code is of course only permissible in some situations, e.g., if the left branch never accesses the stored value, or it stays unharmed in some register.

At *straight-line* control flow, this placement of store code

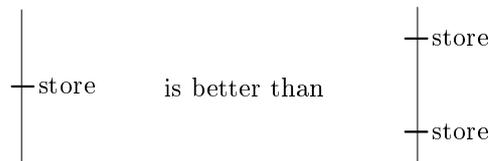


Fig. 28. *The straight-line rule.*

It is clearly not necessary to store the same value twice.

At control flow *meets*, this placement of store code

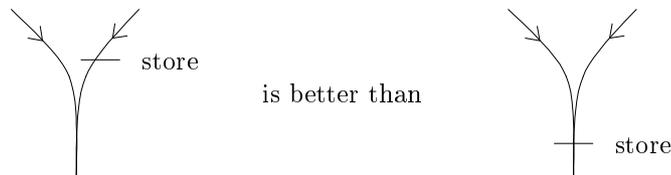


Fig. 29. *The meet rule.*

Analogously to the fork rule, it is profitable to push the store code into a part of the control flow graph that is executed less frequently at run-time.

These rules describe the only benefits with respect to execution frequency that can be gained by moving store code; all other moving of store code is only profitable insofar as it allows the subsequent application of some of the three rules above. The straight-line rule reduces the number of stores executed at run-time by reducing the number of (static) stores in the code. The fork and meet rules reduce the number of (dynamic) stores executed at run-time by moving the store code into less frequently executed parts of the code.

We have deliberately not said when it is permissible to apply a rule; this depends on the concrete situation. There are some examples below.

Using the framework

To stress that this framework can also be used to discuss how to place store code *inter-procedurally*, we have complicated the examples below with functions. In practice, moving store code for a register ϕ inter-procedurally is done by changing ϕ 's caller-save/callee-save status in the convention for the function: Store code is moved *into* λ (*down* in the call graph) by changing ϕ from a caller-save to a callee-save register for λ . Moving the store code *out* of λ (*up* in the call graph) is, dually, achieved by changing ϕ from a callee-save to a caller-save.

An example of a control flow *fork* is the code for an *if*-expression:

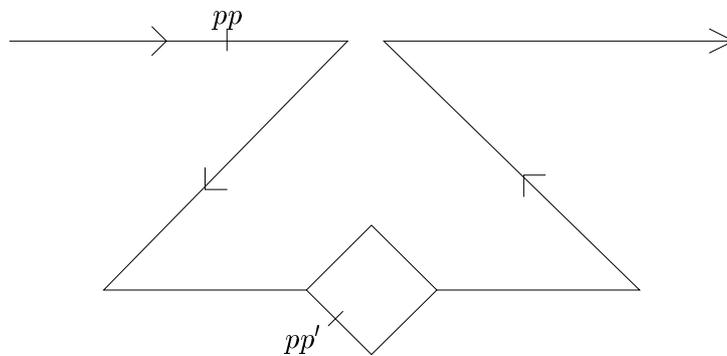


Fig. 30. An *if*-expression inside a function. The upper horizontal lines are the control flow in the caller, before and after the call, respectively. The lower lines are the control flow within the callee.

Assume the store code is at the program point pp . A better placement is obtained by moving it into the function and then, using the fork rule above, moving it into one of the forks, to pp' .

This moving of store code is permissible if the register is only destroyed in one of the forks and it is not already destroyed at pp' .

As an example of an application of the straight-line rule, consider this control flow, where two functions are called sequentially.

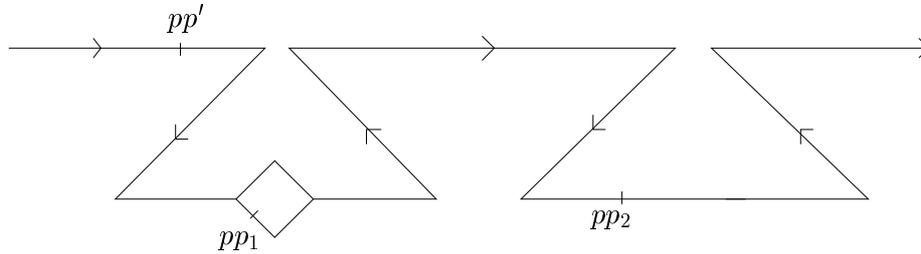


Fig. 31. The store code is better placed at pp' than at both pp_1 and pp_2 .

Here it is profitable to move the store code at pp_1 out of the fork (contrary to the advice of the fork rule) and up from the function, and likewise, move the store code at pp_2 up, for then the two stores can be collapsed according to the straight-line rule; i.e., we need only store at pp' .

Notice the better store code placement is here obtained by, locally, moving store code unoptimally. The rules are local in the sense that applying a rule will locally improve the placement, but it may hinder further applications of rules that might have given a better placement overall.

The move of store code above is only permissible if the register contains the right value at pp' .

Function calls, where control flows from the callers to the called function, provide a pregnant example of a control flow *meet*:

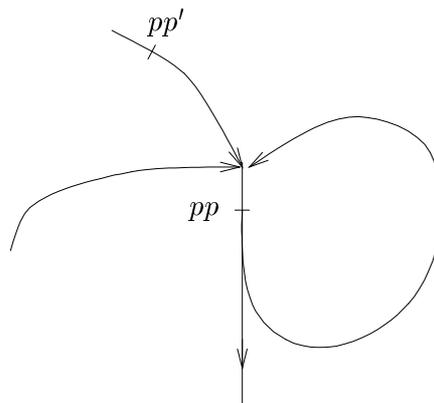


Fig. 32. An example of the application of the *meet* rule. A function calls itself recursively and is called from two other applications.

The store code is profitably moved from pp to pp' ; especially in this example, where it implies moving the store code out of a loop.

As in the previous examples, the moving of store code is only permissible if certain conditions are met. The reason it is permissible here might be that only one of the callers wants the register preserved after the call.

These three examples capture the situations we should consider. We will not consider the rather complicated control flow entailed by exceptions; it is discussed in section 8.10.

To conclude, the guidelines for placing store code are that it should be moved as deeply into `if`-expressions as possible and thereby into infrequently executed branches, and it should be moved as high in the call graph as possible thereby moving it out of loops or collapsing it with other store code.

To learn more exactly how to achieve this informal goal, we consider two questions: an intra-procedural question: how should we move store code into `if`-expressions within a function; and an inter-procedural question: when should we move store code up in the call graph, and when down.

6.10 Placing spill code intra-procedurally

The approach of (Chow, 1988) to moving store code into `if`-expressions within a function is a technique called *shrink wrapping*, which basically is using the fork rule as much as possible. Here is an example of how Chow places store code:

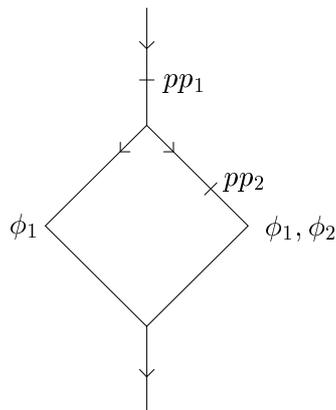


Fig. 33. *Chow's shrink wrapping.* The " ϕ_1, ϕ_2 " at the right fork indicates that these two registers are destroyed in that fork, etc. Chow saves ϕ_2 at pp_2 , because it is only destroyed by the rightmost fork. In comparison, he saves ϕ_1 at pp_1 because it is destroyed in both forks.

Chow inserts the store code at the earliest program point where the register must inevitably be saved in all possible execution paths leading from that point.

An uncertain effect of the shrink wrapping is that it may increase the number of (static) stores in the code, i.e., do the opposite of the straight-line rule above:

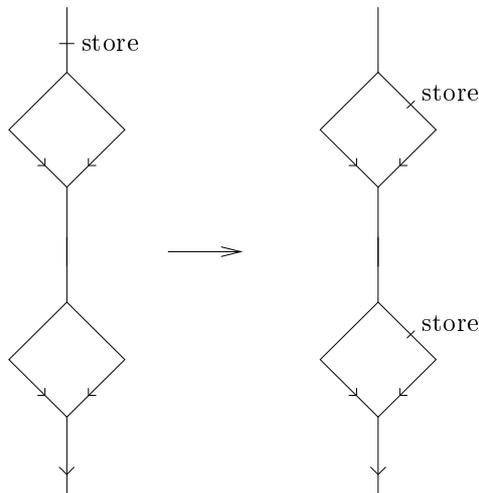


Fig. 34. Depending on the actual execution frequencies of the forks, the shrink wrapping may here improve, preserve, or worsen the execution time of the code.

(Because his language is imperative, Chow can have loops inside a function. If he does not want to move store code inside a loop, he cannot ruthlessly shrink wrap all store code; he must explicitly disallow that store code shrink wraps into a loop. This would not be a concern of ours, if we were to apply shrink wrapping inside functions, as loops are inter-procedural phenomena in our language.)

The *lazy-save* store code placement strategy of (Burger et al., 1995), seems to do about the same as Chow’s shrink wrapping, since both insert the store code at the earliest program point where the register must inevitably be saved in all possible execution paths leading from that point. A difference is that their source language is functional (it is Scheme), as ours, where Chow’s is imperative, with the implications discussed above (p. 93).

One of the main goals of Callahan and Koblenz (1991) was to place spill code in a smart way as we discussed in section 6.1; see also section 6.13.

6.11 Placing spill code inter-procedurally

When should we move store code down in the call graph, hoping that it will eventually end up within some infrequently executed branch of an *if*-expression, and when should it be moved up in the call graph, hoping that it will move out of a loop?

Some guidelines can be given for when a register should be caller-save and when it should be callee-save: Registers that are destroyed in *all* control flow paths through the code of λ (as ϕ_1 in figure 33 above) should be *caller*-save for λ ; for in that case we might as well leave it to the callers of the function to save the register, hoping that this will move the store code out of a loop.

Registers that are not destroyed in *any* control flow path through λ (as ϕ_3 in figure 33) should be *callee-save*.

The problem is the registers that are destroyed in some but not all control flow paths through λ : should ϕ_2 in figure 33 be callee-save or caller-save in the convention for λ ?

Chow chooses the former: registers that are only destroyed in some control flow paths through the function (as ϕ_2) will be callee-save registers. This he implements by shrink wrapping store code as much as possible in the current function and then making the registers that are still stored outermost caller-save; all other registers are callee-save.

The disadvantage of this is that the storing of ϕ_2 , although it is only done in some execution paths through λ , might still be done more efficiently higher up in the call graph. Figure 31 above provides an example where the caller is able to do better than the two callees.

The other solution, to make all registers that may be destroyed in some execution path through λ caller-save, is simpler than the first, for it means that all responsibility of saving registers is pushed onto the caller—the callee-save registers of λ are simply the registers that are not destroyed in any execution path through λ .

The disadvantage is that store code will never be moved inside forks.

The best strategy would be something in between the two extremes above: make ϕ_2 a caller-save register if the callers of λ could place the store code better, and make it a callee-save register otherwise. The problem with this is first of all that inter-procedural information is needed. In our register allocator, for instance, inter-procedural information about *callers* is not readily available since the callees are usually processed before the callers. Second, in general, it is not clear exactly when it is more profitable to let the caller, rather than the callee, save the register, although figure 31 provides an example where it is obvious. Third, since λ may have more than one caller, the inter-procedural information might be inconclusive: some callers might prefer ϕ_2 as a callee-save register, while others might prefer the opposite.

From these three alternatives, numerous combinations can be concocted; e.g., take one of the extremes but combine it with the third strategy in certain easily handled situations.

Chow's experiments show only a small improvement from the shrink wrapping and moving of spill code up the call graph, but this is perhaps because his languages are imperative. In a functional language, where inter-procedural register allocation is expected to give relatively larger improvements, this optimisation might yield more.

Santhanam and Odnert also move spill code between functions. They divide the call graph into *clusters*. Roughly, a cluster is a sub-graph of the call graph which has a cluster root node (function) through which all paths to functions in the cluster must pass. The idea in dividing the call graph into clusters is that the responsibility for saving a register can be moved upwards in the call graph from the functions within the cluster to the cluster root function. Deciding how to divide the call graph into clusters then means

deciding where to place spill code. Santhanam and Odnert use estimations of call frequencies to decide how to divide the call graph into clusters: if the cluster root node is called less frequently than functions within the cluster, the spill code will be executed less frequently.

We shall choose the simplest of all proposed strategies: to always make all registers that could possibly be destroyed by λ caller-save registers of λ . This means that there will be no moving store code into forks. With this approach, the register allocation of each function is *selfish* in the following sense: it takes care of its own values by trying to put them in registers that are not destroyed by the functions it calls, and if this is not possible, it saves them on the stack. It does not save registers merely for the benefit of its callers. (There is, however, one extenuating circumstance: it does try to use as few registers in all as possible.)

This strategy favours the lowest nodes in the call graph. We assume many programs will spend most of the time in the lower parts of the call graph.

6.12 Our spill code placement strategies

Having discussed some of the problems in placing spill code, we will decide and explain the exact way we choose to do it in the following.

Placing store code: the producer-saves strategy

We choose the following way to place the store code, which is particularly simple in our register allocator. If x of `let $x = e_1$ in e_2` is spilled in e_2 , we generate the following code:

$$\phi_x := \boxed{\text{code to evaluate } e_1} ; \text{push } \phi_x ; \phi := \boxed{\text{code to evaluate } e_2} ; \text{pop.}$$

The value x can only be used in e_2 ; so after the evaluation of e_2 , the value saved on the stack is not needed any more, and it is discarded with a `pop` (it is not restored with a `pop ϕ_x`).

The main advantage of this strategy for placing store code is its simplicity: it leaves the responsibility of storing a spilled value exactly one place, namely with the *producer* of the value, and we will call it a *producer-saves* store code placement strategy. The producer of a value can be a sub-expression of the program. For instance, `let $x = e_1$ in e_2` is the producer of the value x . The spill code can simply be inserted when we generate code for the producer, instead of in a separate phase. Because the code to allocate and deallocate memory for a spilled value is inserted around the code for a sub-expression, it will obey a stack discipline.

The disadvantage is that this strategy violates many of the concerns discussed above. The store code is inserted at a program point without regard to the probable execution frequency of that program point; it might, for instance, be placed in a loop. The producer-saves strategy, however, at least ensures that the store code always appears only once in the control flow graph.

See also the case study of the producer-saves store code placement strategy in the Fibonacci function in the assessment section 11.9.

We use the producer-saves strategy for other kinds of values as well. For instance, the producer of a ρ is a `letregion`-expression; the producer of a y is a λ -abstraction; the producer of an f is a `letrec`-expression.

Saving registers because of recursion

There are two known schemes for saving registers when there is recursion.

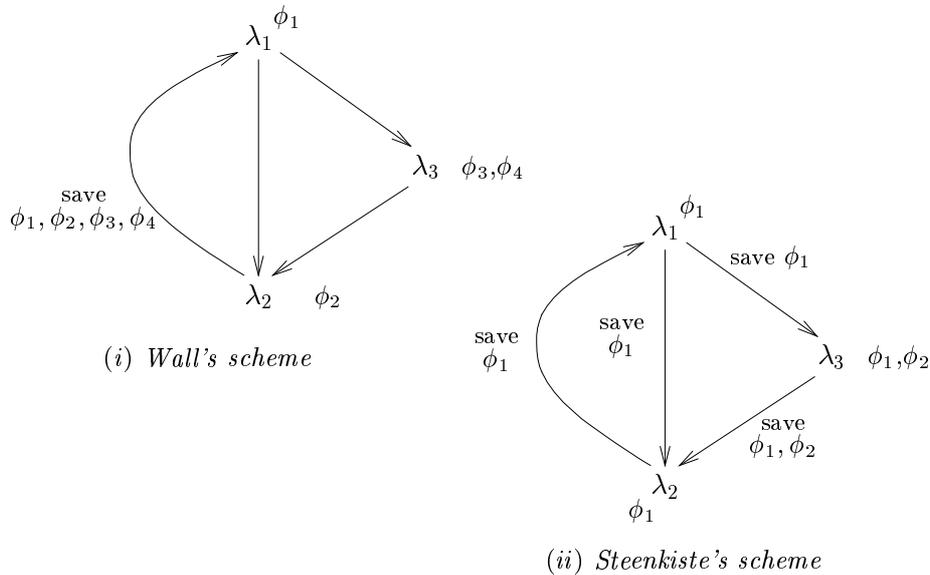


Fig. 35. Schemes for saving registers at recursion. The registers beside a λ indicate which registers are destroyed by that λ ; edges are annotated with the set of registers that are saved at calls.

In Wall's scheme, all registers used by λ 's in the strongly connected component are saved at back edges in the call graph.

In Steenkiste's scheme, each λ saves the registers it uses. The disadvantage is that one pays for the use of recursion at all calls in the strongly connected component although it might be seldom that the program actually recurses. In Wall's scheme, chances are greater that registers are only saved when recursion occurs: Control might flow through all the λ 's above without any of them calling each other recursively, and then no registers will be saved in Wall's scheme, whereas registers will be unnecessarily saved in Steenkiste's.

On the other hand, also Wall's scheme may save more registers than necessary: If λ_1 and λ_2 keep calling each other recursively, ϕ_3 and ϕ_4 are saved unnecessarily each time λ_2 calls λ_1 .

Wall's source languages are imperative, and he actually chooses this scheme, rather than one resembling Steenkiste's, on the assumption that

many apparently recursive programs are so only in exceptional cases. We assume the opposite is the case in our functional language.

In Steenkiste’s scheme, the total use of registers in the component can be smaller (two registers in the example, where Wall uses four): The same registers can be used in the different λ ’s in the component, as the registers are saved at recursive calls anyway.

We choose Steenkiste’s scheme, because it meshes nicely with the producer-saves store code placement strategy. All potentially recursive applications are treated the same way: they destroy all registers containing live values. Then, the producer’s responsibility of saving the value it produces will also take care of saving the values that must be saved because of recursion. If we used Wall’s scheme, the producer-saves mechanism could not handle the saving necessary at the back-edge calls.

Placing load code: the user-loads strategy

The considerations concerning where to place load code are similar in many aspects to the considerations above concerning where to place store code. This will not be explored in depth here. We only remark the following: First, in the cases where one wants the storing and loading to obey a stack discipline, one will want the placement of load code to mirror the placement of the store code. Second, in the cases where this is not a concern (e.g., when placing the load code for a spilled `let`-bound value x), one may try to place the load code at the earliest possible program point where it is inevitable that the value must be loaded—dually to when placing store code. Burger et al., (1995), nevertheless, suggest an “eager” reloading scheme, where a value is instead loaded at the first program point from which some execution path leads to a use of the value. Although this means that values are sometimes loaded unnecessarily, their experiments show that it is as good as the other load code placement strategy. They explain this by conjecturing that the cost of unnecessary reloads is offset by the reduced effect of memory latency.

As with the store code placement strategy, we will not attempt the fancy solution. We will also place the load code in the way that is simplest in our register allocator: the code to reload a spilled value is placed at the uses where the value is not in a register.

When generating code for a use of x in e_2 of `let $x = e_1$ in e_2` , we confer with the current δ to see whether x is allocated to some register. If x has been thrown out of its register (because the register was needed for something else), we have to reload x . So the translation of a use of x is

$$\begin{aligned} \llbracket x \rrbracket_{\text{ra}} \delta &= \text{if } x \text{ is in some } \phi_x \text{ according to } \delta \\ &\quad \text{then } (\delta, \phi_x, \lambda\phi. \langle \phi := \phi_x \rangle) \\ &\quad \text{else } \llbracket x \rrbracket_{\text{load}} \varnothing -_{\text{register}} \omega\delta, \end{aligned}$$

where ω is the ω -information before the expression x , and the *optional move* is defined

$$\langle \phi := \phi' \rangle = \text{if } \phi = \phi' \text{ then } \epsilon \text{ else } \phi := \phi',$$

and $\llbracket x \rrbracket_{\text{load}}$ will return code to reload x .

A value x is *loaded in e* if it is used at a point in e where it is not in a register according to the δ at that point. Hence, it can be determined during register allocation of e whether a value is loaded in e : x is loaded iff $\llbracket \cdot \rrbracket_{\text{load}}$ was called with x . We extend δ to also have a component, $\delta^v \subseteq X$, that records the set of values that are loaded; i.e., x is loaded according to δ iff $x \in \delta^v$. Denote with $\llbracket v \rrbracket_{\text{has-been-loaded}}$ δ the δ updated to record that x is loaded.

The expression $\llbracket x \rrbracket_{\text{load}} \hat{\phi} \hat{\phi} \omega \delta$ will at first pick a register, ϕ_x , for x , and update δ to record that ϕ_x from now on contains x . This is just what $\llbracket x \rrbracket_{\text{def}} \hat{\phi} \hat{\phi} \omega \delta$, which we have already defined (p. 88), does—intuitively a point in the code where a value is reloaded is like a definition of that value. Furthermore, $\llbracket x \rrbracket_{\text{load}} \hat{\phi} \hat{\phi} \omega \delta$ records in δ that x is loaded and returns code to load x :

$$\begin{aligned} \llbracket x \rrbracket_{\text{load}} \hat{\phi} \hat{\phi} \omega \delta &= \text{let } (\delta, \phi_x) = \llbracket x \rrbracket_{\text{def}} \hat{\phi} \hat{\phi} \omega \delta \\ &\quad \delta = \llbracket x \rrbracket_{\text{has-been-loaded}} \delta \\ &\quad \beta = \lambda\phi. \phi_x := \boxed{\text{code to access } x \text{ in memory}}; \\ &\quad \langle \phi := \phi_x \rangle \\ &\text{in } (\delta, \phi_x, \beta). \end{aligned}$$

Notice that the code does not simply load the value into the destination register, ϕ , provided by the context. Rather, the value is loaded into the chosen register, ϕ_x , and then moved into ϕ . This is because we expect *choose*'s choice of register (ϕ_x) to be more farsighted than the context's choice (ϕ). Chances are bigger that we do not have to reload x if it is put in ϕ_x than if it is put in ϕ , because ϕ_x has been chosen specially for x ; and we prefer the extra cost of the move instruction $\phi := \phi_x$ to the potential cost of a reload.

Code is generated with respect to a stack shape

To keep track, at compile-time, of the position on the stack of values that are loaded, each expression is compiled with respect to a *stack shape* $\varsigma = (\varsigma^e, \varsigma^p) \in \mathbf{S}$. This contains a *compile-time stack pointer*, $\varsigma^p \in I$, and an environment $\varsigma^e \in V \rightarrow I$ that maps values to their stack position.

If x of **let** $x = e_1$ **in** e_2 is loaded in e_2 , the code for e_2 is generated with respect to a stack shape in which x has been pushed: If the code for the whole **let**-expression is generated with respect to a stack shape $\varsigma = (\varsigma^e, \varsigma^p)$, the code for e_2 is generated with respect to the stack shape

$$(\varsigma^e + \{x \mapsto \varsigma^p\}, \varsigma^p + 1).$$

This reflects that the stack at the entry to the code for e_2 has changed compared to what it was at the entry to the code for the whole **let**-expression. One element, x , has been pushed, hence the $\varsigma^p + 1$, and it resides at offset ς^p , hence the $x \mapsto \varsigma^p$.

When generating code to load x in some sub-expression of e_2 , one can find x 's offset on the stack using the stack shape $(\zeta^{e'}, \zeta^{p'})$ at that point. The offset is the difference between the compile-time stack pointer, $\zeta^{p'}$, at that point and the compile-time stack pointer, ζ^p , at the point when code was generated for pushing x on the stack. The latter is accessible as $\zeta^{e'}x$. Thus, the code to access x in memory above is $m[\phi_{sp} - \iota]$, where $\iota = \llbracket \zeta^{p'} - \zeta^{e'}x \rrbracket_{I \rightarrow I}$.

Summing up, the code we generate must be abstracted over a stack shape (and not only over the destination register), i.e., the β 's are abstracted over (ζ^e, ζ^p) . The β for `let $x = e_1$ in e_2` where x is loaded in e_2 is

$$\beta = \lambda\phi. \lambda\zeta. \beta_1\phi_x\zeta ; \text{push } \phi_x ; \beta_2\phi(\zeta^e + \{x \mapsto \zeta^p\}, \zeta^p + 1) ; \text{pop}.$$

The code $\beta_1\phi_x$ for e_1 is translated with respect to the same stack shape ζ as the whole `let`-expression, while the code $\beta_2\phi$ for e_2 is translated with a stack shape that reflects that x is pushed around that code.

We use ζ for code abstracted over a stack shape, i.e., $\zeta \in \mathbf{Z} = \mathbf{S} \rightarrow \mathbf{K}$ (and then $\beta \in \mathbf{B} = \Phi \rightarrow \mathbf{Z}$).

A *preserver* of ϕ is a function $p \in \mathbf{Z} \rightarrow \mathbf{Z}$ that takes some code ζ and returns code ζ' which preserves ϕ on the stack around ζ :

$$p\zeta = \lambda\zeta. \text{push } \phi ; \zeta(\zeta^e + \{x \mapsto \zeta^p\}, \zeta^p + 1) ; \text{pop}.$$

If p_x is a preserver of ϕ_x , the β above can be written

$$\beta = \lambda\phi. \lambda\zeta. \beta_1\phi_x\zeta ; p_x(\beta_2\phi)\zeta.$$

To rid the notation of the stack shape, use `;` to glue together ζ 's as `;` glues together κ 's:

$$\zeta_1 ; \zeta_2 = \lambda\zeta. \zeta_1\zeta ; \zeta_2\zeta.$$

Then β is

$$\beta = \lambda\phi. \beta_1\phi_x ; p_x(\beta_2\phi).$$

We can define $\llbracket x \rrbracket_{\text{kill}} \phi_x \delta$ to yield a pair (δ_x, p_x) , where p_x is a preserver of ϕ_x if x is loaded according to δ , and δ_x is δ with x removed from the set of loaded, i.e., $\delta_x^v = \delta^v \setminus \{x\}$. If x is not loaded according to δ , p_x does not preserve ϕ_x , i.e., $p_x = \lambda\zeta. \zeta$.

Then, finally,

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} \delta &= \text{let } (\delta, \dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \delta \\ &\quad (\delta, \phi_x) = \llbracket x \rrbracket_{\text{def}} \dot{\phi}_1 \delta \omega_2 \\ &\quad (\delta, \dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \delta \\ &\quad (\delta, p_x) = \llbracket x \rrbracket_{\text{kill}} \phi_x \delta \\ &\quad \beta = \lambda\phi. \beta_1\phi_x ; p_x(\beta_2\phi) \\ &\text{in } (\delta, \dot{\phi}_2, \beta). \end{aligned}$$

6.13 Comparison with other approaches

Using the structure of the source program

The main difference between the bulk of other register allocators and ours is that we do not use graph colouring.

We wanted to see whether it was possible to make a good register allocation and avoid graph colouring by using information in the source language. That we have developed a translation (in this and the following chapters) and the measurements in chapter 11 indicate that the approach works. (Actually, the measurements only indicate that the specific conglomerate of the inter-procedural register allocation, the source level per-function register allocation, region inference, etc. found in our compiler works.) In another sense, however, we did not succeed.

We hoped that, by taking the structure of the program into account, we could obtain the same global picture of the register allocation problem as an interference graph gives, and solve the optimisation problem it represents without explicitly building and colouring an interference graph.

Our inspiration to do this was from Callahan and Koblenz (1991), whose *hierarchical* register allocation uses the structure of the program to place spill code in a smart way and to allow a value to be spilled in some parts of the program and in registers in other parts and even in different registers in different parts of the program. They build a tree of “tiles” that cover the basic blocks of the control flow graph; the tile tree reflects the program’s hierarchical control structure. Avoiding graph colouring is not their goal: Registers are allocated for each tile using a standard graph-colouring algorithm. The local allocation and interference information is passed around the tile tree by first processing it bottom-up and then top-down. Thereby their register allocation can be sensitive to local usage patterns while retaining a global perspective.

Our idea was to adapt their algorithm to use the syntax tree of a source program as the tile tree. Our tiles would then be the nodes of the syntax tree. Construction and colouring of an interference graph for every tile might be unnecessary because of the extremely simple structure of our tiles; and indeed, it would not be practically possible to do graph colouring at every tile, because we would have so many more tiles than they.

The benefits we hoped to gain by doing this are: 1° avoiding graph colouring but retaining the global picture; 2° reaping the general benefits in their algorithm mentioned above (smart spill code placement, etc.). The question is whether their algorithm can be adapted such that it gives as good a result without graph colouring as with.

We did not succeed in adapting their algorithm; the per-function register allocation presented in this chapter and chapter 8 simply chooses registers in a linear traversal of the program. The ω -information gives a global picture, but the linear traversal of the program and not the structure of the program determines the order in which choices are made.

An example where this algorithm does worse than what a graph-colouring

register allocator should be able to do is: If we have more values than registers and the values are used successively, our register allocator might in principle accidentally throw out the value needed next each time it fetches a value, and thus generate code that loads a value every time it is needed. Graph-colouring register allocation, with its more global perspective, would instead decide once for all which values to spill, and thus generate code that only loads as many times as the spilled values are used. On the other hand, our register allocator has the benefit over graph-colouring register allocation that values can be in different places in different parts of the program, although this is not exploited systematically because of the lack of global perspective.

Although one can construct examples where our register allocator does badly because it does not have the global picture an interference graph gives, one must remember that building an interference graph does not automatically give a solution; the graph must also be coloured, and the heuristics to do that will likely also behave badly on some examples. Possibly, our more simple-minded way of choosing registers for values does as good in practice as many heuristics for colouring graphs.

Furthermore, basic graph-colouring register allocation gives a way of choosing registers for values and a way of deciding which values must be kept in memory, but the things it is not good at (e.g., allocating a value to different registers in different parts of the program; placing the spill code) may well influence the quality of the register allocation more.

Thorup (1995) presents an algorithm that, by using the structure of the program, can “colour the interference graph implicitly”, i.e., without explicitly building it. His work concerns the problem of packing values in as *few* cells as possible, i.e., not in a *fixed* number of cells (registers); so it is not the same as register allocation, and he does not consider how to choose what values to spill and where to place spill code, etc.

Interestingly, his way to the idea of using the structure of the program is completely different from ours. It goes over the graph theoretic concept of tree-width: Thorup observes that the control flow graph of a structured program has small tree-width, and the *intersection graph* of a control flow graph with small tree-width (which is the same as the interference graph) is easier to colour. His algorithm to do it uses the syntax of the program to assign “colours” to live ranges; an interference graph is never explicitly built. In this sense, he succeeds in using the structure of the program to “do interference graph colouring” without explicitly building and colouring an interference graph.

Thorup proves that his algorithm will colour a graph using at most a fixed number as many colours as an optimal colouring would use. The fixed number depends on the tree-width of the control flow graph.

Kannan and Proebsting (1995) also use the structure of the program to make a better graph colouring heuristic. Thorup’s work is a generalisation of their work. It seems that the inspiration of Kannan and Proebsting is neither (Callahan and Koblenz, 1991) nor the concept of tree-width.

Norris and Pollock (1994) investigate another way to adapt the hierar-

chical register allocation of Callahan and Koblenz: they use the *program dependence graph* as the hierarchy. The program dependence graph represents both data and control flow dependencies, and it expresses only the essential partial ordering of statements in a program that must be followed to preserve the semantics of the program. A benefit from using the program dependence graph for the hierarchical register allocation is that it can also be used for other optimisations.

The way the three requirements to an input language to a register allocator are formulated in section 6.1, suggests that they must be satisfied by making the language sufficiently high-level, especially if the register allocation is inter-procedural. Wall (1986) has gone in the other direction in his inter-procedural register allocation. His input language to the register allocator is annotated code. It can be immediately converted to executable code by throwing away the annotations. The annotations indicate how the code should be transformed after an (optional) register allocation phase. Here is an example of his annotations:

$$\begin{array}{ll} \phi_1 := m[\phi_7 + y] & \text{remove.y} \\ \phi_2 := m[\phi_7 + z] & \text{remove.z} \\ \phi_3 := \phi_2 + \phi_2 & \text{op1.y op2.z result.x} \\ m[\phi_7 + x] := \phi_3 & \text{remove.x} \end{array}$$

The annotations are actions qualified by variables. They specify how the instruction should be changed, if the variable is allocated to a register: “remove.y” means that the instruction should be removed if y is allocated to a register; and “op1.y” means to replace the first operand of the addition instruction with the register allocated to y , if y is allocated to a register. Our third requirement (p. 79), that function calls should be recognisable, Wall satisfies partly by the annotations and partly by explicitly having the call graph with the annotated code.

We could have gone the same way, and have devised analogous annotations; e.g., “remove this instruction, if the argument is passed on the stack”.

Eliminating move instructions

Our algorithm tries to avoid register-to-register transfers by letting information flow both upwards in the syntax tree (from sub-expression to context) and downwards (from context to sub-expression) when translating expressions, as described in section 6.2. The method is related to Reynolds’ (1995) way of deciding when to introduce new temporary variables when he generates intermediate code.

In register allocators based on graph colouring, register-to-register transfers are traditionally avoided by *coalescing* live ranges (nodes) in the interference graph: A move instruction can be eliminated from the program if there is no edge in the interference graph between its source and destination, and then the source and destination live ranges in the interference graph can

be coalesced into one node. The edges of the coalesced node is the union of the edges of the two coalesced live ranges.

Chaitin aggressively coalesces as much as possible. This eliminates move instructions at the cost of making the interference graph harder to colour, because the coalesced nodes have more edges than the original nodes.

Chaitin's coalescing strategy is too aggressive in the sense that it can make a graph that can be coloured with k colours into a graph that cannot be coloured with k colours. To mend this, Briggs et al. (1994) introduce a more conservative strategy, which only coalesces nodes in cases where a guarantee can be found that the coalescing will not make a colourable graph uncolourable.

The coalescing strategy of Briggs et al., on the other hand, is too conservative, and George and Appel (1995) present a coalescing strategy that is less conservative than that of Briggs et al., but still not as aggressive as Chaitin's. They report an impressive speedup of 4.4% solely from eliminating moves, and this is over an algorithm that already tries to eliminate moves. This suggests that eliminating move instructions is important. This improvement, however, is in a compiler that generates many move instructions, unlike ours.

These ways of eliminating moves are not applicable in our register allocation algorithm, as they are intimately linked with graph colouring.

Chow and Hennessy (1990) go in the opposite direction from eliminating move instructions: they introduce move instructions in the program to make the interference graph easier to colour. When a graph is uncolourable, they try to split a live range (node) that has many edges into smaller live ranges, each of which has fewer edges than the original live range, thereby increasing the chances that the interference graph will be colourable.

The very ad hoc way we assign values to registers actually means that our register allocator allows a value to reside in different places in different parts of the program; it does not utilise this in any systematic way, however.

Callahan and Koblenz (1991) try to avoid register-to-register transfers by preferring values to registers in the bottom-up pass over the tree of tiles. For instance, a value that is passed as argument to a function is preferred to the argument register of the function. Our allocator could be extended to do this in the ω -analysis and the preferring information could be a part of the ω 's.

Intermediate code generation

We have focused on the issues pertaining to generating code for a higher-order, strict functional language, and not so much on the general issues that are always important when generating code, consequently we shall not refer to the large body of work on general code generation issues.

When compiling an expression to intermediate code in the LISP compiler EPIC, Kessler et al. (1986) use information about what context that expression is in, like we use context information to decide what the destination

register should be, by applying a β to a destination register.

It is standard code generation policy to generate rather naive code, and rely on an ensuing optimisation phase to tighten up the code (see, e.g., (George and Appel, 1995)). We try to avoid generating obviously inefficient code (e.g. by eliminating redundant moves). Partly this is necessary, because we do the register allocation before the code generation: If the register allocation runs after the code generation, the code generator can use as many temporary variables as it likes; we must economise on the registers. Also, we think it is conceptually nicer to generate the “right” code right away and not have to tighten up the code afterwards. In this respect, our approach is akin to Reynolds’ (1995) generation of efficient intermediate code.

Although we think, e.g., the “ β ” of our translation—“some code with a hole in it for the destination register”—and the “optional move”, $\langle \phi := \phi' \rangle$, are natural concepts, generating efficient code in one go nevertheless complicates the translation. On the other hand, an ensuing optimisation phase will also be more complicated if it is rigged to fix the shortcomings of a specific code generator—especially so, if it relies on the fact that the code being optimised comes from a specific code generator.

Because our translation is so closely coupled to the structure of the program, the code we generate evaluates expressions in exactly the order they appear in the program. To minimise the use of registers for temporary values it is profitable to reorder the evaluation of expressions (Sethi and Ullman, 1970): If one register is needed by the code for e_1 , and two are needed by the code for e_2 , the code for e_1+e_2 will need three registers in all, because the result of e_1 must be held in a third register while e_2 is being evaluated. If we could treat the expression as e_2+e_1 , the total need for registers would only be two, because the result of e_1 is not live while e_2 is being evaluated. Another example is (Burger et al., 1995) which changes the evaluation order of function arguments to make a good “shuffling” of registers at function applications.

Since the semantics of SML specifies left-to-right evaluation order, we can only change the evaluation order of expressions if they do not contain side-effects, and in SML, even the simplest expressions may contain side-effects; for instance, $1+1$ contains a potential side-effect: evaluating it may raise a **Sum** exception. Therefore, changing the evaluation order can probably not be done as often in SML as in, e.g., Scheme.

7 Development of the inter-procedural part of the algorithm

7.1 Overview of the back end

The translation $\llbracket \cdot \rrbracket_{\text{compile}} \in E \rightarrow \mathfrak{P}$ from our source language to PA-RISC assembly language consists of the following phases:

$$\begin{array}{ccccccccccc}
 E & \xrightarrow{\llbracket \cdot \rrbracket_{\text{ca}}} & \hat{E} & \xrightarrow{\llbracket \cdot \rrbracket_{\text{sib}}} & \hat{E} & \xrightarrow{\llbracket \cdot \rrbracket_{\text{cr}}} & \dot{E} & \xrightarrow{\llbracket \cdot \rrbracket_{\text{sa}}} & \dot{E} & \xrightarrow{\llbracket \cdot \rrbracket_{\text{cg}}} & ? & \xrightarrow{\text{scs}} & \Gamma \\
 & & & & & & & & & & & & \\
 \Gamma & \xrightarrow{\text{rdfs}} & \mathbf{K} & \xrightarrow{\llbracket \cdot \rrbracket_{\text{bbs}}} & \mathcal{PB} & \xrightarrow{\text{lin}} & \hat{\mathbf{K}} & \xrightarrow{\llbracket \cdot \rrbracket_{\text{pa}}} & \mathfrak{P} & \xrightarrow{\llbracket \cdot \rrbracket_{\text{sched.}}} & \mathfrak{P}
 \end{array}$$

$\llbracket \cdot \rrbracket_{\text{ca}}$	closure analysis—annotate each application with a set λ of functions that may be applied.
$\llbracket \cdot \rrbracket_{\text{sib}}$	convert the name f of each letrec -function to its sibling name f .
$\llbracket \cdot \rrbracket_{\text{cr}}$	assign offsets in the closure to the free variables for each λ .
$\llbracket \cdot \rrbracket_{\text{sa}}$	convert functions to functions of several arguments.
$\llbracket \cdot \rrbracket_{\text{cg}}$	build the call graph.
<i>scs</i>	find strongly connected components in the call graph.
<i>rdfs</i>	take care of the actual translation, i.e., the register allocation and code generation.
$\llbracket \cdot \rrbracket_{\text{bbs}}$	convert the structured language \mathbf{K} to a set of basic blocks.
<i>lin</i>	convert a set of basic blocks to straight-line code.
$\llbracket \cdot \rrbracket_{\text{pa}}$	generate PA-RISC code.
$\llbracket \cdot \rrbracket_{\text{sched.}}$	schedule the PA-RISC code.

Because we have chosen to focus on register allocation, the bulk of the back end is the function *rdfs* that processes the call graph and performs the register allocation and code generation. Some of the phases are unimportant technicalities ($\llbracket \cdot \rrbracket_{\text{sib}}$ and $\llbracket \cdot \rrbracket_{\text{cr}}$), and some are trivial ($\llbracket \cdot \rrbracket_{\text{sib}}$, $\llbracket \cdot \rrbracket_{\text{cr}}$, $\llbracket \cdot \rrbracket_{\text{cg}}$, $\llbracket \cdot \rrbracket_{\text{bbs}}$ and $\llbracket \cdot \rrbracket_{\text{pa}}$), but, for completeness and such that you can see this yourself, we describe them briefly in the following.

The rest of this chapter extends the material in chapter 5 and explains in detail the inter-procedural part of the algorithm, i.e., the phases from E to \mathbf{K} . The next chapter carries on where chapter 6 ended by developing the

per-function part of the translation, i.e., the internals of *rdfs*. The chapter after that then explains the phases from \mathbf{K} to \mathfrak{B} .

Casting a phase of the compilation as a translation from one language into another, annotated language gives a way to describe precisely and explicitly the results of the phase. In some situations, however, the annotations can clutter the picture with unnecessary detail. Therefore, we shall omit annotations that are not of interest in the given situation. For instance, a completely annotated application has the intimidating appearance ${}_{\omega} \check{e}_0 \overset{r}{\lambda} \vec{e}_1 {}_{\omega'}$. If we are only interested in the set λ of λ 's that may be applied, we present the application as $\check{e}_0 \overset{r}{\lambda} \vec{e}_1$, with the understanding that the other annotations are also there. In the same spirit, you may want to ignore the difference between the different kinds of expressions (\dot{e} , \hat{e} , \check{e} , etc.); they all denote expressions of the same underlying structure and only differ in how many kinds of annotations they carry.

7.2 Closure analysis

The set Λ of functions is defined

$$\Lambda ::= \lambda Y. E \text{ at } P \mid F \vec{P} Y = E.$$

The closure analysis translates a program e to a *lambda-annotated* program $\hat{e} \in \hat{E}$ that is defined by the same grammar as E , except that all applications have been annotated with a set λ of λ 's that can be applied at that application:

$$\hat{E} ::= \hat{E}_{\Lambda} \hat{E} \mid F_{\Lambda} \vec{P} \hat{E} \mid \dots$$

where $\Lambda = \mathcal{P}\Lambda$, and the rest of the grammar is like that for E .

At applications of the form $f \vec{\rho} e_2$, the function named f will be applied. At applications of the form $e_1 e_2$, it is not generally decidable at compile-time which functions may be applied, as we can see from the following example:

```
( $\lambda k.k \ x$  at r17)
  (if  $e_0$  then  $\lambda y.y+y$  at r15 else  $\lambda z.4+z$  at r15).
```

The *if*-expression may evaluate to $\lambda y.y+y$ at r15 or $\lambda z.4+z$ at r15, and consequently, either of those λ 's may be applied at the application $k \ x$. We will have to settle with an analysis that gives an approximation of the set of λ 's that may be applied. The analysis must be safe: it must find *at least* each possible λ that may be applied. The analysis we present here is based on the region annotations in the program and has been developed by Mads Tofte.

Using the region annotations for closure analysis

Every expression has a result region where the value the expression evaluates to is put. For instance, the result region of the expression $\lambda y.y+y$ at r15 is $\rho = \mathbf{r15}$. Hence, the result region of the expression k (the second k of $(\lambda k.k$

\mathbf{x} at $\mathbf{r17}$)) must also be $\mathbf{r15}$, because $(\lambda \mathbf{k}. \mathbf{k} \ \mathbf{x} \ \mathbf{at} \ \mathbf{r17})$ is applied to the *if*-expression whose result region is $\mathbf{r15}$. The result region of an expression is only explicitly stated as an “*at* ρ ”-annotation when the expression *builds* a new value in ρ , for that is the only case in which it is actually necessary to know the result region to evaluate the expression; to evaluate an expression that does not build a new value, it is not necessary to know the result region.

It requires a data flow analysis to find the result region of a given expression (as was suggested with the expression \mathbf{k} above). Luckily, the region inference, as a by-product, infers the result region for every sub-expression of the program, and our source language actually has a result region annotation “: ρ ” on each sub-expression. So far we have ignored the “: ρ ”-annotations to simplify the exposition. Our source language is really

$$\begin{aligned} E & ::= \bar{E} : P \\ \bar{E} & ::= E E \\ & \quad | F \vec{P} E \\ & \quad | \mathbf{let} \ X = E \ \mathbf{in} \ E \\ & \quad \vdots \end{aligned}$$

where ρ of $\bar{e} : \rho$ is the region in which the value \bar{e} evaluates to is put. At put points, e.g., $\lambda y. e_0 \ \mathbf{at} \ \rho : \rho'$, the result region ρ' and ρ are, of course, the same region variable.

When devising a data flow analysis, one can think of the regions as channels between expressions in which values flow. To analyse what closures flow to e_1 of an application $e_1 \ e_2$, one must look at all expressions that build a closure and record in what region the closure is put. Then the closures that flow to an expression e_1 can be found as the closures that are put into the result region of e_1 . The closure analysis thus has two phases:

1. *Record in which regions the λ 's are put.* For each sub-expression of the program of this kind

$$\lambda y. e_0 \ \mathbf{at} \ \rho,$$

record that $\lambda y. e_0 \ \mathbf{at} \ \rho$ flows into the region ρ , i.e., for each ρ , maintain a set, $\alpha\rho$, of λ 's that flow into that ρ , and record that $(\lambda y. e_0 \ \mathbf{at} \ \rho) \in \alpha\rho$.

Since *letrec*-functions are named, we do not need regions to find where they may be applied: For each sub-expression

$$\mathbf{letrec} \ f_1 \vec{\rho}_1 y_1 = e_1 \ \cdots \ f_m \vec{\rho}_m y_m = e_m \ \mathbf{at} \ \rho \ \mathbf{in} \ e_{m+1},$$

record that the λ that may be applied at a region polymorphic application of f_i is $f_i \vec{\rho}_i y_i = e_i$.

2. *Annotate applications.* At each application

$$e_1 \ e_2,$$

where e_1 has the form $\bar{e}_1 : \rho_1$, the set λ of λ 's that may be applied is the set $\alpha\rho_1$ of λ 's that flow into ρ_1 .

At a region polymorphic function application

$$f \vec{\rho} e_2,$$

only one specific λ can be applied, viz. the one named f .

Because of *region polymorphism* this algorithm is not quite sufficient. Below we modify the algorithm to deal with this.

Dealing with region polymorphism

Because *formal* region variables can be instantiated to *actual* region variables there is an aliasing problem. In

```
letrec f [r1] y = ... f [r2] ...
      in ... f [r3] ... ,
```

the formal region variable $\mathbf{r1}$ is instantiated to the actual region variables $\mathbf{r2}$ and $\mathbf{r3}$, so occurrences of $\mathbf{r1}$ might actually stand for $\mathbf{r2}$ or $\mathbf{r3}$. Whenever we do something with a region variable, we must also do it with all the region variables it might be aliased with. This complication influences both phases:

1. When we record that a λ flows into a formal region variable ρ , we must also record that it flows into all actual region variables to which ρ may be instantiated. If the body of \mathbf{f} above contains $\lambda_y = \lambda \mathbf{y}.\mathbf{y}$ **at** $\mathbf{r1}$, we must record not only that $\lambda_y \in \alpha(\mathbf{r1})$, but also that $\lambda_y \in \alpha(\mathbf{r2})$ and $\lambda_y \in \alpha(\mathbf{r3})$.

2. Similarly, at an application, where the λ to be applied flows out of some formal region variable ρ , the λ 's that may be applied are those that flow out of all region variables to which ρ may be instantiated. E.g., in the body of \mathbf{f} above, at an application $e_1 e_2$ where e_1 has the form $\bar{e}_1 : \mathbf{r1}$, the set of λ 's that may be applied is $\alpha(\mathbf{r1}) \cup \alpha(\mathbf{r2}) \cup \alpha(\mathbf{r3})$, and not only $\alpha(\mathbf{r1})$.

This is all there is to the closure analysis; if you understand it, you can skip the rest of this section.

The algorithm in detail

The closure analysis now consists of three phases:

$$\begin{aligned} \llbracket e \rrbracket_{\text{ca}} &= \text{let } \varphi = \llbracket e \rrbracket_{\text{alias}} \\ &\quad \alpha = \llbracket e \rrbracket_{\text{flow}} \varphi \\ &\quad \hat{e} = \llbracket e \rrbracket_{\text{annotate}} \varphi \alpha \\ &\text{in } \hat{e}. \end{aligned}$$

The first phase, $\llbracket \cdot \rrbracket_{\text{alias}}$, collects the aliasing information φ , which is used in the following two phases. The next phase, $\llbracket \cdot \rrbracket_{\text{flow}}$, records for each ρ the set $\alpha\rho$ of λ 's that flow into that ρ . This information is used in the last phase, $\llbracket \cdot \rrbracket_{\text{annotate}}$, that annotates each application in the program with the set of λ 's that may be applied at that application.

0. *Build region flow graph*, $\llbracket \cdot \rrbracket_{\text{alias}}$. The region variable aliasing information is collected by traversing the program and looking at which region variables the region polymorphic functions are applied to. This gives a *region flow graph* $\varphi \in \mathcal{P}(\mathbf{P} \times \mathbf{P})$, in which there is an edge between a formal region variable $\dot{\rho}$ and a region variable ρ , i.e., $\dot{\rho}\varphi\rho$, iff $\dot{\rho}$ may be instantiated to ρ . E.g., if φ were the region flow graph for the example above, we would have $\mathbf{r1}\varphi\mathbf{r2}$ and $\mathbf{r1}\varphi\mathbf{r3}$.

Denote by φ^* the reflexive, transitive closure of φ . A ρ is *aliased with* ρ' iff $\rho'\varphi^*\rho$. In the example above, the set of region variables aliased with $\mathbf{r1}$, i.e., the set $\{\rho \mid \mathbf{r1}\varphi^*\rho\}$, is $\{\mathbf{r1}, \mathbf{r2}, \mathbf{r3}\}$. The region variables aliased with $\mathbf{r2}$, i.e. $\{\rho \mid \mathbf{r2}\varphi^*\rho\}$, is $\{\mathbf{r2}\}$.

While constructing the region flow graph φ with $\llbracket \cdot \rrbracket_{\text{alias-0}}$, we keep track of which formal region variables each region polymorphic function has, in an environment $\vartheta \in F \xrightarrow{\perp} \vec{\mathbf{P}}$ which maps names to tuples of formal region variables. Initially, ϑ is \emptyset :

$$\llbracket e \rrbracket_{\text{alias}} = \llbracket e \rrbracket_{\text{alias-0}} \vartheta.$$

At a **letrec**-expression, the region polymorphic functions are added to the environment:

$$\begin{aligned} \llbracket \text{letrec } f_1 \vec{\rho}_1 y_1 = e_1 \cdots f_m \vec{\rho}_m y_m = e_m \text{ at } \rho \text{ in } e_{m+1} \rrbracket_{\text{alias-0}} \vartheta &= \\ \text{let } \vartheta = \vartheta + \{f_1 \mapsto \vec{\rho}_1, \dots, f_m \mapsto \vec{\rho}_m\} & \\ \text{in } \llbracket e_1 \rrbracket_{\text{alias-0}} \vartheta \cup \cdots \cup \llbracket e_{m+1} \rrbracket_{\text{alias-0}} \vartheta. & \end{aligned}$$

At a region polymorphic application of f we confer with the environment to see what the formal region variables of f are, and add edges from each formal region variable to the corresponding actual region variable:

$$\begin{aligned} \llbracket f [\rho_1, \dots, \rho_k] e_2 \rrbracket_{\text{alias-0}} \vartheta &= \\ \text{let } [\dot{\rho}_1, \dots, \dot{\rho}_k] = \vartheta f & \\ \text{in } \{(\dot{\rho}_1, \rho_1), \dots, (\dot{\rho}_k, \rho_k)\} \cup \llbracket e_2 \rrbracket_{\text{alias-0}} \vartheta. & \end{aligned}$$

Other constructs are simply traversed, e.g.:

$$\begin{aligned} \llbracket e_1 e_2 \rrbracket_{\text{alias-0}} \vartheta &= \llbracket e_1 \rrbracket_{\text{alias-0}} \vartheta \cup \llbracket e_2 \rrbracket_{\text{alias-0}} \vartheta \\ \llbracket e \text{ at } \rho \rrbracket_{\text{alias-0}} \vartheta &= \emptyset. \end{aligned}$$

1. *Record in which regions the λ 's are put*, $\llbracket \cdot \rrbracket_{\text{flow}}$. Remember $\alpha\rho$ is the set of λ 's that flow into ρ . Use $+_{\text{flow}}$ to merge α 's, i.e., if $\alpha', \alpha'' \in \mathbf{P} \rightarrow \mathcal{P}\Lambda$,

$$\alpha' +_{\text{flow}} \alpha'' = \lambda\rho. \alpha'\rho \cup \alpha''\rho.$$

If $\alpha \in \mathbf{P} \xrightarrow{\perp} \mathcal{P}\Lambda$ is a partial map, denote by $\text{ext } \alpha$ the extension of α from $\text{Dm } \alpha$ to the whole of \mathbf{P} given by:

$$\text{ext } \alpha = (\lambda\rho. \emptyset) + \alpha.$$

The only construct of interest in this phase is $\lambda y. e_0 \text{ at } \rho$: we must record that $\lambda y. e_0 \text{ at } \rho$ flows into all region variables ρ' aliased with ρ :

$$\begin{aligned} \llbracket \lambda y. e_0 \text{ at } \rho \rrbracket_{\text{flow}} \varphi &= \text{ext}\{ \rho' \mapsto \{ \lambda y. e_0 \text{ at } \rho \} \mid \rho \varphi^* \rho' \} \\ &\quad +_{\text{flow}} \llbracket e_0 \rrbracket_{\text{flow}} \varphi. \end{aligned}$$

Other constructs are simply traversed; e.g.:

$$\begin{aligned} \llbracket \text{letrec } f_1 \vec{\rho}_1 y_1 = e_1 \cdots f_m \vec{\rho}_m y_m = e_m \text{ at } \rho \text{ in } e_{m+1} \rrbracket_{\text{flow}} \varphi &= \\ = \llbracket e_1 \rrbracket_{\text{flow}} \varphi +_{\text{flow}} \cdots +_{\text{flow}} \llbracket e_{m+1} \rrbracket_{\text{flow}} \varphi \end{aligned}$$

$$\llbracket e_1 e_2 \rrbracket_{\text{flow}} \varphi = \llbracket e_1 \rrbracket_{\text{flow}} \varphi +_{\text{flow}} \llbracket e_2 \rrbracket_{\text{flow}} \varphi$$

$$\llbracket f \vec{\rho} e_2 \rrbracket_{\text{flow}} \varphi = \llbracket e_2 \rrbracket_{\text{flow}} \varphi$$

$$\llbracket \hat{e} \text{ at } \rho \rrbracket_{\text{flow}} \varphi = \lambda \rho. \emptyset.$$

2. *Annotate applications*, $\llbracket \cdot \rrbracket_{\text{annotate}}$. An environment $\varpi \in F \xrightarrow{\perp} B$ is used for annotating region polymorphic applications. Initially, ϖ is \emptyset :

$$\llbracket e \rrbracket_{\text{annotate}} \varphi \alpha = \llbracket e \rrbracket_{\text{annotate-0}} \varphi \alpha \emptyset.$$

At an application $e_1 e_2$, the set λ of λ 's that may be applied is the union of the sets of λ 's that flow into the regions that are aliased with the result region ρ_1 of e_1 . I.e., if e_1 has the form $\bar{e}_1 : \rho_1$,

$$\begin{aligned} \llbracket e_1 e_2 \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi &= \text{let } \lambda = \bigcup \{ \alpha \rho' \mid \rho_1 \varphi^* \rho' \} \\ &\quad \hat{e}_1 = \llbracket e_1 \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi \\ &\quad \hat{e}_2 = \llbracket e_2 \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi \\ &\quad \text{in } \hat{e}_1 \lambda \hat{e}_2. \end{aligned}$$

At a region polymorphic application, we assume the one λ that may be applied is recorded in the environment ϖ :

$$\begin{aligned} \llbracket f \vec{\rho} e_2 \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi &= \text{let } \lambda = \varpi f \\ &\quad \hat{e}_2 = \llbracket e_2 \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi \\ &\quad \text{in } f \lambda \vec{\rho} \hat{e}_2. \end{aligned}$$

At a **letrec**-expression, we must record in ϖ what λ each f is bound to. If b_i is $f_i \vec{\rho}_i y_i = e_i$,

$$\begin{aligned} \llbracket \text{letrec } b_1 \cdots b_m \text{ at } \rho \text{ in } e_{m+1} \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi &= \\ \text{let } \varpi = \varpi + \{ f_1 \mapsto b_1, \dots, f_m \mapsto b_m \} & \\ \hat{e}_1 = \llbracket e_1 \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi & \\ \vdots & \\ \hat{e}_{m+1} = \llbracket e_{m+1} \rrbracket_{\text{annotate-0}} \varphi \alpha \varpi & \\ \text{in } \text{letrec } f_1 \vec{\rho}_1 y_1 = \hat{e}_1 \cdots f_m \vec{\rho}_m y_m = \hat{e}_m \text{ at } \rho \text{ in } \hat{e}_{m+1}. & \end{aligned}$$

The sets $\{\alpha\rho' \mid \rho\varphi^*\rho'\}$ etc. are computed by finding the set of ρ' 's that are reachable from ρ in φ using a standard algorithm.

Comparison with other closure analyses and discussion

The closure analysis presented here was invented by Mads Tofte.

Closure analysis algorithms for higher-order functional programming languages (Scheme) have been developed in (Sestoft, 1992) and (Shivers, 1988). Sacrificing what may be insignificant accuracy, a closure analysis can be devised with a better worst-case time complexity (Henglein, 1992).

Aiken et al. (1995) present a closure analysis which is also based on region annotations. They generate a set of constraints using the region annotations and then solve these constraints.

The closure analysis described here (and closure analyses in general) only gives a safe approximation of which functions may be applied at each application, because it works on the whole program. In other words, it will not give a safe approximation with separate compilation (compiling parts of the program separately) or incremental compilation (compiling declarations separately as the user types them in).

With incremental compilation, the user can type in the SML declaration

```
fun apply f x = f x;
```

and then later use **apply**. The compiler cannot know the applications from whence **apply** may be applied, and it cannot know the set of functions that may be applied at the application **f x**.

The solution is to use fixed linking conventions for the function **apply** and for the application **f x**, as (Chow, 1988) does. This means an analysis must find out which functions may be applied from outside, and at which applications functions from outside may be applied. By making two versions of functions, one that uses the standard linking convention and one that uses a specialised one, the penalty of separate compilation can be reduced.

If one wants to avoid using this closure analysis, because of the complexity it adds to the compilation or because of the trouble with separate compilation or because one does not have a region annotated program, it should be possible to replace the closure analysis with a simpler one: Annotate all applications of the form $f \vec{\rho}e_2$ with the function named f , and all applications of the form $e_1 e_2$ with “unknown”, forcing them all to use the same linking convention. (To deal with separate compilation, this simpler closure analysis will still have to be augmented with an analysis that determines which named functions may be applied from the outside.)

7.3 Sibling analysis

Recall how **letrec**-functions are treated (section 4.7): A shared closure is built for the λ 's in the **letrec**-expression:

$$\begin{aligned} \text{letrec } f_1 \vec{\rho}_1 y_1 &= \hat{e}_1 \\ &\vdots \\ f_m \vec{\rho}_m y_m &= \hat{e}_m \text{ at } \rho \text{ in } \hat{e}_{m+1} \end{aligned}$$

Because of this, all occurrences of one of the siblings f_1, \dots, f_m in $\hat{e}_1, \dots, \hat{e}_{m+1}$ are really uses of the same value, the shared closure. Therefore, we do not want to distinguish between f_1, \dots, f_m , and the phase described here replaces each occurrence of any of f_1, \dots, f_m with a new identifier, \mathbf{f} , which we will call a *sibling name*. As this identifier, we simply use the set of siblings, i.e., $\mathbf{f} = \{f_1, \dots, f_m\}$, which is unique.

Formally, a program-wide analysis, $\llbracket \cdot \rrbracket_{\text{sib}}$, translates a lambda-annotated program $\hat{e} \in \hat{E}$ to a *sibling-annotated program* $\hat{e} \in \hat{E}^\diamond$, defined by

$$\hat{E}^\diamond ::= \mathbf{F} \ \Lambda \ \vec{\mathbf{P}} \ \hat{E} \mid \dots$$

where \mathbf{F} is the set of sibling names, i.e., $\mathcal{P}F$. The rest of the grammar is similar to that for \hat{E} . After the sibling analysis, the set Z of variables (p. 19) is

$$\hat{Z}^\diamond ::= X \mid Y \mid \mathbf{F} \mid \mathbf{P} \mid A,$$

i.e., \mathbf{F} replaces F .

The analysis is trivial. It keeps track of sibling names in an environment $\mathcal{F} \in F \xrightarrow{\perp} \mathbf{F}$ that maps a name f to its sibling name \mathbf{f} . Initially, \mathcal{F} is \emptyset :

$$\llbracket \hat{e} \rrbracket_{\text{sib}} = \llbracket \hat{e} \rrbracket_{\text{sib-0}} \emptyset$$

$$\begin{aligned} \llbracket f \ \lambda \ \vec{\rho} \ \hat{e}_2 \rrbracket_{\text{sib-0}} \mathcal{F} &= \text{let } \hat{e}_2 = \llbracket \hat{e}_2 \rrbracket_{\text{sib-0}} \mathcal{F} \\ &\quad \mathbf{f} = \mathcal{F} f \\ &\text{in } \mathbf{f} \ \lambda \ \vec{\rho} \ \hat{e}_2 \end{aligned}$$

$$\begin{aligned}
& \left[\left[\begin{array}{l} \text{letrec } f_1 \vec{\rho}_1 y_1 = \hat{e}_1 \\ \vdots \\ f_m \vec{\rho}_m y_m = \hat{e}_m \text{ at } \rho \text{ in } \hat{e}_{m+1} \end{array} \right] \right]_{\text{sib-0}} \mathcal{F} = \\
& = \text{let } \mathbf{f} = \{f_1, \dots, f_m\} \\
& \quad \mathcal{F} = \mathcal{F} + \{f_1 \mapsto \mathbf{f}, \dots, f_m \mapsto \mathbf{f}\} \\
& \quad \hat{e}_1 = \llbracket \hat{e}_1 \rrbracket_{\text{sib-0}} \mathcal{F} \\
& \quad \vdots \\
& \quad \hat{e}_{m+1} = \llbracket \hat{e}_{m+1} \rrbracket_{\text{sib-0}} \mathcal{F} \\
& \text{in letrec } f_1 \vec{\rho}_1 y_1 = \hat{e}_1 \\
& \quad \vdots \\
& \quad f_m \vec{\rho}_m y_m = \hat{e}_m \text{ at } \rho \text{ in } \hat{e}_{m+1}.
\end{aligned}$$

Other constructs are simply traversed, e.g.:

$$\begin{aligned}
\llbracket \lambda y. \hat{e}_0 \text{ at } \rho \rrbracket_{\text{sib-0}} \mathcal{F} &= \text{let } \hat{e}_0 = \llbracket \hat{e}_0 \rrbracket_{\text{sib-0}} \mathcal{F} \text{ in } \lambda y. \hat{e}_0 \text{ at } \rho, \\
\llbracket \hat{e}_1 \lambda \hat{e}_2 \rrbracket_{\text{sib-0}} \mathcal{F} &= \text{let } \hat{e}_1 = \llbracket \hat{e}_1 \rrbracket_{\text{sib-0}} \mathcal{F} \\
& \quad \hat{e}_2 = \llbracket \hat{e}_2 \rrbracket_{\text{sib-0}} \mathcal{F} \\
& \text{in } \hat{e}_1 \lambda \hat{e}_2.
\end{aligned}$$

7.4 Closure representation analysis

The title of this section is a bit pretentious since the analysis does nothing more than find the free variables of each λ and assign a closure offset to each free variable.

The closure offset of a free variable of λ can be an arbitrary non-negative number as long as it uniquely identifies the free variable among the other free variables of that λ . We cannot just decide this offset ad hoc when necessary, though, for the code that builds the closure for a function and the code for the function must agree on the closure offset of each free variable. Therefore a program-wide analysis annotates each λ with its closure representation.

The closure representation of the function $\lambda y. \hat{e}_0 \text{ at } \rho$ is a map

$$\mathcal{K} \in \mathcal{K} = Z \xrightarrow{\perp} I$$

that maps the free variables of $\lambda y. \hat{e}_0 \text{ at } \rho$ to their offsets in the closure. The closure representation of the functions b_1, \dots, b_m in

$$\text{letrec } b_1 \cdots b_m \text{ at } \rho \text{ in } \hat{e}_{m+1}$$

is a map from the combined free variables of the functions to their offsets in the shared closure.

variable within the bodies $\hat{e}_1, \dots, \hat{e}_m$ of the λ 's; from the point of view of these λ 's, \mathbf{f} is the closure parameter.

Other constructs are simply traversed, e.g.:

$$\begin{aligned} \llbracket \text{let } x = \hat{e}_1 \text{ in } \hat{e}_2 \rrbracket_{\text{cr}} &= \text{let } \hat{e}_1 = \llbracket \hat{e}_1 \rrbracket_{\text{cr}} \\ &\quad \hat{e}_2 = \llbracket \hat{e}_2 \rrbracket_{\text{cr}} \\ &\quad \text{in let } x = \hat{e}_1 \text{ in } \hat{e}_2, \end{aligned}$$

$$\llbracket x \rrbracket_{\text{cr}} = x.$$

7.5 Converting functions to functions of several arguments

When can a function be converted to a function of several arguments? Consider **sumacc**:

```

fun sumacc (0,n) = n
  | sumacc (m,n) = sumacc (m-1, m+n).

```

Converted to our source language, it will look something like:

```

letrec sumacc y = if #0 y = 0 then #1 y
                  else letregion r4:2 in
                      sumacc (#0 y - 1, #0 y + #1 y) at r4:2
at r2:0 in e.

```

The fact that only the components of the tuple are used is reflected in that the argument, y , only appears in expressions of the form $\#i y$.

A very simple, sufficient condition for when a λ may be converted to a function of several arguments is:

1. all occurrences of the argument, y , of λ must be in the context $\#i y$, and it must be directly within λ ; and
2. at each application $\hat{e}_1 \lambda \hat{e}_2$ or $\mathbf{f} \lambda \vec{\rho} \hat{e}_2$ where λ may be applied (i.e., where $\lambda \in \boldsymbol{\lambda}$),
 - (a) e_2 must have the form $(\hat{e}_1, \dots, \hat{e}_n)$ **at** ρ
 - (b) every $\lambda' \in \boldsymbol{\lambda}$ must be a function of several arguments.

The first condition will not allow converting λ to a function of several arguments if y occurs in a λ' within λ . If y appears free in any λ' within λ , λ contains an implicit use of y , for y is used when the closure is built for λ' . This implicit use is not of the form $\#i y$, and that is the reason we disallow converting λ in that case.

The function **sumacc** will be converted to a function of several arguments because it is applied to a tuple expression. If **sumacc** had instead been

```

fun sumacc (0,n) = n
  | sumacc (m,n) = let val x = (m-1, m+n)
                    in
                      sumacc x
                    end

```

it could not have been converted to a function of several arguments, because it is applied to a variable.

We will implement the conversion of functions to functions of several arguments by converting the closure-representation-annotated program $\dot{e} \in \dot{E}$, to a *several-argumented program* $\dot{e} \in \dot{E}$ where the arguments of a function with several arguments have been made explicit. For example, `sumacc` above may be transformed to:

```

letrec sumacc <y0, y1> =
  if y0 = 0 then y1
    else letregion r4:2 in
      sumacc <y0-1, y0+y1> ,

```

where `y0` and `y1` are fresh variables. Note that the memory allocated in `r4:2` for the tuple is never used. This unnecessary allocation and deallocation of memory is not costly (it is done in two K instructions), but indicates that the conversion to several argument functions should be done before the region inference.

The grammar for \dot{E} is:

$$\begin{aligned}
 \vec{E} &::= \langle \dot{E}, \dots, \dot{E} \rangle \\
 \vec{Y} &::= \langle Y, \dots, Y \rangle \\
 \dot{E} &::= \dot{E} \Lambda \vec{E} \\
 &| \mathbf{F} \Lambda \vec{P} \vec{E} \\
 &| \lambda \vec{Y}. \mathcal{K} \dot{E} \text{ at } P \\
 &\vdots \\
 \dot{B} &::= \mathbf{F} \dot{P} \vec{Y} = \mathcal{K} \dot{E}.
 \end{aligned}$$

The rest of the grammar is similar to the grammar for \dot{E} .

The translation $\llbracket \cdot \rrbracket_{\text{sa}} \in \dot{E} \rightarrow \dot{E}$ consists of two phases:

$$\llbracket \dot{e} \rrbracket_{\text{sa}} = \text{let } \mathcal{Y} = \llbracket \dot{e} \rrbracket_{\text{which}} \text{ in } \llbracket \dot{e} \rrbracket_{\text{convert}} \mathcal{Y} - \langle \rangle.$$

First a simple analysis, $\llbracket \cdot \rrbracket_{\text{which}}$, traverses \dot{e} and records which λ 's may be converted to functions of several arguments according to the rules above. The result is a map $\mathcal{Y} \in \Lambda \rightarrow \vec{Y}$, where $\mathcal{Y} \lambda = \langle y_1, \dots, y_n \rangle$ with $n \geq 1$ means that λ is a function of n arguments, and y_1, \dots, y_n are the fresh

variables that are to be inserted. When $\mathcal{Y}\lambda = \langle \rangle$, λ cannot be converted to a function of several arguments. Since $\llbracket \cdot \rrbracket_{\text{which}}$ is straightforward, we will not describe it further.

Second, $\llbracket \cdot \rrbracket_{\text{convert}}$ traverses the program replacing $\#i y$ with the i^{th} argument y_i , and it converts arguments of functions to their new form. To replace $\#i y$ with y_i we need to know the argument y of the current function and the fresh variables $\vec{y} = \langle y_1, \dots, y_n \rangle$ it is replaced with. Therefore $\llbracket \cdot \rrbracket_{\text{convert}}$ has both \mathcal{Y} , and y and \vec{y} as arguments. If the current function cannot be converted, \vec{y} is $\langle \rangle$.

An application must be converted to pass several arguments, if one (and thus all) of the functions that may be applied takes several arguments:

$$\begin{aligned} \llbracket \dot{e}_0_{\{\lambda_1, \dots, \lambda_m\}}(\dot{e}_1, \dots, \dot{e}_n) \text{ at } \rho \rrbracket_{\text{convert}} \mathcal{Y} y \vec{y} &= \\ \text{let } \dot{e}_0 &= \llbracket \dot{e}_0 \rrbracket_{\text{convert}} \mathcal{Y} y \vec{y} \\ &\vdots \\ \dot{e}_n &= \llbracket \dot{e}_n \rrbracket_{\text{convert}} \mathcal{Y} y \vec{y} \\ \text{in if } \mathcal{Y} \lambda_1 = \langle \rangle &\text{ then } \dot{e}_0_{\{\lambda_1, \dots, \lambda_m\}} \langle \dot{e}_1, \dots, \dot{e}_n \rangle \text{ at } \rho > \\ &\text{else } \dot{e}_0_{\{\lambda_1, \dots, \lambda_m\}} \langle \dot{e}_1, \dots, \dot{e}_n \rangle. \end{aligned}$$

Notice that also a function that takes a 1-tuple may be converted to a function of several arguments. If such a function is converted, the argument will have the form $\langle \dot{e}_1 \rangle$; if not, the argument will have the form $\langle \dot{e}_1 \rangle \text{ at } \rho$.

If the argument \dot{e}_1 is not a tuple-expression, only one argument will be passed:

$$\begin{aligned} \llbracket \dot{e}_0 \lambda \dot{e}_1 \rrbracket_{\text{convert}} \mathcal{Y} y \vec{y} &= \text{let } \dot{e}_0 = \llbracket \dot{e}_0 \rrbracket_{\text{convert}} \mathcal{Y} y \vec{y} \\ &\quad \dot{e}_1 = \llbracket \dot{e}_1 \rrbracket_{\text{convert}} \mathcal{Y} y \vec{y} \\ \text{in } \dot{e}_0 \lambda \langle \dot{e}_1 \rangle. \end{aligned}$$

Region polymorphic function applications are treated similarly.

When a function definition is encountered, the argument is converted if possible:

$$\begin{aligned} \llbracket \lambda y. \dot{e}_0 \text{ at } \rho \rrbracket_{\text{convert}} \mathcal{Y} y' \vec{y}' &= \text{let } \dot{e}_0 = \llbracket \dot{e}_0 \rrbracket_{\text{convert}} \mathcal{Y} y(\mathcal{Y}(\lambda y. \dot{e}_0 \text{ at } \rho)) \\ \text{in if } \mathcal{Y}(\lambda y. \dot{e}_0 \text{ at } \rho) &= \langle y_1, \dots, y_n \rangle \\ \text{then } \lambda \langle y_1, \dots, y_n \rangle. &\dot{e}_0 \text{ at } \rho \\ \text{else } \lambda \langle y \rangle. \dot{e}_0 &\text{ at } \rho. \end{aligned}$$

Region polymorphic functions are treated in the same manner.

We replace $\#i y$ with y_i if possible:

$$\llbracket \#i z \rrbracket_{\text{convert}} \mathcal{Y} y \langle y_1, \dots, y_n \rangle = \text{if } z = y \text{ then } y_i \text{ else } \#i z.$$

If the current function does not take several arguments, we do not change $\#i y$:

$$\llbracket \#i z \rrbracket_{\text{convert}} \mathcal{Y} y \langle \rangle = \#i z.$$

All other constructs are just traversed.

7.6 Building the call graph

Since our closure analysis annotates every application in the program with the set, λ , of λ 's that might be applied at that application, we can build the call graph, γ , easily: for each application annotated with λ , add edges from the λ the application is directly within to every λ' in λ . The function, $\llbracket \cdot \rrbracket_{\text{cg}}$, that computes the call graph from a lambda-annotated program λ_{main} is defined via an auxiliary function, $\llbracket \cdot \rrbracket_{\text{cg-0}}$, which takes an extra argument, λ_{cur} , the λ we are currently directly within. The auxiliary function returns the unrooted call graph:

$$\begin{aligned} \llbracket \cdot \rrbracket_{\text{cg}} &\in \dot{E} \rightarrow ? \\ \llbracket \cdot \rrbracket_{\text{cg-0}} &\in \dot{E} \rightarrow \Lambda \rightarrow \mathcal{P}\Lambda \times \mathcal{P}(\Lambda \times \Lambda) \\ \gamma = (\lambda^{\text{cg}}, \mathcal{E}, \lambda_{\text{main}}) &\in ? = \mathcal{P}\Lambda \times \mathcal{P}(\Lambda \times \Lambda) \times \Lambda \end{aligned}$$

$$\Lambda ::= \lambda \vec{Y}. \kappa \dot{E} \text{ at } P \quad | \quad \mathbf{F} \vec{P} \vec{Y} = \kappa \dot{E}.$$

Initially, $\llbracket \dot{e} \rrbracket_{\text{cg}}$ turns the program \dot{e} into a function λ_{main} , which becomes the initial λ_{cur} :

$$\begin{aligned} \llbracket \dot{e} \rrbracket_{\text{cg}} &= \text{let } \lambda_{\text{main}} = \lambda \mathbf{y}_{\text{main}}. \emptyset \dot{e} \text{ at } \mathbf{r}_{\text{main}} \\ &\quad (\lambda^{\text{cg}}, \mathcal{E}) = \llbracket \lambda_{\text{main}} \rrbracket_{\text{cg-0}} \lambda_{\text{main}} \\ &\quad \text{in } (\lambda^{\text{cg}}, \mathcal{E}, \lambda_{\text{main}}), \end{aligned}$$

where \mathbf{y}_{main} and \mathbf{r}_{main} do not occur in \dot{e} , and \emptyset is a dummy closure representation.

Let $\cup_{(\cdot)}$ denote union of graphs:

$$(\lambda^{\text{cg}}, \mathcal{E}) \cup_{(\cdot)} (\lambda^{\text{cg}'}, \mathcal{E}') = (\lambda^{\text{cg}} \cup \lambda^{\text{cg}'}, \mathcal{E} \cup \mathcal{E}').$$

At λ -abstractions, $\llbracket \cdot \rrbracket_{\text{cg-0}}$ records the λ -abstraction in λ^{cg} , and λ_{cur} is changed:

$$\begin{aligned} \llbracket \lambda \vec{y}. \kappa \dot{e}_0 \text{ at } \rho \rrbracket_{\text{cg-0}} \lambda_{\text{cur}} &= (\{\lambda \vec{y}. \kappa \dot{e}_0 \text{ at } \rho\}, \emptyset) \\ &\quad \cup_{(\cdot)} \llbracket \dot{e}_0 \rrbracket_{\text{cg-0}} (\lambda \vec{y}. \kappa \dot{e}_0 \text{ at } \rho). \end{aligned}$$

At a **letrec**-expression, the region polymorphic functions b_i of the form $f_i \vec{\rho}_i \vec{y}_i = \kappa \dot{e}_i$ are recorded in λ^{cg} , and the bodies, \dot{e}_i , are traversed, each with the proper λ_{cur} :

$$\begin{aligned} \llbracket \text{letrec } \kappa b_1 \cdots b_m \text{ at } \rho \text{ in } \dot{e}_{m+1} \rrbracket_{\text{cg-0}} \lambda_{\text{cur}} &= \\ (\{b_1, \dots, b_m\}, \emptyset) \cup_{(\cdot)} \llbracket \dot{e}_1 \rrbracket_{\text{cg-0}} b_1 \cup_{(\cdot)} \cdots \cup_{(\cdot)} \llbracket \dot{e}_m \rrbracket_{\text{cg-0}} b_m \\ &\quad \cup_{(\cdot)} \llbracket \dot{e}_{m+1} \rrbracket_{\text{cg-0}} \lambda_{\text{cur}}. \end{aligned}$$

At applications, edges are added from λ_{cur} to all λ 's that can be applied:

$$\begin{aligned} \llbracket \dot{e}_0 \lambda \langle \dot{e}_1, \dots, \dot{e}_n \rangle \rrbracket_{\text{cg-0}} \lambda_{\text{cur.}} &= \\ (\emptyset, \{(\lambda_{\text{cur.}}, \lambda) \mid \lambda \in \boldsymbol{\lambda}\}) & \\ \cup_{(\cdot)} \llbracket \dot{e}_0 \rrbracket_{\text{cg-0}} \lambda_{\text{cur.}} \cup_{(\cdot)} \dots \cup_{(\cdot)} \llbracket \dot{e}_n \rrbracket_{\text{cg-0}} \lambda_{\text{cur.}}, & \end{aligned}$$

Similarly with region polymorphic applications.

Other expressions are just traversed, e.g.:

$$\llbracket \dot{e}_1 o \dot{e}_2 \rrbracket_{\text{cg-0}} \lambda_{\text{cur.}} = \llbracket \dot{e}_1 \rrbracket_{\text{cg-0}} \lambda_{\text{cur.}} \cup_{(\cdot)} \llbracket \dot{e}_2 \rrbracket_{\text{cg-0}} \lambda_{\text{cur.}}$$

$$\llbracket \dot{i} \rrbracket_{\text{cg-0}} \lambda_{\text{cur.}} = (\emptyset, \emptyset).$$

7.7 Finding strongly connected components

From the call graph, $sccs \in ? \rightarrow \Gamma$ constructs the strongly connected components graph $\gamma \in \Gamma = \mathcal{P}\Lambda \times \mathcal{P}(\Lambda \times \Lambda) \times \Lambda$, where $\Lambda = \mathcal{P}\Lambda$. This can be done with a standard algorithm. The code to find the strongly connected components graph, kindly given to us by Kristian Nielsen, is based on (Launchbury, 1993).

7.8 Traversing the strongly connected components graph

To some extent, this section repeats section 5.11. It describes $rdfs$, the overall algorithm for processing the graph of strongly connected components, i.e., the inter-procedural part of the algorithm. The following chapter is devoted to developing the functions called by $rdfs$.

The function $rdfs \in \Gamma \rightarrow \mathbf{K}$ first finds the set $\boldsymbol{\lambda}^{\equiv \text{s}}$ of equivalence classes of λ 's that must use the same linking convention, then sets up the initial inter-procedural environment η_0 , and calls $rdfs_0$ to process the strongly connected components graph in reverse-depth-first search order. Assume γ has the form $(\boldsymbol{\lambda}^{\circ \text{s}}, \mathcal{S}, \boldsymbol{\lambda}_{\text{main}}^{\circ})$ and $\boldsymbol{\lambda}_{\text{main}}^{\circ}$ is $\{\lambda_{\text{main}}\}$:

$$\begin{aligned} rdfs \gamma &= \text{let } \boldsymbol{\lambda}^{\equiv \text{s}} = \llbracket \lambda_{\text{main}} \rrbracket_{\text{uf}} \\ &\quad \eta_0 = (\{ \boldsymbol{\lambda}^{\equiv} \mapsto \text{-lc} \mid \boldsymbol{\lambda}^{\equiv} \in \boldsymbol{\lambda}^{\equiv \text{s}} \}, \\ &\quad \quad \{ \boldsymbol{\lambda}^{\circ} \mapsto \emptyset \mid \boldsymbol{\lambda}^{\circ} \in \boldsymbol{\lambda}^{\circ \text{s}} \}) \\ &\quad (\kappa, \eta) = rdfs_0 \gamma \boldsymbol{\lambda}_{\text{main}}^{\circ} \eta_0 \\ &\quad \text{in } \kappa. \end{aligned}$$

$rdfs_0$ uses *do-scc* to process each node:

$$\begin{aligned}
\text{rdfs}_0 \gamma \lambda_{\text{cur}}^\circ \eta &= \text{let } \{\lambda_1^\circ, \dots, \lambda_l^\circ\} = \text{children } \gamma \lambda_{\text{cur}}^\circ. \\
&(\kappa_1, \eta) = \text{rdfs}_0 \gamma \lambda_1^\circ \eta \\
&\vdots \\
&(\kappa_l, \eta) = \text{rdfs}_0 \gamma \lambda_l^\circ \eta \\
&(\kappa, \eta) = \text{do-scc } \lambda_{\text{cur}}^\circ \eta \\
&\text{in } (\kappa ; \kappa_1 ; \dots ; \kappa_l, \eta).
\end{aligned}$$

Roughly, $\text{do-scc } \lambda^\circ \eta$ uses $\llbracket \cdot \rrbracket_{\text{donode}}$ on all λ 's in λ° :

$$\begin{aligned}
\text{do-scc } \lambda^\circ \eta &= \text{let } \{\lambda_1^\circ, \dots, \lambda_j^\circ\} = \{\llbracket \lambda \rrbracket_{\text{ar-}\Lambda} \lambda^\circ \mid \lambda \in \lambda^\circ\} \\
&\check{\phi} = \llbracket \lambda_1^\circ \rrbracket_{\text{da-}\Lambda} \eta \cup \dots \cup \llbracket \lambda_j^\circ \rrbracket_{\text{da-}\Lambda} \eta \\
&\nu = (\eta, \check{\phi}) \\
&(\kappa_1, \nu) = \llbracket \lambda_1^\circ \rrbracket_{\text{donode}} \nu \\
&\vdots \\
&(\kappa_j, \nu) = \llbracket \lambda_j^\circ \rrbracket_{\text{donode}} \nu \\
&\eta = (\nu^l, \nu^d + \{\lambda^\circ \mapsto \nu^{\check{\phi}}\}) \\
&\text{in } (\kappa_1 ; \dots ; \kappa_j, \eta).
\end{aligned}$$

The arguments of do-scc are the strongly connected component λ° that must be processed and the current inter-procedural environment, η , and do-scc returns the code for the functions in λ° and an updated inter-procedural environment. Before the functions are processed, all applications in them are annotated by $\llbracket \cdot \rrbracket_{\text{ar-}\Lambda}$, which is described below, yielding the recursiveness annotated functions $\{\lambda_1^\circ, \dots, \lambda_j^\circ\}$. Furthermore, the set, $\check{\phi}$, of registers that will be destroyed anyway by λ° is approximated, using $\llbracket \cdot \rrbracket_{\text{da-}\Lambda}$, described below. The function $\llbracket \cdot \rrbracket_{\text{donode}}$ takes and returns a *per-strongly-connected-component environment* ν , which comprises the inter-procedural environment η and the approximation $\check{\phi}$: $\nu = (\eta, \check{\phi})$. We use ν^η to denote the η in ν , and ν^d to denote the η^d in η in ν ; etc. There is no natural order in which to process the λ 's in a strongly connected component; do-scc simply processes them in arbitrary order.

While processing a λ , $\llbracket \cdot \rrbracket_{\text{donode}}$ updates the $\check{\phi}$ in ν whenever a value is allocated to some register. Hence the $\check{\phi}$ in the ν returned by the last $\llbracket \cdot \rrbracket_{\text{donode}}$ tells which registers will be destroyed when a function in the strongly connected component λ° is applied. The inter-procedural environment η returned by do-scc is updated to record this.

The analyses $\llbracket \cdot \rrbracket_{\text{ar-}\Lambda}$ and $\llbracket \cdot \rrbracket_{\text{da-}\Lambda}$ are described in the following sections, and $\llbracket \cdot \rrbracket_{\text{donode}}$ is the subject of the next chapter.

While processing a λ , we have a *per-function environment* $\varepsilon \in \mathbf{E}$ which comprises the current strongly connected component environment, ν , and the function, λ_{cur} , currently being processed, i.e., $\varepsilon = (\nu, \lambda_{\text{cur}})$. The reason is that we need to access, e.g., η^d to translate applications; we need to update

$\check{\phi}$ when a register is destroyed; and we need λ_{cur} . to, e.g., access its closure representation. Summing up the different environments:

$$\underbrace{\underbrace{\underbrace{((\eta^l, \eta^d), \check{\phi}), \lambda_{\text{cur.}})}_{\eta}}_{\nu}}_{\varepsilon} \quad \begin{array}{l} \text{inter-procedural environment} \\ \text{per-strongly-connected-component environment} \\ \text{per-function environment.} \end{array}$$

7.9 Finding the equivalence classes of λ 's

The analysis, $\llbracket \cdot \rrbracket_{\text{uf}}$, to split the λ 's of a program into equivalence classes is a simple union-find algorithm: Traverse the program and make sure that all λ 's that may be applied at the same application are in the same equivalence class.

Here is the algorithm in detail. The actual work is done by $\llbracket \cdot \rrbracket_{\text{uf-0}}$, which has an argument, $\lambda^{\equiv s}$, the set of equivalence classes:

$$\llbracket \cdot \rrbracket_{\text{uf-0}} \in \dot{E} \rightarrow \mathcal{P}(\mathcal{P}\Lambda) \rightarrow \mathcal{P}(\mathcal{P}\Lambda)$$

All $\llbracket \check{e} \rrbracket_{\text{uf}}$ does is to call $\llbracket \check{e} \rrbracket_{\text{uf-0}}$ with an empty set of equivalence classes:

$$\llbracket \check{e} \rrbracket_{\text{uf}} = \llbracket \check{e} \rrbracket_{\text{uf-0}} \emptyset$$

At an application, the equivalence classes of the λ 's that may be applied are merged:

$$\llbracket \check{e}_1 \lambda \check{e}_2 \rrbracket_{\text{uf-0}} \lambda^{\equiv s} = \llbracket \check{e}_1 \rrbracket_{\text{uf-0}} (\llbracket \check{e}_2 \rrbracket_{\text{uf-0}} (\text{union } \{ \text{find } \lambda \lambda^{\equiv s} \mid \lambda \in \lambda \} \lambda^{\equiv s})),$$

where

$$\begin{aligned} \text{union } \{ \lambda_1^{\equiv}, \dots, \lambda_j^{\equiv} \} \lambda^{\equiv s} &= (\lambda^{\equiv s} \setminus \{ \lambda_1^{\equiv}, \dots, \lambda_j^{\equiv} \}) \cup \{ \lambda_1^{\equiv} \cup \dots \cup \lambda_j^{\equiv} \} \\ \text{find } \lambda \lambda^{\equiv s} &= \begin{cases} \lambda^{\equiv}, & \text{if } \lambda \in \lambda^{\equiv} \wedge \lambda^{\equiv} \in \lambda^{\equiv s} \\ \{ \lambda \}, & \text{otherwise.} \end{cases} \end{aligned}$$

Region polymorphic application is dealt with analogously. Other kinds of expressions are simply traversed, e.g.:

$$\begin{aligned} \llbracket \lambda \check{y}. \check{e}_0 \text{ at } \rho \rrbracket_{\text{uf}} &= \llbracket \check{e}_0 \rrbracket_{\text{uf}} \\ \llbracket (\check{e}_1, \dots, \check{e}_n) \text{ at } \rho \rrbracket_{\text{uf}} &= \llbracket \check{e}_1 \rrbracket_{\text{uf}} \circ \dots \circ \llbracket \check{e}_n \rrbracket_{\text{uf}} \\ \llbracket i \rrbracket_{\text{uf}} &= \lambda \lambda^{\equiv s}. \lambda^{\equiv s}. \end{aligned}$$

7.10 Potentially recursive applications

Recall that we expect each application to be annotated with “ \cup ” or “ \emptyset ” according to whether it is potentially recursive or not (section 5.10). This annotation of applications is done by the function $\llbracket \cdot \rrbracket_{\text{ar}}$. It translates a several-argumented expression $\check{e} \in \dot{E}$ to a *recursiveness-annotated expression*

$\overset{\circ}{e} \in \overset{\circ}{E}$ in which all applications are annotated with an $r \in R$ where $R ::= \circ \mid \emptyset$, i.e., the target language, $\overset{\circ}{E}$, of the translation is

$$\overset{\circ}{E} ::= \overset{\circ}{E} \overset{R}{\Lambda} \overset{\vec{\circ}}{E} \mid \overset{R}{F} \overset{\vec{\circ}}{\Lambda} \overset{\vec{\circ}}{E} \mid \dots$$

The rest of the grammar is similar to that for $\overset{\circ}{E}$. The translation function, $\llbracket \cdot \rrbracket_{\text{ar}}$, takes as extra argument the strongly connected component λ° that contains the λ that the expression being translated is directly within:

$$\llbracket \cdot \rrbracket_{\text{ar}} \in \overset{\circ}{E} \rightarrow \Lambda \rightarrow \overset{\circ}{E}.$$

The translation simply traverses the expression and annotates each application directly within it:

$$\begin{aligned} \llbracket \overset{\circ}{e}_0 \lambda \langle \overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n \rangle \rrbracket_{\text{ar}} \lambda^\circ &= \text{let } r = \text{if } \lambda \cap \lambda^\circ \neq \emptyset \text{ then } \circ \text{ else } \emptyset \\ &\quad \overset{\circ}{e}_0 = \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{ar}} \lambda^\circ \\ &\quad \vdots \\ &\quad \overset{\circ}{e}_n = \llbracket \overset{\circ}{e}_n \rrbracket_{\text{ar}} \lambda^\circ \\ &\text{in } \overset{\circ}{e}_0 \overset{r}{\lambda} \langle \overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n \rangle. \end{aligned}$$

Region polymorphic application is dealt with analogously with normal application. Applications that are not directly within the expression are directly within some other λ that may belong to another strongly connected component, and these applications must receive their annotation in the context of that strongly connected component. Therefore, only applications directly within the expression are annotated, and consequently, $\llbracket \cdot \rrbracket_{\text{ar}}$ does not traverse bodies of λ 's within the expression:

$$\llbracket \lambda \vec{y}. \overset{\circ}{e}_0 \text{ at } \rho \rrbracket_{\text{ar}} \lambda^\circ = \lambda \vec{y}. \overset{\circ}{e}_0 \text{ at } \rho$$

$$\begin{aligned} \llbracket \text{letrec } b_1 \dots b_m \text{ at } \rho \text{ in } \overset{\circ}{e}_{m+1} \rrbracket_{\text{ar}} \lambda^\circ &= \\ \text{let } \overset{\circ}{e}_{m+1} = \llbracket \overset{\circ}{e}_{m+1} \rrbracket_{\text{ar}} \lambda^\circ & \\ \text{in letrec } b_1 \dots b_m \text{ at } \rho \text{ in } \overset{\circ}{e}_{m+1}. & \end{aligned}$$

The other constructs are simply traversed, e.g.:

$$\begin{aligned} \llbracket \text{let } x = \overset{\circ}{e}_1 \text{ in } \overset{\circ}{e}_2 \rrbracket_{\text{ar}} \lambda^\circ &= \text{let } \overset{\circ}{e}_1 = \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{ar}} \lambda^\circ \\ &\quad \overset{\circ}{e}_2 = \llbracket \overset{\circ}{e}_2 \rrbracket_{\text{ar}} \lambda^\circ \\ &\text{in let } x = \overset{\circ}{e}_1 \text{ in } \overset{\circ}{e}_2. \end{aligned}$$

The translation has been defined for expressions; a λ is translated simply by translating its body, i.e., define $\llbracket \cdot \rrbracket_{\text{ar}-\Lambda}$ by

$$\begin{aligned} \llbracket \lambda \vec{y}. \overset{\circ}{e}_0 \text{ at } \rho \rrbracket_{\text{ar}-\Lambda} &= \lambda \vec{y}. \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{ar}} \text{ at } \rho \\ \llbracket f \vec{\rho} \vec{y} = \overset{\circ}{e}_0 \rrbracket_{\text{ar}-\Lambda} &= f \vec{\rho} \vec{y} = \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{ar}}. \end{aligned}$$

7.11 Approximating the set of registers that will be destroyed by the code for an expression

This section describes a function that approximates the set of registers that the code for an expression destroys. This is used by the per-function part of the register allocation, $\llbracket \cdot \rrbracket_{\text{ra}}$, but it is also used in the inter-procedural part of the algorithm for approximating the set of registers that will be destroyed anyway by a function, as was described in section 5.10.

Given a recursiveness-annotated expression \mathring{e} and an inter-procedural environment $\eta \in \mathbf{H}$,

$$\llbracket \cdot \rrbracket_{\text{da}} \in \mathring{E} \rightarrow \mathbf{H} \rightarrow \mathcal{P}\Phi$$

must return an approximation of the set of registers that will be destroyed by the code for \mathring{e} . The inter-procedural environment η is needed for approximating which registers are destroyed by applications in \mathring{e} . At an application, the set of registers destroyed is the union of the sets of registers destroyed by the strongly connected components that may be applied. If the set of functions that may be applied is λ , the strongly connected components λ° that may be applied are those for which $\lambda \cap \lambda^\circ \neq \emptyset$. And thus, the set of registers that may be destroyed is

$$\text{destroys } \lambda\eta = \bigcup \{ \eta^{\text{d}} \lambda^\circ \mid \lambda \cap \lambda^\circ \neq \emptyset \wedge \lambda^\circ \in \text{Dm } \eta^{\text{d}} \},$$

and then

$$\begin{aligned} \llbracket \mathring{e}_0 \lambda \langle \mathring{e}_1, \dots, \mathring{e}_n \rangle \rrbracket_{\text{da}} \eta &= \text{destroys } \lambda\eta \\ &\quad \cup \llbracket \mathring{e}_0 \rrbracket_{\text{da}} \eta \cup \dots \cup \llbracket \mathring{e}_n \rrbracket_{\text{da}} \eta \end{aligned}$$

Similarly with region polymorphic applications.

For many constructs there is no way to predict which registers will be destroyed; the approximation of the set of registers destroyed is simply the union of the approximations of their sub-expressions:

$$\begin{aligned} \llbracket v \rrbracket_{\text{da}} \eta &= \emptyset \\ \llbracket i \rrbracket_{\text{da}} \eta &= \emptyset \\ \llbracket u \mathring{e}_2 \rrbracket_{\text{da}} \eta &= \llbracket \mathring{e}_2 \rrbracket_{\text{da}} \eta \\ \llbracket \mathring{e}_1 \circ \mathring{e}_2 \rrbracket_{\text{da}} \eta &= \llbracket \mathring{e}_1 \rrbracket_{\text{da}} \eta \cup \llbracket \mathring{e}_2 \rrbracket_{\text{da}} \eta \\ \llbracket \text{let } x = \mathring{e}_1 \text{ in } \mathring{e}_2 \rrbracket_{\text{da}} \eta &= \llbracket \mathring{e}_1 \rrbracket_{\text{da}} \eta \cup \llbracket \mathring{e}_2 \rrbracket_{\text{da}} \eta. \end{aligned}$$

The code for a put point (an expression that has an “at ρ ”-annotation) will destroy the set $\hat{\phi}_{\text{at}}$ of registers, if the region being allocated in has unknown size, because a $\phi_1 := \text{at } \phi_2 : \nu$ -instruction is used to allocate in regions of unknown size (section 4.4) and that instruction destroys $\hat{\phi}_{\text{at}}$ (chapter 3). Region variables of the form $\rho : ?$ will always be bound to regions of unknown size; so define

$$\llbracket \rho : ? \rrbracket_{\text{da-at}} = \hat{\phi}_{\text{at}}.$$

If the size of the region is known, memory has already been allocated, and no registers will be destroyed by put point code to allocate. Region variables of the form $\varrho:i$ will always be bound to regions of known size; so define

$$\llbracket \varrho:i \rrbracket_{\text{da-at}} = \emptyset.$$

Finally, if the region variable has the form $\varrho:\psi$, it may be bound to both regions of known and regions of unknown size. We cannot predict what registers may be destroyed by the code to allocate in the region bound to ρ ; we rather arbitrarily define

$$\llbracket \varrho:\psi \rrbracket_{\text{da-at}} = \hat{\phi}_{\text{at}}.$$

(We do not require that $\llbracket \cdot \rrbracket_{\text{da}}$ gives a safe approximation in any sense—i.e., $\llbracket \overset{\circ}{e} \rrbracket_{\text{da}} \eta$ does not have to be an approximation that contains at least all the registers that will actually be destroyed by the code for $\overset{\circ}{e}$ —so we could also have chosen to define $\llbracket \varrho:\psi \rrbracket_{\text{da-at}} = \emptyset$.)

Then the approximations for the put point constructs are:

$$\begin{aligned} \llbracket \overset{\circ}{c}_1 \overset{\circ}{e}_2 \text{ at } \rho \rrbracket_{\text{da}} \eta &= \llbracket \rho \rrbracket_{\text{da-at}} \cup \llbracket \overset{\circ}{e}_2 \rrbracket_{\text{da}} \eta \\ \llbracket (\overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n) \text{ at } \rho \rrbracket_{\text{da}} \eta &= \llbracket \rho \rrbracket_{\text{da-at}} \cup \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{da}} \eta \cup \dots \cup \llbracket \overset{\circ}{e}_n \rrbracket_{\text{da}} \eta \\ \llbracket \lambda \vec{y}. \overset{\circ}{e}_0 \text{ at } \rho \rrbracket_{\text{da}} \eta &= \llbracket \rho \rrbracket_{\text{da-at}} \\ \llbracket \text{letrec } b_1 \dots b_m \text{ at } \rho \text{ in } \overset{\circ}{e}_{m+1} \rrbracket_{\text{da}} \eta &= \llbracket \rho \rrbracket_{\text{da-at}} \cup \llbracket \overset{\circ}{e}_{m+1} \rrbracket_{\text{da}} \eta, \end{aligned}$$

etc. Notice that, since this is a per-function analysis, λ 's within the expression are not traversed.

Branches of conditional expressions are traversed:

$$\llbracket \text{if } \overset{\circ}{e}_0 \text{ then } \overset{\circ}{e}_1 \text{ else } \overset{\circ}{e}_2 \rrbracket_{\text{da}} \eta = \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{da}} \eta \cup \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{da}} \eta \cup \llbracket \overset{\circ}{e}_2 \rrbracket_{\text{da}} \eta.$$

The expression `letregion $\varrho:?$ in $\overset{\circ}{e}_1$` will translate to code that uses the instructions `$\phi := \text{letregion}$` and `endregion` (section 4.4). These instructions destroy $\hat{\phi}_{\text{letregion}}$ and $\hat{\phi}_{\text{endregion}}$, respectively (chapter 3). Hence,

$$\llbracket \text{letregion } \varrho:?\text{ in } \overset{\circ}{e}_1 \rrbracket_{\text{da}} \eta = \hat{\phi}_{\text{letregion}} \cup \hat{\phi}_{\text{endregion}} \cup \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{da}} \eta.$$

Memory for regions with known size is allocated on the stack (section 4.4), i.e., no registers are destroyed (except ϕ_{sp} , of course):

$$\llbracket \text{letregion } \varrho:i \text{ in } \overset{\circ}{e}_1 \rrbracket_{\text{da}} \eta = \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{da}} \eta.$$

The analysis has been defined for expressions; a λ is analysed simply by analysing its body, i.e., define $\llbracket \cdot \rrbracket_{\text{da-}\Lambda}$ by

$$\begin{aligned} \llbracket \lambda \vec{y}. \overset{\circ}{e}_0 \text{ at } \rho \rrbracket_{\text{da-}\Lambda} &= \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{da}} \\ \llbracket f \vec{\rho} \vec{y} = \overset{\circ}{e}_0 \rrbracket_{\text{da-}\Lambda} &= \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{da}}. \end{aligned}$$

8 Development of the per-function part of the algorithm

This section develops $\llbracket \cdot \rrbracket_{\text{donode}}$, the per-function part of the algorithm. Basically, $\llbracket \cdot \rrbracket_{\text{donode}}$ processes a λ by first performing the ω -analysis, $\llbracket \cdot \rrbracket_{\text{oa}}$, on it, and then translating its body with $\llbracket \cdot \rrbracket_{\text{ra}}$.

We will develop $\llbracket \cdot \rrbracket_{\text{oa}}$ and $\llbracket \cdot \rrbracket_{\text{ra}}$, before we describe $\llbracket \cdot \rrbracket_{\text{donode}}$ in detail. The chapter is organised thus:

Section 8.1 develops the ω -analysis $\llbracket \cdot \rrbracket_{\text{oa}}$. Then, the register allocation $\llbracket \cdot \rrbracket_{\text{ra}}$ is developed for each construct, in the following 9 sections. Confer with chapter 4 regarding what code to generate for each construct; this chapter only discusses the *register allocation* for the constructs.

Section 8.2 develops the register allocation of $e_1 \text{ oe } e_2$. The interesting issue in this is how to manage temporary values.

When this has been decided, section 8.3 can finally explain how the descriptor δ works.

Section 8.4 develops the translation of **letregion** ρ **in** e_1 . This involves the questions how to implement region variables (especially region variables with known size) and how the register allocation should deal with instructions that destroy a given set of registers.

Section 8.5 discusses the central question of how to translate definitions and uses of values.

Section 8.6 develops the translation of (e_1, \dots, e_n) **at** ρ . The issue of interest is how the register allocation should deal with code that allocates. Register-allocation-wise all put points are dealt with basically as this construct is dealt with.

Section 8.7 develops the translation of the **case**-construct. The problem for the register allocation is the unlinear control flow, which must be taken into account when processing the descriptors, δ .

Section 8.8 develops the translation of **if** e_0 **then** e_1 **else** e_2 . This construct can be treated quite like the **case**-construct, but it is more fun to compile it properly, i.e., to short-circuit code. This is more difficult in SML than in some other languages because the **if**-construct is an expression, and in particular, the condition in an **if**-expression can (and will often in practice) be an **if**-expression itself. Translating Boolean expressions to short-circuit code seems to fit nicely into our method of register allocation and code generation.

Section 8.9 develops the translation of application. This connects the per-function part of the algorithm with the inter-procedural part.

Section 8.10 develops the translation of the exception constructs. The main problem is the irregular control flow exceptions cause.

Section 8.11 wraps up the per-function part of the algorithm by developing $\llbracket \cdot \rrbracket_{\text{donode}}$

The main issues in the register allocation have been covered by the discussion of the constructs mentioned above. The register allocation of the

remaining constructs presents only minor variations of the already discussed. For instance, the register allocation of $\dot{c}_1 e_2$ at ρ is similar to that for a pair (e_1, e_2) at ρ . For completeness, section 8.12 briefly presents the translation of these constructs.

8.1 The ω -analysis

Remember that the ω -information is a map,

$$\omega \in \Omega = V \xrightarrow{\perp} \mathcal{P}\Phi,$$

from live values to sets of registers to which those values are hostile.

Given an inter-procedural environment $\eta \in \mathbb{H}$, the ω -analysis

$$\llbracket \cdot \rrbracket_{\text{oa}} \in \overset{\circ}{E} \rightarrow \mathbb{H} \rightarrow \overset{*}{E}$$

translates a recursiveness-annotated expression $\overset{\circ}{e}$ to an ω -annotated expression $\overset{*}{e} \in \overset{*}{E}$, where

$$\begin{aligned} \overset{*}{E} ::= & \Omega X \Omega \\ & | \Omega \overset{*}{E} \overset{R}{\Lambda} \overset{\vec{}}{E} \Omega \\ & | \Omega \overset{R}{F} \overset{\vec{}}{P} \overset{\vec{}}{E} \Omega \\ & | \Omega \text{let } X = \overset{*}{E} \text{ in } \overset{*}{E} \Omega \\ & | \Omega \text{letregion } \overset{\dot{}}{P} \text{ in } \overset{*}{E} \Omega \\ & \vdots \end{aligned}$$

The ω -analysis is a straightforward backwards analysis. It traverses the expression using an auxiliary function

$$\llbracket \cdot \rrbracket_{\text{oa-0}} \in \overset{\circ}{E} \rightarrow \mathbb{H} \rightarrow \Omega \rightarrow \Omega \times \overset{*}{E},$$

which takes an expression $\overset{\circ}{e}$, an environment η , and an *in-flowing* ω' and yields an *out-flowing* ω and an ω -annotated expression $\overset{*}{e}$. How this works is illustrated by the binary operation construct:

$$\begin{aligned} \llbracket \overset{\circ}{e}_1 \circ \overset{\circ}{e}_2 \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } (\omega_2, \overset{*}{e}_2) = \llbracket \overset{\circ}{e}_2 \rrbracket_{\text{oa-0}} \eta \omega' \\ & \quad (\omega_1, \overset{*}{e}_1) = \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{oa-0}} \eta \omega_2 \\ & \text{in } (\omega_1, \omega_1 \overset{*}{e}_1 \circ \overset{*}{e}_2 \omega'). \end{aligned}$$

In general, the ω -information *after* the $\overset{*}{e}$ that $\llbracket \cdot \rrbracket_{\text{oa-0}}$ yields (i.e. the ω' annotated on the right of $\overset{*}{e}$) is the in-flowing ω' to $\llbracket \cdot \rrbracket_{\text{oa-0}}$, and the ω -information *before* $\overset{*}{e}$ (i.e. the ω annotated on the left of $\overset{*}{e}$) is the same as the out-flowing ω from $\llbracket \cdot \rrbracket_{\text{oa-0}}$.

In the backwards traversal, a variable becomes live the first time it is in a non-binding position, and it dies when it is in a binding position. Since

the ω -analysis is used on a by-function basis, values are only bound by the `let`-, `letregion`-, `letrec`-, and `exception`-constructs. Each time a use of a value, v , is encountered, it is looked up in ω . If v is not in the domain of ω , it is inserted. We define \dagger_{oa} to take care of this:

$$\omega \dagger_{\text{oa}} v = \text{if } v \in \text{Dm } \omega \text{ then } \omega \text{ else } \omega + \{v \mapsto \emptyset\},$$

and then

$$\begin{aligned} \llbracket z \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } \omega = \omega' \dagger_{\text{oa}} z \\ &\quad \text{in } (\omega, \omega z \omega'). \end{aligned}$$

The expression `let $x = \overset{\circ}{e}_1$ in $\overset{\circ}{e}_2$` defines x in $\overset{\circ}{e}_2$; x is not live in the code for $\overset{\circ}{e}_1$, and consequently, it is erased from the ω that flows out of $\overset{\circ}{e}_2$ and into $\overset{\circ}{e}_1$:

$$\begin{aligned} \llbracket \text{let } x = \overset{\circ}{e}_1 \text{ in } \overset{\circ}{e}_2 \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } (\omega_2, \overset{\circ}{e}_2) = \llbracket \overset{\circ}{e}_2 \rrbracket_{\text{oa-0}} \eta \omega' \\ &\quad \omega'_1 = \omega_2 \setminus \{x\} \\ &\quad (\omega_1, \overset{\circ}{e}_1) = \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{oa-0}} \eta \omega'_1 \\ &\quad \text{in } (\omega_1, \omega_1 \text{let } x = \overset{\circ}{e}_1 \text{ in } \overset{\circ}{e}_2 \omega'), \end{aligned}$$

where $\omega \setminus v$ is the restriction of ω to the domain $\text{Dm } \omega \setminus v$.

The expressions `letregion ρ in $\overset{\circ}{e}_1$` and `exception a in $\overset{\circ}{e}_2$` are treated analogously.

At an application which is not potentially recursive, we must take into account that a number of λ 's may be applied at that point. This is recorded in the λ annotated on the application. Since λ 's are processed bottom-up in the call graph, we will know exactly which registers may be destroyed, namely *destroys* $\lambda\eta$ (see section 7.11, p. 128). All variables live across the application must be made hostile to *destroys* $\lambda\eta$. To this end, we introduce the function *antagonise* $\in \mathcal{P}\Phi \rightarrow \Omega \rightarrow \Omega$:

$$\textit{antagonise } \hat{\phi} \omega = \{x \mapsto \omega x \cup \hat{\phi} \mid x \in \text{Dm } \omega\}.$$

Then

$$\begin{aligned} \llbracket \overset{\circ}{e}_0 \overset{\emptyset}{\lambda} \langle \overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n \rangle \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } \omega'_n = \textit{antagonise } (\textit{destroys } \lambda\eta) \omega' \\ &\quad (\omega_n, \overset{\circ}{e}_n) = \llbracket \overset{\circ}{e}_n \rrbracket_{\text{oa-0}} \eta \omega'_n \\ &\quad \vdots \\ &\quad (\omega_0, \overset{\circ}{e}_0) = \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{oa-0}} \eta \omega_1 \\ &\quad \text{in } (\omega_0, \omega_0 \overset{\emptyset}{\lambda} \langle \overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n \rangle \omega'). \end{aligned}$$

If the application *is* potentially recursive, control might flow back to the same program point. In this case, all values in registers will be destroyed, and they will have to be reloaded after the call. Therefore the information about which registers each variable is hostile to is of no relevance to program

points before the application, and the ω -information for each variable is reset to \emptyset using the function $unantagonise \in \Omega \rightarrow \Omega$:

$$\begin{aligned} unantagonise \omega &= \{x \mapsto \emptyset \mid x \in \text{Dm } \omega\}, \\ \llbracket \overset{\circ}{e}_0 \overset{\circ}{\lambda} \langle \overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n \rangle \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } \omega'_n = unantagonise \omega' \\ &\quad (\omega_n, \check{e}_n) = \llbracket \overset{\circ}{e}_n \rrbracket_{\text{oa-0}} \eta \omega'_n \\ &\quad \vdots \\ &\quad (\omega_0, \check{e}_0) = \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{oa-0}} \eta \omega_1 \\ &\text{in } (\omega_0, \omega_0 \overset{\circ}{\lambda} \langle \overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n \rangle \omega'). \end{aligned}$$

The code for an expression that has an “at ρ ”-annotation (i.e. a put point) will destroy the registers $\llbracket \rho \rrbracket_{\text{da-at}}$ (section 7.11). Therefore, we introduce $\llbracket \rho \rrbracket_{\text{oa-at}} \in \Omega \rightarrow \Omega$ to record this in ω . A put point is also a use of the region variable ρ , and this must also be recorded in ω , hence the \dagger_{oa} :

$$\llbracket \rho \rrbracket_{\text{oa-at}} \omega = antagonise \llbracket \rho \rrbracket_{\text{da-at}} \omega \dagger_{\text{oa}} \rho.$$

Notice that ρ is added to ω *after* all the other variables in ω have been *antagonise*'d to $\llbracket \rho \rrbracket_{\text{da-at}}$. This is because ρ is used before the allocation takes place; it need not be live across the code that does the allocation.

The code for the put point $(\overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n)$ **at** ρ (p. 28) first allocates memory in ρ for the tuple. Then the sub-expressions $\overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n$ are evaluated (in that order) and the results are stored in memory. Correspondingly, in the backwards analysis, the sub-expressions are traversed in the order $\overset{\circ}{e}_n$ through $\overset{\circ}{e}_1$, and then $\llbracket \rho \rrbracket_{\text{oa-at}}$ is applied to ω :

$$\begin{aligned} \llbracket (\overset{\circ}{e}_1, \dots, \overset{\circ}{e}_n) \text{ at } \rho \rrbracket_{\text{oa-at}} \eta \omega' &= \text{let } (\omega_n, \check{e}_n) = \llbracket \overset{\circ}{e}_n \rrbracket_{\text{oa-0}} \eta \omega' \\ &\quad \vdots \\ &\quad (\omega_1, \check{e}_1) = \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{oa-0}} \eta \omega_2 \\ &\quad \omega = \llbracket \rho \rrbracket_{\text{oa-at}} \omega_1 \\ &\text{in } (\omega, \omega (\check{e}_1, \dots, \check{e}_n) \text{ at } \rho \omega'). \end{aligned}$$

Other put points are treated similarly, e.g.:

$$\begin{aligned} \llbracket \overset{\circ}{c}_1 \overset{\circ}{e}_2 \text{ at } \rho \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } (\omega_2, \check{e}_2) = \llbracket \overset{\circ}{e}_2 \rrbracket_{\text{oa-0}} \eta \omega' \\ &\quad \omega = \llbracket \rho \rrbracket_{\text{oa-at}} \omega_2 \\ &\text{in } (\omega, \omega \overset{\circ}{c}_1 \check{e}_2 \text{ at } \rho \omega'). \end{aligned}$$

Only the put point $\lambda \vec{y}.^{\mathcal{K}} \overset{\circ}{e}_0 \text{ at } \rho$ is a bit different. The code for it allocates memory for a closure, and stores the free variables of $\lambda \vec{y}.^{\mathcal{K}} \overset{\circ}{e}_0 \text{ at } \rho$ in it. This constitutes a sequence of uses of the free variables, which must be noted in ω as if it were a sequence of normal uses of values:

$$\begin{aligned} \llbracket \lambda \vec{y}.^{\mathcal{K}} \overset{\circ}{e}_0 \text{ at } \rho \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } \{z_1, \dots, z_n\} = \text{Dm } \mathcal{K} \\ &\quad \omega = \omega' \dagger_{\text{oa}} z_1 \dagger_{\text{oa}} \dots \dagger_{\text{oa}} z_n \\ &\quad \omega = \llbracket \rho \rrbracket_{\text{oa-at}} \omega \\ &\text{in } (\omega, \omega \lambda \vec{y}.^{\mathcal{K}} \overset{\circ}{e}_0 \text{ at } \rho \omega'). \end{aligned}$$

The ω -analysis is a per-function analysis, so $\overset{\circ}{e}_0$ is not traversed.

The expression **letrec** $b_1 \cdots b_m$ **at** ρ **in** $\overset{\circ}{e}_{m+1}$ is treated almost exactly the same way; the difference is that the sibling name (p. 117) of the **letrec**-bound λ 's should be removed from ω as if it were an x of a **let**-expression, and $\overset{\circ}{e}_{m+1}$ should also be traversed.

In **if** $\overset{\circ}{e}_0$ **then** x **else** y control may flow through either x or y . Both x and y must be considered live in the ω before the **if**-expression; i.e., the out-flowing ω from $\llbracket \text{if } \overset{\circ}{e}_0 \text{ then } x \text{ else } y \rrbracket_{\text{oa-0}}$ must tell that both x and y are live. Therefore the function \sqcup_{oa} to join the two ω 's that result from the branches is defined

$$\omega_1 \sqcup_{\text{oa}} \omega_2 = \{v \mapsto \omega_1 v \cup \omega_2 v \mid v \in \text{Dm } \omega_1 \cup \text{Dm } \omega_2\},$$

and then

$$\begin{aligned} \llbracket \text{if } \overset{\circ}{e}_0 \text{ then } \overset{\circ}{e}_1 \text{ else } \overset{\circ}{e}_2 \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } (\omega_2, \overset{\circ}{e}_2) = \llbracket \overset{\circ}{e}_2 \rrbracket_{\text{oa-0}} \eta \omega' \\ &\quad (\omega_1, \overset{\circ}{e}_1) = \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{oa-0}} \eta \omega' \\ &\quad \omega'_0 = \omega_1 \sqcup_{\text{oa}} \omega_2 \\ &\quad (\omega_0, \overset{\circ}{e}_0) = \llbracket \overset{\circ}{e}_0 \rrbracket_{\text{oa-0}} \eta \omega'_0 \\ &\text{in } (\omega_0, \omega_0 \text{ if } \overset{\circ}{e}_0 \text{ then } \overset{\circ}{e}_1 \text{ else } \overset{\circ}{e}_2 \omega'). \end{aligned}$$

Notice that the in-flowing ω to both branches is the same.

The expression **case** $\overset{\circ}{e}_0$ **of** $c_1 \Rightarrow \overset{\circ}{e}_1 \mid \dots \mid c_n \Rightarrow \overset{\circ}{e}_n \mid _ \Rightarrow \overset{\circ}{e}_{n+1}$ can be handled similarly, joining the ω 's from the branches $\overset{\circ}{e}_1, \dots, \overset{\circ}{e}_{n+1}$ with \sqcup_{oa} .

When an exception is raised, control flows to the nearest enclosing **handle**-expression, not to the context as usual. Hence, the in-flowing ω to $\llbracket \text{raise } \overset{\circ}{e}_1 \rrbracket_{\text{oa-0}}$ is of no use. This in-flowing ω should be the join \sqcup_{oa} of the out-flowing ω 's of the handlers that could handle a raised exception from this **raise**-expression. This would, however, complicate the ω -analysis with more ω 's, and it would require a non-trivial control flow analysis to decide which **handle**-expressions control might flow to when an exception is raised. To keep things simple, we choose to make a worse approximation: the in-flowing ω to a **raise**-expression is \emptyset :

$$\begin{aligned} \llbracket \text{raise } \overset{\circ}{e}_1 \rrbracket_{\text{oa-0}} \eta \omega' &= \text{let } (\omega, \overset{\circ}{e}_1) = \llbracket \overset{\circ}{e}_1 \rrbracket_{\text{oa-0}} \eta \emptyset \\ &\text{in } (\omega, \omega \text{ raise } \overset{\circ}{e}_1 \emptyset). \end{aligned}$$

With this, the central aspects of the ω -analysis have been covered.

8.2 Temporary values

In this and the following 8 sections, we carry on the development of the translation $\llbracket \cdot \rrbracket_{\text{ra}}$ from where we stopped in chapter 6 with the definition of $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$. In that chapter, $\llbracket e \rrbracket_{\text{ra}}$ took a δ as an argument and returned a resulting δ , but actually it takes and returns a *pair* (δ, ε) containing δ and a per-function environment ε (p. 125). I.e., with $M = \Delta \times E$

$$\llbracket \cdot \rrbracket_{\text{ra}} \in E \rightarrow M \rightarrow (M \times \Phi_{\perp} \times B).$$

The code for $e_1 \circ e_2$ has the form

$$\phi_1 := \boxed{\text{code to evaluate } e_1} ; \phi_2 := \boxed{\text{code to evaluate } e_2} ; \llbracket o \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi,$$

where $\llbracket o \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi$ translates the primitive operator o to instructions that compute the result from ϕ_1 and ϕ_2 and put it in ϕ :

$$\begin{aligned} \llbracket + \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi &= \phi := \phi_1 + \phi_2 \\ \llbracket - \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi &= \phi := \phi_1 - \phi_2 \\ \llbracket := \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi &= \text{m}[\phi_1 + 0] := \phi_2. \end{aligned}$$

Thus, β for $e_1 \circ e_2$ is

$$\beta = \lambda\phi. \lambda\varsigma. \beta_1 \phi_1 \varsigma ; \beta_2 \phi_2 \varsigma ; \llbracket o \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi,$$

and the register allocation has this form

$$\begin{aligned} \llbracket e_1 \circ e_2 \rrbracket_{\text{ra}} (\delta, \varepsilon) &= \text{let } ((\delta, \varepsilon), \overset{\dagger}{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} (\delta, \varepsilon) \\ &\quad \boxed{\text{pick } \phi_1} \\ &\quad ((\delta, \varepsilon), \overset{\dagger}{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} (\delta, \varepsilon) \\ &\quad \boxed{\text{pick } \phi_2} \\ &\quad \beta = \lambda\phi. \lambda\varsigma. \beta_1 \phi_1 \varsigma ; \beta_2 \phi_2 \varsigma ; \llbracket o \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi \\ &\quad \text{in } ((\delta, \varepsilon), \text{-register}, \beta). \end{aligned}$$

When choosing ϕ_1 , we have the following (not always compatible) goals:

1. To allow the result of e_1 to stay in ϕ_1 across the code for e_2 , avoid registers that are known to be destroyed by e_2 , i.e., avoid the registers $\llbracket e_2 \rrbracket_{\text{da}} \varepsilon^\eta$ (section 7.11).
2. To avoid register-to-register moves, prefer the natural destination register, $\overset{\dagger}{\phi}_1$, of e_1 .
3. Aim at the previously discussed four general objectives (p. 85).

Thus, we can pick ϕ_1 with $\text{choose } \varnothing \overset{\dagger}{\phi}_1 (\llbracket e_2 \rrbracket_{\text{da}} \varepsilon^\eta) \omega'_1 \delta$, where ω'_1 is the ω -information after e_1 , the point where the temporary value is defined. (Check the specification of *choose*, p. 87.)

When choosing ϕ_2 , there are no registers that should preferably not be chosen; instead there is a register that *must not* be chosen, viz. ϕ_1 . Thus, we can pick ϕ_2 with $\text{choose } \{\phi_1\} \overset{\dagger}{\phi}_2 \varnothing \omega'_2 \delta$, where ω'_2 is the ω -information after e_2 .

After choosing ϕ_1 and ϕ_2 , we must also record in the descriptor that they contain new values. An auxiliary function, *new-tmp*, similar in spirit to $\llbracket \cdot \rrbracket_{\text{def}}$ (p. 88), does this: $\text{new-tmp } \hat{\phi} \overset{\dagger}{\phi} \hat{\phi} \omega (\delta, \varepsilon)$ chooses a register in the same way $\text{choose } \hat{\phi} \overset{\dagger}{\phi} \hat{\phi} \omega \delta$ does, but also returns an updated δ and ε . Summing up, $\boxed{\text{pick } \phi_1}$ should be

$$((\delta, \varepsilon), \phi_1) = \text{new-tmp } \varnothing \overset{\dagger}{\phi}_1 (\llbracket e_2 \rrbracket_{\text{da}} \varepsilon^\eta) \omega'_1 (\delta, \varepsilon),$$

and $\boxed{\text{pick } \phi_2}$ should be

$$((\delta, \varepsilon), \phi_2) = \text{new-tmp } \{\phi_1\} \dot{\phi}_2 \mathcal{O} \omega'_2(\delta, \varepsilon).$$

If ϕ_1 is destroyed by the code for e_2 , it must be preserved around that code.

Assume *kill-tmp* δ checks whether the value that was last *new-tmp*'ed has been thrown out of its register, and returns a *preserver* $p \in \mathbf{Z} \rightarrow \mathbf{Z}$ of that temporary. The idea is that if ζ is the code that destroys the register ϕ containing the temporary value, then $p\zeta$ is the same code, except that it does not destroy ϕ . Thus, if the temporary value that was last *new-tmp*'ed has been thrown out of its register ϕ , the preserver will be

$$p\zeta = \lambda(\zeta^e, \zeta^p). \text{push } \phi ; \zeta(\zeta^e, \zeta^p + 1) ; \text{pop } \phi.$$

Otherwise, the preserver does nothing:

$$p\zeta = \zeta.$$

kill-tmp δ must also record in δ that ϕ again contains the temporary value.

Then

$$\begin{aligned} \llbracket e_1 \circ e_2 \rrbracket_{\text{ra}}(\delta, \varepsilon) = & \text{let } ((\delta, \varepsilon), \dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}}(\delta, \varepsilon) \\ & ((\delta, \varepsilon), \phi_1) = \text{new-tmp } \mathcal{O} \dot{\phi}_1 (\llbracket e_2 \rrbracket_{\text{da}} \varepsilon^\eta) \omega'_1(\delta, \varepsilon) \\ & ((\delta, \varepsilon), \dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}}(\delta, \varepsilon) \\ & ((\delta, \varepsilon), \phi_2) = \text{new-tmp } \{\phi_1\} \dot{\phi}_2 \mathcal{O} \omega'_2(\delta, \varepsilon) \\ & (\delta, p_2) = \text{kill-tmp } \delta \\ & (\delta, p_1) = \text{kill-tmp } \delta \\ & \beta = \lambda\phi. \lambda\varsigma. \beta_1\phi_1\varsigma ; p_1(\beta_2\phi_2)\varsigma ; \llbracket \mathcal{O} \rrbracket_{\text{o-prim}} \phi_1\phi_2\phi \\ \text{in } & ((\delta, \varepsilon), -\text{register}, \beta). \end{aligned}$$

The first *kill-tmp* δ corresponds to the *new-tmp* for ϕ_2 ; since ϕ_2 cannot have been destroyed, the preserver p_2 will not do anything, and we ignore it. The second *kill-tmp* δ corresponds to the *new-tmp* for ϕ_1 ; the preserver, p_1 , it returns is used to save ϕ_1 across $\beta_2\phi_2$. Notice that this way of treating temporary values is only possible because live ranges of temporaries happen to be nested inside each other.

Intuitively, *new-tmp* marks the beginning of the live range of a temporary, and the corresponding *kill-tmp* marks the end. For instance, the *new-tmp* for ϕ_1 is between $\llbracket e_1 \rrbracket_{\text{ra}}$ and $\llbracket e_2 \rrbracket_{\text{ra}}$ because the live range of the result of e_1 starts between the code for e_1 and that for e_2 .

Notice how *new-tmp* and *kill-tmp* correspond closely to $\llbracket \cdot \rrbracket_{\text{def}}$ (p. 88) and $\llbracket \cdot \rrbracket_{\text{kill}}$ (p. 104), respectively. Their implementation is explained in the next section.

Expressions $e_1 \circ e_2$ do not have a natural destination register. If we returned, say, ϕ_2 as the natural destination register, the context would be

more inclined to choose ϕ_2 . This would be unfortunate, since ϕ_2 might contain a value worth preserving. For instance, choosing ϕ_2 to hold the result of $\mathbf{a+b}$ in the expression $(\mathbf{a+b})-\mathbf{b}$ would evict \mathbf{b} from its register and necessitate a reload of \mathbf{b} to compute the subtraction.

A remark on notation: Some meta-functions have discouragingly many arguments, but many arguments can be ignored. As δ records the current state of the registers, it is passed to and returned from almost every meta-function. The flow of δ 's is only non-trivial where control flow is not linear, e.g., in **if**-expressions. You can also ignore the environment ε , which is passed to and returned from any meta-function that may need to access or update the inter-procedural information, e.g., the set $\varepsilon^{\check{\phi}}$ of registers that will be destroyed by $\lambda_{\text{cur.}}$, or the linking conventions map ε^d . The $\hat{\phi}$ -argument is always a set of registers that must not be chosen. The $\hat{\phi}^{\dagger}$ -argument is always a register that should preferably be chosen. The $\hat{\phi}$ -argument is always a set of registers that should preferably be avoided. The ω -argument is always the ω -annotation at the appropriate program point. These arguments are supposed to ultimately end up as arguments to *choose*, and can to a wide extent be ignored. The arguments $\hat{\phi}$, $\hat{\phi}^{\dagger}$, $\hat{\phi}$, ω , and (δ, ε) are always passed in that order.

8.3 The descriptor δ

Having explained the treatment of temporaries, we are now in a position to explain precisely what a descriptor δ is, and how exactly the operations on descriptors work. This explanation can be skipped, as it should be possible to comprehend the rest of the development of the translation with only an intuitive understanding of how the descriptor δ and, e.g., *new-tmp* and *kill-tmp* works.

The descriptor has the form

$$\delta = (\delta^v, \delta^t, \delta^d).$$

The component $\delta^d \in \Phi \rightarrow D$ maps each register to a *description* of its contents. The descriptions D are

$$\begin{array}{l} D ::= W \mid \square \\ W ::= V \mid -_d \\ V ::= Z \mid \mathbf{clos} \mid \mathbf{ret} \\ Z ::= X \mid Y \mid \mathbf{F} \mid \mathbf{P} \mid A. \end{array}$$

(The set V of values is recalled from section 6.3.) If $\delta^d \phi = v \in V$, then ϕ contains the value v ; if $\delta^d \phi = -_d$, then ϕ contains an unnamed value (either a temporary value or a dead value); otherwise, $\delta^d \phi = \square$, meaning that ϕ has not been touched yet.

If a value bound directly within $\lambda_{\text{cur.}}$ is loaded (p. 103), its producer must preserve it. We record in $\delta^v \subseteq V$ the set of values bound directly within $\lambda_{\text{cur.}}$

that are loaded. Therefore, $\llbracket v \rrbracket_{\text{has-been-loaded}}$ δ , which records in δ that v is loaded, is defined:

$$\llbracket v \rrbracket_{\text{has-been-loaded}} \left(\delta^v, \delta^t, \delta^d \right) = \left(\delta^v \cup \{v\}, \delta^t, \delta^d \right).$$

And $\llbracket v \rrbracket_{\text{kill}}$ $\phi \delta$, which checks whether v is loaded according to δ and returns a preserver of ϕ , is defined:

$$\begin{aligned} \llbracket v \rrbracket_{\text{kill}} \phi \left(\delta^v, \delta^t, \delta^d \right) &= \\ &= \left(\delta^v \setminus \{v\}, \delta^t, \delta^d \right), \text{ if } v \in \delta^v \text{ then } \textit{preserve } \phi \text{ else } \textit{don't} \end{aligned}$$

$$\textit{preserve } \phi = \lambda \zeta. \lambda (\zeta^e, \zeta^p). \text{ push } \phi ; \zeta (\zeta^e + \{x \mapsto \zeta^p\}, \zeta^p + 1) ; \text{ pop}$$

$$\textit{don't} = \lambda \zeta. \zeta.$$

See $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$ (p. 104) for an example of a use of $\llbracket v \rrbracket_{\text{kill}}$.

The live ranges of temporary values are nested inside each other such that the currently live temporaries can be kept on a stack: when a temporary value is introduced in δ (e.g., using *new-tmp*) it is pushed on this stack; when it is killed (using *kill-tmp*), it is popped. The function *kill-tmp* returns a preserver of the register containing the temporary, hence we must also keep a preserver with each temporary on the stack. More specifically, using $P = \mathbb{Z} \rightarrow \mathbb{Z}$ for the set of preservers, $\delta^t \in (\Phi \times (W \times P))^*$ is a stack of currently live temporaries of the form $\phi \mapsto (w, p)$, meaning that the register ϕ contains the temporary value w and is preserved by p .

We use $\delta \vdash_t \phi$ to introduce the value in ϕ as a temporary. The temporary value is the value $\delta^d \phi$ currently in ϕ , and the preserver should initially not save ϕ ; thus $\delta \vdash_t \phi$ should push $\phi \mapsto (\delta^d \phi, \textit{don't})$ on δ^t :

$$\left(\delta^v, \delta^t, \delta^d \right) \vdash_t \phi = \left(\delta^v, (\phi \mapsto (\delta^d \phi, \textit{don't})) \cdot \delta^t, \delta^d \right),$$

where \cdot is defined $\mathbf{t} \cdot (\mathbf{t}_1, \dots, \mathbf{t}_m) = (\mathbf{t}, \mathbf{t}_1, \dots, \mathbf{t}_m)$.

Whenever a ϕ containing a temporary value is destroyed, we must change its preserver to

$$\textit{preserve-tmp } \phi = \lambda \zeta. \lambda (\zeta^e, \zeta^p). \text{ push } \phi ; \zeta (\zeta^e, \zeta^p + 1) ; \text{ pop } \phi.$$

(Compare with *preserve* ϕ above.) We use $\phi \odot \delta^t$ to change the preserver of ϕ if ϕ occurs in δ^t , i.e.,

$$\begin{aligned} \phi \odot (\phi_1 \mapsto (w_1, p_1), \dots, \phi_i \mapsto (w_i, p_i), \dots, \phi_m \mapsto (w_m, p_m)) \\ = (\phi_1 \mapsto (w_1, p_1), \dots, \phi_i \mapsto (w_i, \textit{preserve-tmp } \phi_i), \dots, \phi_m \mapsto (w_m, p_m)), \end{aligned}$$

if $\phi = \phi_i$ and $\phi \neq \phi_j$ for all $j < i$; if ϕ does not occur in δ^t , $\phi \odot \delta^t = \delta^t$.

We use *kill-tmp* δ to pop the topmost temporary and return its preserver:

$$\begin{aligned} \textit{kill-tmp} \left(\delta^v, (\phi_1 \mapsto (w_1, p_1), \dots, \phi_m \mapsto (w_m, p_m)), \delta^d \right) = \\ = \left(\delta^v, (\phi_2 \mapsto (w_2, p_2), \dots, \phi_m \mapsto (w_m, p_m)), \delta^d + \{\phi_1 \mapsto w_1\} \right), p_1, \end{aligned}$$

Notice that *kill-tmp* sets δ^d to map ϕ to the temporary value, implicitly assuming that the caller of *kill-tmp* will ensure that ϕ is preserved, i.e., that p_1 is used appropriately.

The basic operation of the register allocation is to assign some value w to a register ϕ . $ra(\phi \mapsto w)(\delta, \varepsilon)$ yields $(\delta_1, \varepsilon_1)$ that records that w has been assigned to ϕ . ra must do three things: 1° the register descriptor δ^d must be updated to map ϕ to w (i.e., $\delta_1^d \phi = w$); 2° if ϕ contained a temporary before this assignment to ϕ , we must take measures to preserve ϕ (i.e., $\delta_1^t = \phi \odot \delta^t$); 3° for the benefit of the inter-procedural part of the register allocation we must record that ϕ will be destroyed by the code for λ_{cur} . (i.e., $\phi \in \varepsilon_1^{\check{\phi}}$): (remember ε has the form $((\eta, \check{\phi}), \lambda_{\text{cur}})$)

$$\begin{aligned} ra(\phi \mapsto w)((\delta^v, \delta^t, \delta^d), ((\eta, \check{\phi}), \lambda_{\text{cur}})) &= \\ &= ((\delta^v, \phi \odot \delta^t, \delta^d + \{\phi \mapsto w\}), ((\eta, \check{\phi} \cup \{\phi\}), \lambda_{\text{cur}})). \end{aligned}$$

These were the basic operations on δ . We now explain some auxiliary functions that use them.

The translation $\llbracket e \rrbracket_{ra}$ returns the code β for e along with a natural destination register $\check{\phi}$. If $\check{\phi}$ is not $-\text{register}$, this has the interpretation “the code puts the result of e in $\check{\phi}$; if you choose a different destination register ϕ , a move instruction will be inserted”.

This move from $\check{\phi}$ to ϕ should be reflected in δ . To update δ , we define the function *move* such that $move \check{\phi} \phi(\delta, \varepsilon)$ yields (δ', ε') where δ' records that ϕ contains the same value as $\check{\phi}$ does:

$$\begin{aligned} move -\text{register } \phi_2(\delta, \varepsilon) &= ra(\phi_2 \mapsto -_d)(\delta, \varepsilon) \\ move \phi_1 \phi_2(\delta, \varepsilon) &= ra(\phi_2 \mapsto \delta^d \phi_1)(\delta, \varepsilon). \end{aligned}$$

The function *new-tmp* is used to find a temporary register in the situation sketched above, i.e., given a natural destination register $\check{\phi}$. It uses *choose* to pick a register ϕ ; *move* to record in δ the potential copying of $\check{\phi}$ to ϕ ; and $\delta \vdash_t \phi$ to record in δ that ϕ , from this point on contains a temporary value:

$$\begin{aligned} new-tmp \check{\phi} \phi \hat{\phi} \omega(\delta, \varepsilon) &= \text{let } \phi = choose \hat{\phi} \check{\phi} \omega \delta \\ &(\delta, \varepsilon) = move \check{\phi} \phi(\delta, \varepsilon) \\ &\delta = \delta \vdash_t \phi \\ &\text{in } ((\delta, \varepsilon), \phi). \end{aligned}$$

Often a temporary value is *kill-tmp*'ed right after it has been *new-tmp*'ed, viz. when its live range does not extend across the code for any sub-expression. Therefore, we introduce the shorthand *tmp-tmp*:

$$\begin{aligned} tmp-tmp \hat{\phi} \check{\phi} \omega(\delta, \varepsilon) &= \text{let } ((\delta, \varepsilon), \phi) = new-tmp \hat{\phi} \check{\phi} \omega(\delta, \varepsilon) \\ &(\delta, p) = kill-tmp \delta \\ &\text{in } ((\delta, \varepsilon), \phi). \end{aligned}$$

(The preserver p will be *don't* because the temporary value cannot have been evicted.)

8.4 Allocating regions

The code for `letregion ρ :? in e_1` is (p. 31)

$$\phi_\rho := \text{letregion} ; \phi_1 := \boxed{\text{code to evaluate } e_1} ; \text{endregion.}$$

Aside from the allocation and deallocation, a `letregion`-expression is like a `let`-expression: Both define a variable in the sub-expression, and the ρ of the `letregion`-expression is treated no differently than the x of the `let`-expression. For instance, if ρ is loaded in e_1 , it must be pushed on the stack around the code for e_1 :

$$\phi_\rho := \text{letregion} ; \text{push } \phi_\rho ; \phi_1 := \boxed{\text{code to evaluate } e_1} ; \text{pop} ; \text{endregion};$$

if ρ is not loaded in e_1 , just delete `push ϕ_ρ` and `pop`.

The δ and ε are always passed together so we shall use μ to stand for (δ, ε) . As usual, μ^δ then denotes the δ -component of μ , and μ^D denotes the D -component of the δ -component of μ , etc.

When doing register allocation, we must record in (the δ -component of) μ that `$\phi := \text{letregion}$` destroys the set, $\hat{\phi}_{\text{letregion}}$, of registers, and `endregion` the set $\hat{\phi}_{\text{endregion}}$. This is done with *wipe*, defined

$$\begin{aligned} \text{wipe } \phi \mu &= \text{ra } (\phi \mapsto -_d) \mu \\ \text{wipe } \{\phi_1, \dots, \phi_k\} \mu &= \text{wipe } \phi_1 (\dots (\text{wipe } \phi_k \mu) \dots). \end{aligned}$$

Here follows $\llbracket \text{letregion } \rho: ? \text{ in } e_1 \rrbracket_{\text{ra}}$. Explanations are below.

$$\begin{aligned} \llbracket \text{letregion } \rho: ? \text{ in } e_1 \rrbracket_{\text{ra}} \mu &= \\ \text{let } \mu &= \text{wipe } \hat{\phi}_{\text{letregion}} \mu \\ (\mu, \phi_\rho) &= \llbracket \rho: ? \rrbracket_{\text{def}} \phi_{\text{letregion}} \omega_1 \mu \\ (\mu, \hat{\phi}_1, \beta_1) &= \llbracket e_1 \rrbracket_{\text{ra}} \mu \\ (\mu, p_\rho) &= \llbracket \rho: ? \rrbracket_{\text{kill}} \phi_\rho \mu \\ (\mu, \phi_t) &= \text{tmp-tmp } \hat{\phi}_{\text{endregion}} \hat{\phi}_1 \omega' \mu \\ \mu &= \text{wipe } \hat{\phi}_{\text{endregion}} \mu \\ \beta &= \lambda \phi. \lambda \varsigma. \text{let } \phi' = \text{if } \phi \in \hat{\phi}_{\text{endregion}} \text{ then } \phi_t \text{ else } \phi \text{ in} \\ &\quad \phi_\rho := \text{letregion} ; p_\rho(\beta_1 \phi')(\varsigma^e, \varsigma^p + n_{\text{r.d.}}) ; \text{endregion} ; \\ &\quad \langle \phi := \phi' \rangle \\ \text{in } &(\mu, -_{\text{register}}, \beta), \end{aligned}$$

where ω_1 is the ω -information before e_1 , and ω' is the ω -information after the whole `letregion`-expression. $n_{\text{r.d.}}$ is the number of words that `$\phi_\rho := \text{letregion}$` pushes on the stack, i.e., the size of a region descriptor (with the implementation sketched in section 3.2, $n_{\text{r.d.}}$ is 4). The sub-expression e_1 gets a stack shape that is $n_{\text{r.d.}}$ words bigger than the stack shape of its context.

Notice the resemblance with $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$. The *wipe* $\hat{\phi}_{\text{letregion}} \mu$ is first, because `$\phi_\rho := \text{letregion}$` is the first instruction in the code. Then a

register for the region variable is found using $\llbracket \varrho : ? \rrbracket_{\text{def}}$, exactly as if $\varrho : ?$ were a **let**-bound x . The **letregion**-instruction will preferably return the pointer to the region in the register $\phi_{\text{letregion}}$. Therefore we pass $\phi_{\text{letregion}}$ to $\llbracket \varrho : ? \rrbracket_{\text{def}}$ as the preferred register for $\varrho : ?$, hoping to avoid a move instruction. The *wipe* $\hat{\phi}_{\text{endregion}}\mu$ is last because **endregion** is last in the code.

We tacitly extend functions that take and return a δ (e.g., $\llbracket \varrho : ? \rrbracket_{\text{kill}} \phi_\rho$) to take and return a μ . Thus $\llbracket \varrho : ? \rrbracket_{\text{kill}} \phi_\rho \mu$ stands for $(\llbracket \varrho : ? \rrbracket_{\text{kill}} \phi_\rho \mu^\delta, \mu^\varepsilon)$.

Why is it necessary to have the temporary register ϕ_t ? The result of the **letregion**-expression is the result of e_1 , hence the result of e_1 must be in some register ϕ' at the end of the code for the **letregion**-expression. This ϕ' must be chosen such that it is not destroyed by the **endregion** after the code for e_1 , i.e., we must choose ϕ' such that $\phi' \notin \hat{\phi}_{\text{endregion}}$. But it is the context of **letregion** $\varrho : ?$ in e_1 that decides the destination register, and the context does not know that it must not choose a register in $\hat{\phi}_{\text{endregion}}$, and hence, we cannot simply apply β_1 to the register to which β is applied; we must take measures to ensure that the result of e_1 is preserved across **endregion**. Thus ϕ_t is needed to hold the value across **endregion**. If ϕ , the register to which β is applied, happens not to be destroyed by **endregion**, we can avoid using ϕ_t , and hence, avoid the move instruction $\phi := \phi'$ —but notice that the *tmp-tmp* is never avoided, and thus ϕ_t always appears (in μ) to be changed, even when it is not.

We could have a concept of registers that must not be chosen as destination registers, and return this set together with a β as the translation of an expression. Then we could demand that a β never be applied to a register in this set. The **letregion** $\varrho : ?$ in e_1 is, however, the only construct whose code does not compute the result as the last thing, and this single construct does not warrant such a complication of the whole algorithm.

The code for **letregion** $\varrho : i$ in e_1 is (p. 32)

$$\begin{aligned} \phi_{\text{sp}} &:= \phi_{\text{sp}} + \llbracket i \rrbracket_{I \rightarrow I} ; \\ \phi_1 &:= \boxed{\text{code to evaluate } e_1} ; \\ \phi_{\text{sp}} &:= \phi_{\text{sp}} - \llbracket i \rrbracket_{I \rightarrow I} . \end{aligned}$$

Here the region variable is treated differently from a **let**-bound variable in two respects: (1) It is not automatically in a register at the entry to the code for the sub-expression, because the region is allocated by just changing the stack pointer. A **let**-bound x is automatically in a register, because it is the result of an expression, and a **letregion**-bound $\dot{\rho}$ with unknown size is automatically in a register, because the $\phi := \text{letregion}$ -instruction assigns it to ϕ . (2) It is not necessary to explicitly store a pointer to a region with known size, since it can always be found at a statically known offset on the stack. Therefore, the pointer to the region need not be saved on the stack, although it is spilled in e_1 , and we need not reserve a register to hold it. Note that this does not mean that $\varrho : i$ will never reside in a register—if it first gets loaded to a register, it might stay there till the next time it is needed.

We keep track of the place of a region on the stack the same way we keep

track of variables that are saved on the stack (section 6.12): by recording in the environment, ζ^e , the current (compile-time) stack pointer, ζ^p , when $\varrho:i$ is allocated on the stack. (So $\zeta^e \rho = i$ can mean two very different things! If ρ has the form $\varrho:?$ or $\varrho:\psi$, it means the same as $\zeta^e x = i$: that ρ is a variable loaded in e_1 , and can be found on the stack at position i . If ρ has the form $\varrho:i$, it means that the *region itself* can be found on the stack at position i .)

$$\begin{aligned} \llbracket \text{letregion } \varrho:i \text{ in } e_1 \rrbracket_{\text{ra}} \mu &= \\ \text{let } (\mu, \hat{\phi}_1, \beta_1) &= \llbracket e_1 \rrbracket_{\text{ra}} \mu \\ \beta &= \lambda\phi. \lambda(\zeta^e, \zeta^p). \phi_{\text{sp}} := \phi_{\text{sp}} + \llbracket i \rrbracket_{I \rightarrow I} ; \\ &\quad \beta_1 \phi(\zeta^e + \{(\varrho:i) \mapsto \zeta^p\}, \zeta^p + i) ; \\ &\quad \phi_{\text{sp}} := \phi_{\text{sp}} - \llbracket i \rrbracket_{I \rightarrow I} \\ \text{in } (\mu, \hat{\phi}_1, \beta). \end{aligned}$$

8.5 Defining and using values

Variables are defined by the constructs **let** $x = e_1$ **in** e_2 , **letregion** $\varrho:?$ **in** e_1 , **letrec** $b_1 \cdots b_m$ **at** ρ **in** e_{m+1} , and **exception** a **in** e_2 . In these cases, $\llbracket v \rrbracket_{\text{def}} \hat{\phi} \omega \mu$ is used to find a register for v and update μ accordingly:

$$\begin{aligned} \llbracket v \rrbracket_{\text{def}} \hat{\phi} \omega \mu &= \text{let } \phi_v = \text{choose } \hat{\phi} \omega (\omega v) \omega \mu \\ &\quad \mu = \text{ra } (\phi_v \mapsto v) \mu \\ \text{in } (\mu, \phi_v). \end{aligned}$$

We discussed how to translate a use of a **let**-bound x on p. 102. Now we generalise this to uses of any value v .

If v is not in a register, we must reload it (remember that v is in ϕ_v according to μ iff $\mu^d \phi_v = v$):

$$\begin{aligned} \llbracket v \rrbracket_{\text{use}} \hat{\phi} \omega \mu &= \\ \text{if } \exists \phi_v \in \text{Dm } \mu^d : \mu^d \phi_v = v &\text{ then } (\mu, \phi_v, \lambda\phi. \lambda\zeta. \langle \phi := \phi_v \rangle) \\ \text{else } \llbracket v \rrbracket_{\text{load}} \hat{\phi} \omega \mu. \end{aligned}$$

The way v must be loaded depends on its kind. If v is a free variable in λ_{cur} , it is loaded from the closure: (We explain code after it has been presented.)

$$\begin{aligned} \llbracket v \rrbracket_{\text{load}} \hat{\phi} \omega \mu &= \\ \text{if } v \in \text{Dm } \mathcal{K} &\text{ then} \\ \text{let } (\mu, \phi_{\text{clos}}, \beta_{\text{clos}}) &= \llbracket \text{clos} \rrbracket_{\text{use}} \hat{\phi} - \text{register } \omega \mu \\ (\mu, \phi_v) &= \llbracket v \rrbracket_{\text{def}} \hat{\phi} \omega \mu \\ \beta_v &= \lambda\phi. \lambda\zeta. \beta_{\text{clos}} \phi_{\text{clos}} \zeta ; \phi_v := \text{m}[\phi_{\text{clos}} + \mathcal{K}v] ; \langle \phi := \phi_v \rangle \\ \text{in } (\mu, \phi_v, \beta_v) \\ \text{else } \dots \end{aligned}$$

Here \mathcal{K} is the closure representation annotated on $\lambda_{\text{cur.}}$ ($= \mu^{\lambda_{\text{cur.}}}$). v is a free variable of $\lambda_{\text{cur.}}$ iff $v \in \text{Dm } \mathcal{K}$.

To fetch something from the closure, we need **clos**. Since **clos** is a value like any other, it can be obtained with $\llbracket \mathbf{clos} \rrbracket_{\text{use}}$.

After that, a register, ϕ_v , is chosen for v and μ is updated accordingly by $\llbracket v \rrbracket_{\text{def}}$ —intuitively a point in the code where a value is loaded is like a “def” of that value.

Thus, the code to fetch v consists of code $\beta_{\text{clos}}\phi_{\text{clos}}$ that ensures ϕ_{clos} points to a tuple of the free variables, and then code $\phi_v := \mathfrak{m}[\phi_{\text{clos}} + \mathcal{K}v]$ that fetches v from its offset, $\mathcal{K}v$, in this tuple.

If v is a region variable with known size, i.e., v has the form $\varrho:i$ (and it is not a free variable in $\lambda_{\text{cur.}}$), it is allocated on the stack at an offset that can be computed at compile-time as follows. If $\varsigma = (\varsigma^e, \varsigma^p)$ is the current stack shape, i.e., ς^p is the current (compile-time) stack pointer, and $\varsigma^e(\varrho:i)$ is the stack offset where $\varrho:i$ resides, then the address of the region can be computed at run-time by subtracting $\varsigma^p - \varsigma^e(\varrho:i)$ from ϕ_{sp} :

```

else if  $\exists(\varrho:i) : v = \varrho:i$  then
  let  $(\mu, \phi_v) = \llbracket \varrho:i \rrbracket_{\text{def}} \hat{\phi} \hat{\omega} \mu$ 
       $\beta_v = \lambda\phi. \lambda(\varsigma^e, \varsigma^p). \phi_v := \phi_{\text{sp}} - \llbracket \varsigma^p - \varsigma^e(\varrho:i) \rrbracket_{I \rightarrow I} ; \langle \phi := \phi_v \rangle$ 
  in  $(\mu, \phi_v, \beta_v)$ 
else ...

```

Again, this use of $\varrho:i$ is conceptually a “def” of the value, hence the $\llbracket \varrho:i \rrbracket_{\text{def}}$.

If v is sibling name \mathbf{f} (p. 117), and \mathbf{f} is the same as the sibling name of $\lambda_{\text{cur.}}$, i.e., if $\lambda_{\text{cur.}}$ has the form $f\vec{\rho}\vec{y} = \mathcal{K}e_0$ and $f \in \mathbf{f}$, then \mathbf{f} is actually an access to the shared closure built for $\lambda_{\text{cur.}}$. This shared closure is **clos**, hence:

```

else if  $v = \mathbf{f} \wedge \mu^{\lambda_{\text{cur.}}} = (f\vec{\rho}\vec{y} = \mathcal{K}e_0) \wedge f \in \mathbf{f}$ 
  then  $\llbracket \mathbf{clos} \rrbracket_{\text{use}} \hat{\phi} \hat{\omega} \mu$ 
  else ...

```

To see that this special case is not an ungraceful, ad hoc optimisation, consider the example

```

letrec f y = y+f v at r1 in f 117.

```

To evaluate this expression a closure containing \mathbf{v} is built and passed to \mathbf{f} . At the recursive call, the function must pass the closure to itself. The special treatment given to sibling accesses ensures that this is done by simply leaving **clos** ($=\mathbf{f}$) in its register. If we did not take care of this special case, we would have to regard \mathbf{f} as a free variable, which is unnatural, and fetch it from the closure at each recursive call, which is less efficient.

Finally, if none of the situations above apply, v must be bound in $\lambda_{\text{cur.}}$; it will be accessible on the stack, and $\varsigma^e v$ gives the stack offset where it resides:

else let $(\mu, \phi_v) = \llbracket v \rrbracket_{\text{def}} \hat{\phi} \hat{\phi} \omega \mu$
 $\mu = \llbracket v \rrbracket_{\text{has-been-loaded}} \mu$
 $\beta_v = \lambda \phi. \lambda (\zeta^e, \zeta^p). \phi_v := \mathfrak{m}[\phi_{\text{sp}} - \llbracket \zeta^p - \zeta^e v \rrbracket_{I \rightarrow I}] ; \langle \phi := \phi_v \rangle$
in (μ, ϕ_v, β_v) .

As always, the reload of v corresponds to a “def”. The stack offset from which to fetch v is computed exactly like the stack offset of a region on the stack is computed. The only new thing is that we must record in μ that v is loaded, such that when the producer of v is translated, it will generate code to push v and record in ζ^e the stack offset of v (recall that, e.g., **let** $x = e_1$ **in** e_2 is the producer of x ; compare with the translation of that expression, p. 104).

Because we regard function parameters (**clos**, **ret**, the arguments y_1, \dots, y_n , and the region arguments $\hat{\rho}_1, \dots, \hat{\rho}_k$) and exception constructors as bound values, the algorithm above also caters for these.

8.6 Put points

This section explains register allocation of put points by developing the translation of the tuple construct. The code for (e_1, \dots, e_n) **at** ρ is (p. 28):

$$\begin{aligned}
\phi_t &:= \boxed{\text{the address of } n \text{ new cells in } \rho} ; \\
\phi_1 &:= \boxed{\text{code to evaluate } e_1} ; \\
\mathfrak{m}[\phi_t + \llbracket 0 \rrbracket_{I \rightarrow I}] &:= \phi_1 ; \\
&\vdots \\
\phi_n &:= \boxed{\text{code to evaluate } e_n} ; \\
\mathfrak{m}[\phi_t + \llbracket n - 1 \rrbracket_{I \rightarrow I}] &:= \phi_n ; \\
\phi &:= \phi_t.
\end{aligned}
\tag{*}$$

An address of the consecutive memory cells supposed to hold the tuple is put into ϕ_t (with $\phi_t := \boxed{\text{the address of } n \text{ new cells in } \rho}$, which has been explained previously (p. 33)), and then the values of the tuple are stored at offsets 0 through $n - 1$ from ϕ_t .

This way of viewing the code to put data in a region is not entirely correct. If the region has known size, it will be allocated on the stack, and in some cases (to be made more precise below) it is not necessary to have an explicit pointer to the region in a register. Instead, the location of the region is represented by its offset from the stack pointer, which (in those cases) is known at compile-time. In such a case, the code for (e_1, \dots, e_n) **at** ρ will be

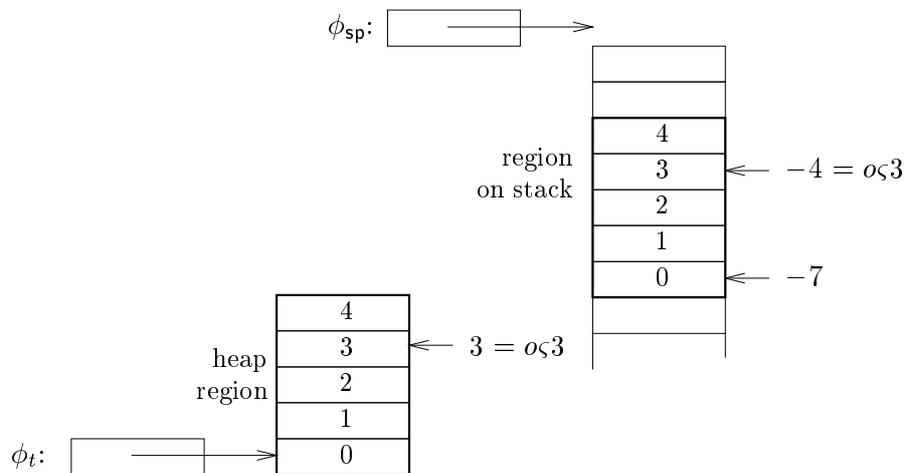
$$\begin{aligned}
\phi_1 &:= \boxed{\text{code to evaluate } e_1} ; \\
m[\phi_{\text{sp}} + \boxed{\text{stack offset of } \rho} + 0]_{I \rightarrow I'} &:= \phi_1 ; \\
&\vdots \\
\phi_n &:= \boxed{\text{code to evaluate } e_n} ; \\
m[\phi_{\text{sp}} + \boxed{\text{stack offset of } \rho} + n - 1]_{I \rightarrow I'} &:= \phi_n ; \\
\phi &:= \phi_{\text{sp}} + \boxed{\text{stack offset of } \rho} .
\end{aligned}
\tag{**}$$

The differences between the two code fragments are:

1. the register in the store instructions (the *indexing register*): a temporary, ϕ_t , in (*); the stack pointer, ϕ_{sp} , in (**);
2. the code to allocate n words in ρ : $\phi_t := \boxed{\text{the address of } n \text{ new cells in } \rho}$ in (*); nothing in (**);
3. the offset from the indexing register: 0 in (*); $\boxed{\text{stack offset of } \rho}$ in (**).

We factor the similarities in allocating in a region into an auxiliary function $\llbracket \cdot \rrbracket_{\text{ra-at}} \cdot \llbracket \rho \rrbracket_{\text{ra-at}} n \hat{\phi} \omega \mu$ will return $(\mu_1, \phi_*, \zeta, o)$, where

1. ϕ_* is either some temporary register, ϕ_t , to point into the region or it is ϕ_{sp} .
2. ζ is code to allocate n words in ρ . It is abstracted over a stack shape, because it might be necessary to access ρ on the stack.
3. o is a function that gives the offset that should be added to the indexing register, given ζ and the offset i into the newly allocated area. If the region is on the stack, e.g., at offset -7 (remember the K stack grows upwards), $o_{\zeta}3$ will yield -4 , the offset from ϕ_{sp} to access the 3rd word in the region (counting from 0). If the region is not on the stack, $o_{\zeta}3$ will yield 3.



The $\hat{\phi}$ -argument to $\llbracket \cdot \rrbracket_{\text{ra-at}}$ is a set of registers that ϕ_* should preferably not be chosen among. The value in ϕ_* must be preserved across the code for the sub-expressions e_1, \dots, e_n , hence, ϕ_* should preferably not be chosen among the set of registers that e_1, \dots, e_n are known to destroy. This set can be computed by the analysis $\llbracket \cdot \rrbracket_{\text{da}}$ described in section 7.11, i.e.,

$$\hat{\phi} = \llbracket e_1 \rrbracket_{\text{da}} \mu^\eta \cup \dots \cup \llbracket e_n \rrbracket_{\text{da}} \mu^\eta.$$

(Remember μ has an ε -component which has an η -component.)

Using $\llbracket \cdot \rrbracket_{\text{ra-at}}$, the translation of (e_1, \dots, e_n) **at** ρ can be written generally:

$$\begin{aligned} \llbracket (e_1, \dots, e_n) \text{ at } \rho \rrbracket_{\text{ra}} \mu = & \\ \text{let } \hat{\phi} = \llbracket e_1 \rrbracket_{\text{da}} \mu^\eta \cup \dots \cup \llbracket e_n \rrbracket_{\text{da}} \mu^\eta & \\ (\mu, \phi_*, \zeta, o) = \llbracket \rho \rrbracket_{\text{ra-at}} n \hat{\phi} \omega_1 \mu & \\ \left. \begin{aligned} (\mu, \dot{\phi}_1, \beta_1) &= \llbracket e_1 \rrbracket_{\text{ra}} \mu \\ (\mu, \phi_1) &= \text{tmp-tmp } \{\phi_*\} \dot{\phi}_1 \omega'_1 \mu \\ (\mu, p_1) &= \text{kill-tmp } \mu \\ \mu &= \mu \dagger_t \phi_* \\ &\vdots \\ (\mu, \dot{\phi}_n, \beta_n) &= \llbracket e_n \rrbracket_{\text{ra}} \mu \\ (\mu, \phi_n) &= \text{tmp-tmp } \{\phi_*\} \dot{\phi}_n \omega'_n \mu \\ (\mu, p_n) &= \text{kill-tmp } \mu \end{aligned} \right\} \text{process } e_1 & \\ \left. \begin{aligned} (\mu, \dot{\phi}_n, \beta_n) &= \llbracket e_n \rrbracket_{\text{ra}} \mu \\ (\mu, \phi_n) &= \text{tmp-tmp } \{\phi_*\} \dot{\phi}_n \omega'_n \mu \\ (\mu, p_n) &= \text{kill-tmp } \mu \end{aligned} \right\} \text{process } e_n & \\ \beta &= \lambda \phi. \lambda \zeta. \\ &\zeta \zeta ; \\ &p_1 (\beta_1 \phi_1) \zeta ; \text{m}[\phi_* + o\zeta 0] := \phi_1 ; \\ &\vdots \\ &p_n (\beta_n \phi_n) \zeta ; \text{m}[\phi_* + o\zeta (n-1)] := \phi_n ; \\ &\langle \phi := \phi_* + o\zeta 0 \rangle \\ \dot{\phi} &= \text{if } \phi_* = \phi_{\text{sp}} \text{ then } \text{-register} \text{ else } \phi_* \\ \text{in } &(\mu, \dot{\phi}, \beta), \end{aligned}$$

where ω'_j is the ω -information after e_j , and ω_1 is the ω -information before e_1 .

The results of the sub-expressions are stored into the tuple right after they have been computed, so the registers ϕ_1, \dots, ϕ_n hold temporary values that are not live across any sub-expression, and hence they are obtained using *tmp-tmp*. As it must be ensured that ϕ_* is not accidentally chosen as one of ϕ_1, \dots, ϕ_n , the set of registers that *must not* be chosen by *tmp-tmp* is $\{\phi_*\}$.

The indexing register, ϕ_* , must be preserved across any sub-expression whose code destroys that register. This is done the same way ϕ_1 was pre-

served across the code for e_2 in $\llbracket e_1 \circ e_2 \rrbracket_{\text{ra}}$ (section 8.2): If the code for e_j destroys ϕ_* , the corresponding preserver p_j returned by *kill-tmp* μ will take care of saving ϕ_* across the code for e_j . After each *kill-tmp* (except the last), ϕ_* is reinstalled as a temporary register using $\mu +_t \phi_*$.

The *optional add instruction* $\langle \phi := \phi' + \iota \rangle$ is defined

$$\langle \phi := \phi' + \iota \rangle = \text{if } \iota = 0 \text{ then } \langle \phi := \phi' \rangle \text{ else } \phi := \phi' + \iota.$$

If the code for (e_1, \dots, e_n) at ρ has the form $(*)$, $\langle \phi := \phi_* + o\zeta 0 \rangle$ will be $\langle \phi := \phi_t + 0 \rangle$, i.e., $\langle \phi := \phi_t \rangle$, and hence ϕ_t is a natural destination register. If the code has the form $(**)$, $\langle \phi := \phi_* + o\zeta 0 \rangle$ will be $\phi := \phi_{\text{sp}} + \boxed{\text{stack offset of } \rho}$, and there will be no natural destination register.

Now the auxiliary function $\llbracket \cdot \rrbracket_{\text{ra-at}}$ is discussed. As mentioned above, we want to give special treatment to a known-size region when it can be determined at compile-time that it will always reside at a specific offset from the stack pointer. Consider

```
letregion r666:2 in let x = (λy.(3,33) at r666:2) at r1:?
                    in x(letregion r999:2
                        in x((6,69) at r999:2))
```

At the two applications of \mathbf{x} , the pair $(3,33)$ must be stored at different offsets from the stack pointer, because $\mathbf{r666:2}$ is on the top of the stack at the outer application, while $\mathbf{r999:2}$ is above $\mathbf{r666:2}$ at the inner application.

Generally, regions that are created before λ_{cur} is applied (as $\mathbf{r666:2}$ is created before \mathbf{x} is applied) do not usually have fixed offsets from the stack pointer from the point of view of the code for λ_{cur} . On the other hand, regions that are created after λ_{cur} is applied have fixed offsets from the stack pointer that can be decided at compile-time. The latter regions are exactly the regions bound to region variables by expressions of the form **letregion** $\varrho:i$ in e_1 directly within λ_{cur} , i.e., $\varrho:i$'s that are not free variables of λ_{cur} .

In summary, if ρ has the form $\varrho:i$ and is not a free variable of λ_{cur} , the code to put data in the region bound to ρ is $(**)$: ϕ_* is ϕ_{sp} and the code, ζ , to allocate is ϵ . This is the then-branch in $\llbracket \varrho:i \rrbracket_{\text{ra-at}}$ below. In all other cases, i.e., when ρ is a free variable of λ_{cur} (the else-branch in $\llbracket \varrho:i \rrbracket_{\text{ra-at}}$ below) or has the form $\varrho:?$ or $\varrho:\psi$ (the other case for $\llbracket \cdot \rrbracket_{\text{ra-at}}$), the code to put data in the region bound to ρ is $(*)$: ϕ_* is some ϕ_t , and the code, ζ , to allocate is $\phi_t := \boxed{\text{the address of } n \text{ new cells in } \rho}$. The form of this code depends on the form of ρ ; this was explained on p. 33.

$$\llbracket \varrho : i \rrbracket_{\text{ra-at}} n\hat{\phi}\omega\mu =$$

if $(\varrho : i) \notin \text{Dm } \mathcal{K}$ then let $\zeta = \lambda\varsigma. \epsilon$
 $o = \lambda\varsigma. \lambda i. \llbracket i - |\varsigma^{\text{P}} - \varsigma^{\text{e}}(\varrho : i) \rrbracket \rrbracket_{I \rightarrow I}$
in $(\mu, \phi_{\text{sp}}, \zeta, o)$

else let $(\mu, \phi_{\rho}, \beta_{\rho}) = \llbracket \varrho : i \rrbracket_{\text{use}} \wp_{\text{-register}} \omega\mu$
 $(\mu, \phi_t) = \text{new-tmp} \wp_{\phi_{\rho}} \hat{\phi}\omega\mu$
 $\zeta = \beta_{\rho} \phi_{\rho}$
 $o = \lambda\varsigma. \lambda i. \llbracket i \rrbracket_{I \rightarrow I}$
in (μ, ϕ_t, ζ, o) .

where \mathcal{K} is the closure representation annotated on $\lambda_{\text{cur.}} = \mu^{\lambda_{\text{cur.}}}$. $(\varrho : i)$ is a free variable of $\lambda_{\text{cur.}}$ iff $(\varrho : i) \in \text{Dm } \mathcal{K}$. If an indexing register, ϕ_t , is necessary, it is chosen with *new-tmp*, and the preferred choice is ϕ_{ρ} , the register chosen by $\llbracket \cdot \rrbracket_{\text{use}}$ to hold the region variable.

When ρ has the form $\dot{\rho} \in \{\varrho : ?, \varrho : \psi\}$:

$$\llbracket \dot{\rho} \rrbracket_{\text{ra-at}} n\hat{\phi}\omega\mu =$$

let $(\mu, \phi_{\rho}, \beta_{\rho}) = \llbracket \dot{\rho} \rrbracket_{\text{use}} \wp_{\phi_{\text{arg.}}^{\text{at}}} \omega\mu$
 $\mu = \text{wipe } \hat{\phi}_{\text{at}}\mu$
 $(\mu, \phi_t) = \text{new-tmp} \wp_{\phi_{\text{res.}}^{\text{at}}} \hat{\phi}\omega\mu$
 $\zeta = \lambda\varsigma. \beta_{\rho} \phi_{\rho} \varsigma ;$
if $\exists \varrho : \dot{\rho} = \varrho : ?$
then $\phi_t := \text{at } \phi_{\rho} : \llbracket n \rrbracket_{I \rightarrow I}$
else if $\phi_{\rho}.\text{lsb}$ then ι_{unknown} else $\iota_{\text{known}} ;$
 $\iota_{\text{known}} : \langle \phi_t := \phi_{\rho} \rangle ; \text{goto } \checkmark ;$
 $\iota_{\text{unknown}} : \phi_t := \text{at } \phi_{\rho} : \llbracket n \rrbracket_{I \rightarrow I} ; \text{goto } \checkmark ;$
 $\checkmark : \epsilon$

$o = \lambda\varsigma. \lambda i. \llbracket i \rrbracket_{I \rightarrow I}$
in (μ, ϕ_t, ζ, o) ,

where the labels are fresh.

We tell $\llbracket \cdot \rrbracket_{\text{use}}$ to preferably put $\dot{\rho}$ into $\phi_{\text{arg.}}^{\text{at}}$, the register in which the $\phi_1 := \text{at } \phi_2 : \iota$ -instruction prefers its argument. (Remember this instruction may be implemented on the concrete machine as a sub-routine; then $\phi_{\text{arg.}}^{\text{at}}$ is the register in which this sub-routine takes its argument.)

Although the instruction $\phi_t := \text{at } \phi_{\rho} : \llbracket n \rrbracket_{I \rightarrow I}$ might not be executed, we must conservatively assume that it is and set $\mu = \text{wipe } \hat{\phi}_{\text{at}}\mu$. The indexing register ϕ_t is chosen with *new-tmp*, and we prefer that *new-tmp* chooses the preferred result register $\phi_{\text{res.}}^{\text{at}}$ of the $\phi_1 := \text{at } \phi_2 : \iota$ -instruction.

8.7 Control flow forks

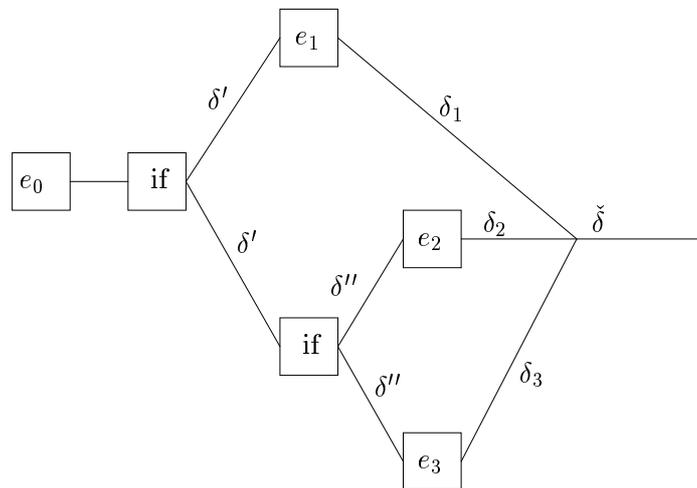
The code for

$$\text{case } e_0 \text{ of Sex} \Rightarrow e_1 \mid \text{Drugs} \Rightarrow e_2 \mid _ \Rightarrow e_3$$

must evaluate e_0 to get the constructed value, fetch the constructor from this, and, according to what this is, jump to code that evaluates either e_1 , e_2 or e_3 (p. 31):

$$\begin{aligned} \phi_0 &:= \boxed{\text{code to evaluate } e_0} ; \\ \phi_c &:= \text{m}[\phi_0 + 0] ; \quad (\text{fetch constructor}) \\ \text{if } \phi_c = \boxed{\text{Sex}}_{C \rightarrow I} &\text{ then } \iota_1 \text{ else } \bar{\iota}_1 ; \\ \bar{\iota}_1 : \text{if } \phi_c = \boxed{\text{Drugs}}_{C \rightarrow I} &\text{ then } \iota_2 \text{ else } \bar{\iota}_2 ; \\ \iota_1 : \phi &:= \boxed{\text{code to evaluate } e_1} ; \text{ goto } \check{\iota} ; \\ \iota_2 : \phi &:= \boxed{\text{code to evaluate } e_2} ; \text{ goto } \check{\iota} ; \\ \bar{\iota}_2 : \phi &:= \boxed{\text{code to evaluate } e_3} ; \text{ goto } \check{\iota} ; \\ \check{\iota} : &\epsilon . \end{aligned}$$

The forking of control flow must be taken into account when processing the descriptors δ , or else they will contain wrong information. For the example, the δ 's must flow as illustrated here:



Boxes are code. Edges indicate control flow and the corresponding “flow of δ 's”: δ_1 is the δ resulting from the register allocation of e_1 , etc. The **if**-code does not destroy any registers, so actually $\delta' = \delta''$. We must decide what the descriptor, $\check{\delta}$, at the point where the control flow paths meet should be: If ϕ' has been destroyed according to one of the δ_j 's, then ϕ' must be marked as destroyed in $\check{\delta}$. If two branches have put different values in ϕ' , then

we mark ϕ' in $\check{\delta}$ as containing some undefined value. We denote by $\delta_1 \sqcap_\delta \delta_2$ a δ that will “agree with” δ_1 and δ_2 ; how \sqcap_δ works is described below.

$$\left[\begin{array}{l} \text{case } e_0 \text{ of } c_1 \Rightarrow e_1 \mid \\ \quad \vdots \\ c_n \Rightarrow e_n \mid \\ - \Rightarrow e_{n+1} \end{array} \right]_{\text{ra}} (\delta, \varepsilon) = \text{let } \begin{array}{l} ((\delta, \varepsilon), \dot{\phi}_0, \beta_0) = \llbracket e_0 \rrbracket_{\text{ra}} (\delta, \varepsilon) \\ ((\delta, \varepsilon), \phi_0) = \text{tmp-tmp} \oslash \dot{\phi}_0 \omega'_0 (\delta, \varepsilon) \\ ((\delta, \varepsilon), \phi_c) = \text{tmp-tmp} \oslash -_{\text{register}} \omega'_0 (\delta, \varepsilon) \\ ((\delta_1, \varepsilon), \dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} (\delta, \varepsilon) \\ \quad \vdots \\ ((\delta_{n+1}, \varepsilon), \dot{\phi}_{n+1}, \beta_{n+1}) = \llbracket e_{n+1} \rrbracket_{\text{ra}} (\delta, \varepsilon) \\ \delta = \delta_1 \sqcap_\delta \cdots \sqcap_\delta \delta_{n+1} \\ \beta = \lambda\phi. \lambda\varsigma. \beta_0 \phi_0 \varsigma ; \phi_c := \text{m}[\phi_0 + 0] ; \\ \quad \text{if } \phi_c = \llbracket c_1 \rrbracket_{C \rightarrow 1} \text{ then } \iota_1 \text{ else } \bar{\iota}_1 ; \\ \bar{\iota}_1 : \quad \text{if } \phi_c = \llbracket c_2 \rrbracket_{C \rightarrow 1} \text{ then } \iota_2 \text{ else } \bar{\iota}_2 ; \\ \bar{\iota}_2 : \\ \quad \vdots \\ \bar{\iota}_{n+1} : \text{if } \phi_c = \llbracket c_n \rrbracket_{C \rightarrow 1} \text{ then } \iota_n \text{ else } \iota_{n+1} ; \\ \iota_1 : \quad \beta_1 \phi_\varsigma ; \text{goto } \check{\iota} ; \\ \iota_2 : \\ \quad \vdots \\ \iota_{n+1} : \beta_{n+1} \phi_\varsigma ; \text{goto } \check{\iota} ; \\ \check{\iota} : \quad \epsilon \end{array} \\ \text{in } ((\delta, \varepsilon), -_{\text{register}}, \beta).$$

where ω'_0 is the ω -information at the program point after e_0 , and all labels are fresh. The result of e_0 is used right away; it is not live across any sub-expressions, so *tmp-tmp* will do to find a register, ϕ_0 , to hold it temporarily. Likewise with ϕ_c . By simply applying all β_i 's to ϕ we force all branches to put the result in the same register.

The details of meeting δ 's

To understand the following, you must have read section 8.3. Define \sqcap_δ by

$$\left(\delta_1^v, \delta_1^t, \delta_1^d \right) \sqcap_\delta \left(\delta_2^v, \delta_2^t, \delta_2^d \right) = \left(\delta_1^v \sqcap_v \delta_2^v, \delta_1^t \sqcap_t \delta_2^t, \delta_1^d \sqcap_d \delta_2^d \right).$$

The set of values that are loaded in $\delta_1 \sqcap_\delta \delta_2$ is the union of the sets of values that are loaded in the two δ 's:

$$\sqcap_v = \cup.$$

The register descriptor $\delta_1^d \sqcap_d \delta_2^d$ maps each register to a value if both δ_1^d and δ_2^d map it to that value; otherwise that register is mapped to $-_d$:

$$\delta_1^d \sqcap_d \delta_2^d = \lambda\phi. \delta_1^d \phi \sqcap_D \delta_2^d \phi$$

$$d_1 \sqcap_D d_2 = \text{if } d_1 = d_2 \text{ then } d_1 \text{ else } -_d.$$

Live ranges of temporaries respect the structure of the source language in the sense that temporaries are always *kill-tmp*'ed in the register allocation of the same expression that it was *new-tmp*'ed. If a branch in a **case**-expression pushes a new temporary on the stack of temporaries it will also pop it again. Therefore, the stacks of temporaries in δ_1 and δ_2 have the same size, and the registers and the temporary values are the same. Only the preservers may be different, since one branch may destroy a register the other leaves untouched. If ϕ should be preserved in δ_1^t or δ_2^t , it must also be preserved in $\delta_1^t \sqcap_t \delta_2^t$:

$$\begin{aligned} & (\phi_1 \mapsto (w_1, p_1), \dots, \phi_m \mapsto (w_m, p_m)) \\ & \sqcap_t (\phi_1 \mapsto (w_1, p'_1), \dots, \phi_m \mapsto (w_m, p'_m)) \\ = & (\phi_1 \mapsto (w_1, \sqcap_P \phi_1 p_1 p'_1), \dots, \phi_m \mapsto (w_m, \sqcap_P \phi_m p_m p'_m)), \end{aligned}$$

where

$$\sqcap_P \phi p p' = \text{if } p \neq p' \text{ then } \textit{preserve-tmp } \phi \text{ else } p.$$

8.8 Boolean expressions

Boolean expressions can be translated like any other kind of expression. But generating code that evaluates the expression and puts the result (true or false) in a register is not necessary when the resulting Boolean value is only used to decide which of two branches to evaluate.

The expression

if b andalso i<=j then 3 else 666

is a *derived form* (Milner et al., 1990) of

if (if b then i<=j else false) then 3 else 666.

If we did not want to treat Boolean expressions specially, we could simply regard **true** and **false** as constructors and **if e_0 then e_1 else e_2** as a derived form of **case e_0 of true => e_1 | _ => e_2** (as does indeed the definition of SML). The expression would then be a derived form of the expression

**case (case b of true => i<=j | _ => false) of
true => 3 | _ => 666,**

and the code for this expression would, according to the discussion in the previous section, be as in figure 37 (*i*).

<pre> if $\phi_b = 1$ then ι_1 else ι_4 ; ι_1 : if $\phi_i \leq \phi_j$ then ι_2 else ι_3 ; ι_2 : $\phi_1 := 1$; goto ι_5 ; ι_3 : $\phi_1 := 0$; goto ι_5 ; ι_4 : $\phi_1 := 0$; ι_5 : if $\phi_1 = 1$ then ι_6 else ι_7 ; ι_6 : $\phi := 3$; goto ι_8 ; ι_7 : $\phi := 666$; ι_8 : ϵ </pre>	<pre> if $\phi_b = 1$ then ι_1 else ι_7 ; ι_1 : if $\phi_i \leq \phi_j$ then ι_6 else ι_7 ; ι_6 : $\phi := 3$; goto ι_8 ; ι_7 : $\phi := 666$; ι_8 : ϵ </pre>
(i) naive translation	(ii) "right" translation

Fig. 37. (i) The generated code if if-expressions are treated as case-expressions. True is represented as 1 and false as 0. (ii) This code avoids explicitly representing the Boolean result from the sub-expressions $i \leq j$ and $\text{if } b \text{ then } i \leq j \text{ else false}$.

The central idea to avoid the unnecessary manipulation of run-time representations of Boolean values is not to translate a Boolean expression into code β that accepts a *register*, but rather into a *selector* σ that accepts a *pair of labels* $(\iota, \bar{\iota})$ and returns code that evaluates the Boolean expression and jumps to ι if the result is true and to $\bar{\iota}$ if it is false: $\sigma \in \Sigma = (I \times I) \rightarrow Z$. (Compare this with $\beta \in B = \Phi \rightarrow Z$.)

We change the translation function to translate a Boolean expression into a $\sigma \in \Sigma$ while expressions of other types are still translated into a $\beta \in B$. In other words, we introduce $\tau \in T ::= \Sigma \mid B$, and then instead of

$$\llbracket \cdot \rrbracket_{\text{ra}} \in E \rightarrow M \rightarrow (M \times \Phi_{\perp} \times B)$$

we have

$$\llbracket \cdot \rrbracket_{\text{ra}} \in E \rightarrow M \rightarrow (M \times \Phi_{\perp} \times T).$$

The code for $e_1 \leq e_2$ should jump to one of two labels according to what Boolean value the expression evaluates to:

$$\phi_1 := \boxed{\text{code to evaluate } e_1} ; \phi_2 := \boxed{\text{code to evaluate } e_2} ; \\ \text{if } \phi_1 \leq \phi_2 \text{ then } \iota \text{ else } \bar{\iota}.$$

Compare this with the code for $e_1 + e_2$ (section 8.2):

$$\phi_1 := \boxed{\text{code to evaluate } e_1} ; \phi_2 := \boxed{\text{code to evaluate } e_2} ; \llbracket + \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi,$$

where

$$\llbracket + \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \phi = \phi := \phi_1 + \phi_2.$$

We define $\llbracket \leq \rrbracket_{\text{o-prim}}$ and $\llbracket = \rrbracket_{\text{o-prim}}$ analogously:

$$\llbracket \leq \rrbracket_{\text{o-prim}} \phi_1 \phi_2 (\iota, \bar{\iota}) = \text{if } \phi_1 \leq \phi_2 \text{ then } \iota \text{ else } \bar{\iota} \\ \llbracket = \rrbracket_{\text{o-prim}} \phi_1 \phi_2 (\iota, \bar{\iota}) = \text{if } \phi_1 = \phi_2 \text{ then } \iota \text{ else } \bar{\iota}.$$

Notice how $\llbracket o \rrbracket_{\text{o-prim}} \phi_1 \phi_2$ is a function of a pair of destination *labels* when o is a conditional operator, and a function of the destination *register* when o is not a conditional operator. (The translation of = works because we only allow = on values that can be represented in one word (p. 2).)

Having factored the individual properties of the operators into the function $\llbracket \cdot \rrbracket_{\text{o-prim}}$, the translation $\llbracket e_1 o e_2 \rrbracket_{\text{ra}}$ can be stated generally for all operators (Boolean as well as others). It is the same as in section 8.2 except that we use τ instead of β to indicate that what was previously always a β now might also be a σ , and $\xi \in \Xi = \Phi \cup (\text{I} \times \text{I})$ instead of ϕ where either a register ϕ or a pair $(\iota, \bar{\iota})$ of labels may appear:

$$\begin{aligned} \llbracket e_1 o e_2 \rrbracket_{\text{ra}} \mu &= \text{let } (\mu, \dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \mu \\ &\quad (\mu, \phi_1) = \text{new-tmp } \wp \dot{\phi}_1 (\llbracket e_2 \rrbracket_{\text{da}} \mu^\eta) \omega_2 \mu \\ &\quad (\mu, \dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \mu \\ &\quad (\mu, \phi_2) = \text{tmp-tmp } \{\phi_1\} \dot{\phi}_2 \omega'_2 \mu \\ &\quad (\mu, p_1) = \text{kill-tmp } \mu \\ &\quad \tau = \lambda \xi. \lambda \varsigma. \beta_1 \phi_1 \varsigma ; p_1 (\beta_2 \phi_2) \varsigma ; \llbracket o \rrbracket_{\text{o-prim}} \phi_1 \phi_2 \xi \\ &\text{in } (\mu, \text{-register}, \tau). \end{aligned}$$

(The use of β_1 rather than τ_1 and ϕ_1 rather than ξ_1 , etc. is deliberate: e_1 cannot be a Boolean expression; $\llbracket e_1 \rrbracket_{\text{ra}}$ will never return a σ .)

The code for the expressions **true** and **false** jump to the true label and the false label, respectively:

$$\begin{aligned} \llbracket \text{true} \rrbracket_{\text{ra}} \mu &= (\mu, \text{-register}, \lambda(\iota, \bar{\iota}). \lambda \varsigma. \text{goto } \iota), \\ \llbracket \text{false} \rrbracket_{\text{ra}} \mu &= (\mu, \text{-register}, \lambda(\iota, \bar{\iota}). \lambda \varsigma. \text{goto } \bar{\iota}). \end{aligned}$$

For expressions that return a selector the natural destination register is always -register . (It really does not make sense to have a natural destination register for an expression that returns a selector, but it is convenient to define that the natural destination register for such an expression is -register .)

The selector for the expression **not** e_1 is obtained by swapping the labels of the selector for e_1 :

$$\begin{aligned} \llbracket \text{not } e_1 \rrbracket_{\text{ra}} \mu &= \text{let } (\mu, \dot{\phi}_1, \sigma_1) = \llbracket e_1 \rrbracket_{\text{ra}} \mu \\ &\quad \text{in } (\mu, \dot{\phi}_1, \lambda(\iota, \bar{\iota}). \sigma_1(\bar{\iota}, \iota)). \end{aligned}$$

To generate code for **if** e_0 **then** e_1 **else** e_2 the Boolean expression e_0 is translated to a selector σ . This is applied to labels ι and $\bar{\iota}$ that label the code for the branches e_1 and e_2 , respectively:

$$\begin{aligned}
\llbracket \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rrbracket_{\text{ra}} \mu &= \\
\text{let } ((\delta_0, \varepsilon), \dot{\phi}_0, \sigma) &= \llbracket e_0 \rrbracket_{\text{ra}} (\delta, \varepsilon) \\
((\delta_1, \varepsilon), \dot{\phi}_1, \tau_1) &= \llbracket e_1 \rrbracket_{\text{ra}} (\delta_0, \varepsilon) \\
((\delta_2, \varepsilon), \dot{\phi}_2, \tau_2) &= \llbracket e_2 \rrbracket_{\text{ra}} (\delta_0, \varepsilon) \\
\delta &= \delta_1 \sqcap_{\delta} \delta_2 \\
\tau &= \lambda \xi. \lambda \varsigma. \\
&\sigma(\iota, \bar{\iota})_{\varsigma} ; \iota : \tau_1 \xi_{\varsigma} ; \text{goto } \check{\iota} ; \bar{\iota} : \tau_2 \xi_{\varsigma} ; \check{\iota} : \varepsilon \\
\text{in } ((\delta, \varepsilon), \text{-register}, \tau), &
\end{aligned}$$

where the labels are fresh.

It is an important point that also the branches of the **if**-expression may translate to σ 's. Consider the expression **if (if 48<=i then i<=57 else false) then i-48 else 0** (which in SML has the derived form **if 48<=i andalso i<=57 then i-48 else 0**).

The branches of the inner **if** are translated to the σ 's:

$$\begin{aligned}
\sigma_{\text{i}<=57} &= \lambda(\iota, \bar{\iota}). \lambda \varsigma. \phi_{57} := 57 ; \text{if } \phi_i \leq \phi_{57} \text{ then } \iota \text{ else } \bar{\iota}, \\
\sigma_{\text{false}} &= \lambda(\iota, \bar{\iota}). \lambda \varsigma. \text{goto } \bar{\iota}
\end{aligned}$$

And then the inner **if**-expression is translated to the following σ that will give code to jump to ι , if i is between 48 and 57, and to $\bar{\iota}$ otherwise:

$$\begin{aligned}
\sigma_{\text{if}} &= \lambda(\iota, \bar{\iota}). \lambda \varsigma. \phi_{48} := 48 ; \text{if } \phi_{48} \leq \phi_i \text{ then } \iota_{\text{i}<=57} \text{ else } \iota_{\text{false}} ; \\
&\iota_{\text{i}<=57} : \phi_{57} := 57 ; \text{if } \phi_i \leq \phi_{57} \text{ then } \iota \text{ else } \bar{\iota} ; \\
&\iota_{\text{false}} : \text{goto } \bar{\iota}.
\end{aligned}$$

The “... else ι_{false} ... $\iota_{\text{false}} : \text{goto } \bar{\iota}$ ” should, of course, be converted to “... else $\bar{\iota}$ ”. This is best done in an ensuing phase (section 9.1).

Note that both branches of an **if**-expression must either translate to σ 's or they must both translate to β 's.

The branches of a **case**-expression can be translated to either β 's or σ 's, as the branches of an **if**-expression can. So modify the translation of **case** e_0 **of** $c_1 \Rightarrow e_1 \mid \dots \mid c_n \Rightarrow e_n \mid _ \Rightarrow e_{n+1}$ in the previous section by replacing β_1, \dots, β_n with τ_1, \dots, τ_n , β with τ , and ϕ with ξ .

Some Boolean expressions do not naturally translate to code that jumps to labels: The constructs z , $\#i e_2$, $\downarrow e_2$, and $! e_2$ naturally translate to β 's. Applications, $e_1 \vec{e}_2$ and $f \vec{\rho} \vec{e}_2$, also naturally translate to β 's, for we do not want to have “Boolean functions” that return by jumping to one of two argument labels. The code for **letregion** ρ **in** e_1 must end with code to deallocate the region ρ , therefore we cannot (easily) translate a **letregion** ρ **in** e_1 into a selector, although e_1 translates to one. So we require a **letregion**-expression to always translate to a β .

If one of these Boolean expressions occur in a context that needs a σ rather than a β (e.g. as e_0 of **if** e_0 **then** e_1 **else** e_2), the β must be converted

to a σ . This is done by generating code that checks what truth value the β computes and then jumps to the labels accordingly: $coerce_{B \rightarrow \Sigma}(\mu, \dot{\phi}, \beta)\omega'$ converts β to a σ (assuming $\dot{\phi}$ is the natural destination register for β , and μ is the appropriate environment, and ω' is the relevant ω -information):

$$\begin{aligned} coerce_{B \rightarrow \Sigma}(\mu, \dot{\phi}, \beta)\omega' &= \text{let } (\mu, \phi) = tmp-tmp \ \varnothing \dot{\phi} \omega' \mu \\ &\quad \sigma = \lambda(\iota, \bar{\iota}). \lambda\varsigma. \beta\phi\varsigma ; \text{ if } \phi = 1 \text{ then } \iota \text{ else } \bar{\iota} \\ &\quad \text{in } (\mu, \text{-register}, \sigma). \end{aligned}$$

Conversely, Boolean expressions may appear in contexts that want the code for the expression to put the resulting value in a register instead of jumping to labels. For instance in `let b=a<=c in e`, we want the result of the expression `a<=c` as a value in a register.

If an expression is in a context that wants a β , but it translates to a σ , the σ must be converted to a β . The function $coerce_{\Sigma \rightarrow B}$ gives a β that puts the representation of true or false into the destination register according to what the selector σ chooses:

$$\begin{aligned} coerce_{\Sigma \rightarrow B}(\mu, \dot{\phi}, \sigma) &= \\ \text{let } \beta &= \lambda\phi. \lambda\varsigma. \sigma(\iota, \bar{\iota})\varsigma ; \iota : \phi := 1 ; \text{ goto } \check{\iota} ; \bar{\iota} : \phi := 0 ; \check{\iota} : \epsilon \\ &\quad \text{in } (\mu, \dot{\phi}, \beta). \end{aligned}$$

The reason the approach encompasses the special treatment of Boolean expressions this nicely is that β 's and σ 's are so alike: Both are code abstracted over the *destination*; in the former case the destination is a destination register, in the latter it is a pair of destinations in the program.

With this way of translating Boolean expressions, the example from above will generate the code in figure 37 (ii). A more sophisticated example that shows the generality of the solution is the SML source language expression

```

if (b orelse (case d of
      C1 _ => true
     | C2 _ => false
     | C3 => true))
andalso i<j
then 3 else 666

```

It is translated to the following code (after appropriate removal of superfluous jumps (section 9.1)):

```

    if  $\phi_b = 1$  then  $\iota_3$  else  $\iota_1$  ;           (assuming 1 represents true)
 $\iota_1$  :  $\phi_k := m[\phi_d + 0]$  ;                 (fetch constructor)
    if  $\phi_k = \llbracket C1 \rrbracket_{C \rightarrow I}$  then  $\iota_3$  else  $\iota_2$  ;
 $\iota_2$  : if  $\phi_k = \llbracket C2 \rrbracket_{C \rightarrow I}$  then  $\iota_5$  else  $\iota_3$  ;
 $\iota_3$  : if  $\phi_i \leq \phi_j$  then  $\iota_4$  else  $\iota_5$  ;
 $\iota_4$  :  $\phi := 3$  ; goto  $\iota_6$  ;
 $\iota_5$  :  $\phi := 666$  ;
 $\iota_6$  :  $\epsilon$ .

```

Comparison with other work

Translating Boolean expressions to “short-circuit code” is a commonplace way of implementing them. The way we do it is closely related to Reynolds’ (1995). His “selectors” are passed a pair of continuations instead of a pair of labels, but would probably in an implementation also use labels (to avoid uncontrollable code duplication). Unlike ours, his source language restricts the set of language constructs that can have Boolean type. In particular, he does not allow an if-expression as e_0 of **if** e_0 **then** e_1 **else** e_2 . Consequently, he cannot treat **andalso** and **orelse** as derived forms of nested if-expressions; they must be treated explicitly.

Like we do, the S-1 Lisp compiler of Brooks et al. (1982) achieves the “right” short-circuit translation of Boolean expressions without treating **andalso** and **orelse** explicitly. They also regard these as derived forms of if-expressions. They do not let the context pass labels (or continuations) to the if-expression. Instead they rely on preceding transformations on the source language to transform an if-expression into one that will be translated to the right code. An if-expression occurring in e_0 of **if** e_0 **then** e_1 **else** e_2 will be “pushed into” the branches e_1 and e_2 , and that is the essential transformation for the short-circuiting translation. Other transformations then take care of the rest. (E.g., constant propagation handles the **true** introduced by the derived form of **orelse**.) (According to them, this way of translating nested if’s first appeared in (Steele, 1977).) The resulting code should be the same as ours.

SML/NJ’s “Boolean idiom simplification” (Appel, 1992,) accomplishes the same efficient translation for primitive Boolean expressions. E.g., **if** $a < b$ **then** e_1 **else** e_2 is translated to the same code by us and SML/NJ, but **if** a **andalso** b **then** e_1 **else** e_2 would not be translated efficiently by SML/NJ. The existing back end for the ML Kit does the same “Boolean idiom simplification”, but on a simple, RISC-like source language (Elsman and Hallenberg, 1995).

8.9 Function application

Passing parameters to functions

In this sub-section, we discuss how to pass parameters to functions. This information is part of the linking convention. How the linking convention is decided is dealt with in the next sub-section.

At function applications, $f\vec{p}\vec{e}_2$ and $e_1\vec{e}_2$, a closure, a return pointer, at least one argument, and zero or more region parameters must be passed to the applied function.

When there are more parameters than registers, some parameters are passed on the stack. In this case, it might be better to pass, e.g., parameters that are not always used by the callee, on the stack. However, since the situation where the number of parameters exceeds the number of available registers is rare, we will not invest any energy in doing this in a smart way. The space reserved on the stack for parameters is called a *frame*.

Evaluating an argument will probably destroy more registers than evaluating a region parameter. Each time a parameter that is to be passed in a register is evaluated, the number of free registers decreases by one. Therefore arguments are evaluated first, so as few registers as possible will be occupied.

Generating code for a function application becomes more difficult if the closure and the return label are eligible to be passed on the stack. To keep things simple, we will always pass them in registers.

As an example of how parameters are passed to a function, consider the application $e_0\langle e_1, e_2, e_3, e_4, e_5 \rangle$, and assume that the calling convention dictates that e_1 , e_3 , and e_5 must be passed on the stack, while e_2 and e_4 are passed in ϕ_2 and ϕ_4 , respectively. Then the situation just before the jump to the function will be as in this figure:

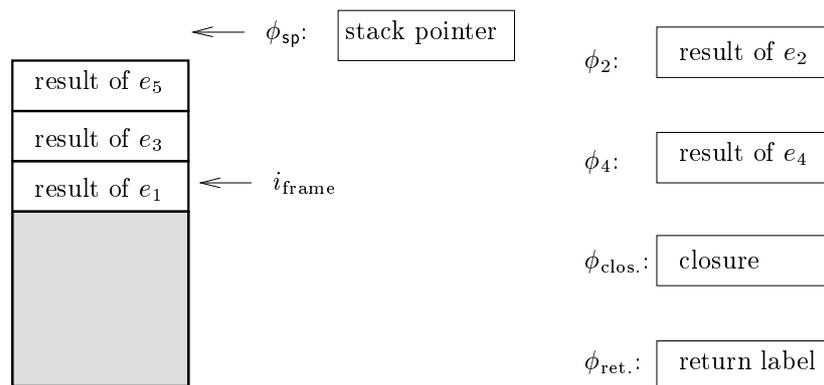


Fig. 38. *Passing parameters to functions.* i_{frame} points to the frame; e_1 is passed in the frame at offset 0, e_3 at offset 1, and e_5 at offset 2; e_2 and e_4 are passed in ϕ_2 and ϕ_4 ; the closure (the result of e_0) is passed in $\phi_{\text{clos.}}$, and the return label is passed in $\phi_{\text{ret.}}$.

Deciding the linking convention

A linking convention comprises the necessary information to generate code for an application, and for the entry and exit code for a function. This section describes what a linking convention is and when it is decided.

A *parameter convention* is a

$$\pi \in \Pi = I \cup \Phi.$$

If $\pi = i$, the parameter must be passed on the stack at offset i in the frame. If $\pi = \phi$, the parameter must be passed in register ϕ .

A *calling convention* is a

$$\mathcal{C} \in \mathcal{C} = (\Phi \times \Pi^* \times \Phi) \cup \{-cc\}.$$

If $\mathcal{C} = -cc$, we say the calling convention is undecided. The other possibility is explained below.

A *returning convention* is an

$$\mathcal{R} \in \mathcal{R} = \Phi \cup \{-rc\}.$$

If $\mathcal{R} = -rc$, we say the returning convention is undecided. The other possibility is explained below.

A *linking convention* is a calling convention and a returning convention:

$$\mathcal{L} \in \mathcal{L} = \mathcal{C} \times \mathcal{R}.$$

The linking convention is divided in a calling and a returning convention because these are decided at different points in the algorithm.

Linking conventions are associated with equivalence classes λ^\equiv of λ 's. Define the linking convention for λ to be the linking convention for the λ^\equiv that contains λ . Define the linking convention for an application $e_1 \lambda e_2$ to be the linking convention for the λ^\equiv that contains λ .

If the calling convention for λ is

$$\mathcal{C} = (\phi_{\text{clos.}}, (\pi_1, \dots, \pi_n, \pi_{1+n}, \dots, \pi_{k+n}), \phi_{\text{ret.}}),$$

with $n \geq 1$ and $k \geq 0$, λ expects its closure, **clos**, in $\phi_{\text{clos.}}$, the return label, **ret**, in $\phi_{\text{ret.}}$, its normal argument y_i in register/frame position π_i , and any region argument ρ_j in register/frame position π_{j+n} .

If the returning convention for λ is $\mathcal{R} = \phi_{\text{res.}}$, λ will return its result in $\phi_{\text{res.}}$.

The calling convention for an equivalence class, λ^\equiv , is decided the first time it is needed, i.e., the first time

1. a $\lambda \in \lambda^\equiv$ is processed by $\llbracket \cdot \rrbracket_{\text{donode}}$, or
2. code is generated for an application where a $\lambda \in \lambda^\equiv$ may be applied.

The second case can only occur at potentially recursive applications: at an application which is not potentially recursive, all λ 's that may be applied will already have been processed, because the call graph is processed bottom-up.

In the first case, we choose a register for each parameter the same way we choose a register for a value that is defined during the register allocation: we use $\llbracket \cdot \rrbracket_{\text{def}}$.

In the second case, there are different approaches to deciding the calling convention. For instance, if any of the values that are passed happen to reside in registers at the application, we could decide the calling convention such that they could remain in their registers. To keep things simple, we brutally choose arbitrary registers for the parameters.

Analogous to the calling convention, the returning convention for an equivalence class is decided the first time it is needed.

Translation of function application

At an application, for each parameter, we must generate some code that computes the parameter and places the result according to the parameter convention for that parameter.

We factor the similarities of generating code for region-polymorphic and normal function application into the meta-function *app* :

$$\begin{aligned} \llbracket \mathbf{f} \vec{\rho} \vec{e}_2 \rrbracket_{\text{ra}} \mu &= \text{app } \mathbf{f} \vec{e}_2 \vec{\rho} r \lambda(cc \lambda \mu)(rc \lambda \mu) \mu \\ \llbracket e_1 \vec{\lambda} \vec{e}_2 \rrbracket_{\text{ra}} \mu &= \text{app } e_1 \vec{e}_2 \llbracket r \lambda(cc \lambda \mu)(rc \lambda \mu) \mu \rrbracket. \end{aligned}$$

The first argument $g \in E \cup F$ of *app* is something that evaluates to a closure (\mathbf{f} or e_1). The third argument is a tuple of actual region variables ($\vec{\rho}$ if it is a region polymorphic function application; $\llbracket \cdot \rrbracket$ if it is not). Other information needed to process an application is μ , the recursiveness annotation r , the functions λ that may be applied, the calling and returning conventions, obtained with $cc \lambda \mu$ and $rc \lambda \mu$, respectively. (Remember the linking convention is part of the inter-procedural information, η , in ε in μ .)

We now develop *app*.

If the calling convention is not decided when *app* is called (i.e., $\mathcal{C} = -_{cc}$), we decide it now and call *app* again:

$$\begin{aligned} \text{app } g \vec{e} \vec{\rho} r \lambda -_{cc} \mathcal{R} \mu &= \\ \text{let } \mathcal{C} &= (\phi_1, (0, \dots, k+n-(N+1), \phi_2, \dots, \phi_{N+1}), \phi_N) \\ \text{in } \text{app } g \vec{e} \vec{\rho} r \lambda \mathcal{C} \mathcal{R} &(\text{set-cc } \lambda \mathcal{C} \mu), \end{aligned}$$

assuming $\vec{e} = \langle e_1, \dots, e_n \rangle$ and $\vec{\rho} = [\rho_1, \dots, \rho_k]$.

There are $k+n+2$ parameters. Only the first N get to be passed in a register, where N must be at least one less than the number of registers (for one register is needed by the code to call a function, p. 164). To keep things simple, the registers ϕ_1, \dots, ϕ_N are arbitrarily chosen. The closure is passed in ϕ_1 , the return label in ϕ_N , etc. Any remaining parameters are passed on the stack (in frame offsets 0 through $k+n-(N+1)$). The expression

$set-cc \lambda \mathcal{C} \mu$ yields a μ that records that the calling convention for λ_{cur} is \mathcal{C} . (It is, of course, the inter-procedural environment component, η , of μ which $set-cc$ updates.)

If the returning convention is not decided yet (i.e., $\mathcal{R} = -rc$), we decide it now. We (rather arbitrarily) choose to put **ret** in $\phi_{clos.}$:

$$\begin{aligned} app \ g \ \vec{e} \ \vec{r} \ \lambda \ \mathcal{C} \ -rc \ \mu &= \\ app \ g \ \vec{e} \ \vec{r} \ \lambda \ \mathcal{C} \ \phi_{clos.} (set-rc \ \lambda \ \phi_{clos.} \ \mu), \end{aligned}$$

where $\phi_{clos.}$ is from $\mathcal{C} = (\phi_{clos.}, (\pi_1, \dots, \pi_n, \pi_{1+n}, \dots, \pi_{k+n}), \phi_{ret.})$ and $set-rc$ is analogous to $set-cc$.

Now we come to the case where both the calling and the returning conventions have been decided when app is called. Look at the definition of app in its entirety on p. 162.

At run-time, the first thing that must be evaluated is the closure, g :

$$\begin{aligned} (\mu, \overset{\dagger}{\phi}_{clos.}, \overset{\dagger}{\beta}_{clos.}) &= \llbracket g \rrbracket_{ra-clos} \phi_{clos.} \omega \mu \\ \mu &= move \ \overset{\dagger}{\phi}_{clos.} \ \phi_{clos.} \ \mu \\ \mu &= \mu \ \dagger_t \ \phi_{clos.} \end{aligned}$$

The function $\llbracket \cdot \rrbracket_{ra-clos}$ to translate a g is explained below. To record in μ that the result of g will be moved into $\phi_{clos.}$ we use $move \ \overset{\dagger}{\phi}_{clos.} \ \phi_{clos.} \ \mu$. $\mu \ \dagger_t \ \phi_{clos.}$ introduces the value in $\phi_{clos.}$ as a temporary, and must be matched by a $kill-tmp \ \mu$ later on.

The arguments e_1, \dots, e_n and ρ_1, \dots, ρ_k are translated using $\llbracket \cdot \rrbracket_{ra-arg}$. Code, θ_i , is generated for each of e_i :

$$\begin{aligned} (\mu, \theta_1) &= \llbracket e_1 \rrbracket_{ra-arg} \pi_1 \omega'_1 \mu \\ &\vdots \\ (\mu, \theta_n) &= \llbracket e_n \rrbracket_{ra-arg} \pi_n \omega'_n \mu, \end{aligned}$$

where ω'_i is the ω after e_i . Each θ_i is the code to evaluate e_i and put the result where the parameter convention π_i says (i.e., either in some register or the frame). (A θ is a ζ abstracted over the position on the stack of the frame, i.e., $\theta \in I \rightarrow \mathbb{Z}$. This abstraction is necessary, because the offset from the stack pointer to the frame is not known at the time θ is produced, and θ may need it to put the parameter in the frame.)

After that, θ 's are generated for the region arguments, also using $\llbracket \cdot \rrbracket_{ra-arg}$:

$$\begin{aligned} (\mu, \theta_{1+n}) &= \llbracket \rho_1 \rrbracket_{ra-arg} \pi_{1+n} \omega'_n \mu \\ &\vdots \\ (\mu, \theta_{k+n}) &= \llbracket \rho_k \rrbracket_{ra-arg} \pi_{k+n} \omega'_n \mu. \end{aligned}$$

There is no ω -annotation on the ρ_i 's, so we use ω'_n , the ω after e_n .

Next in the code for an application, the return label is loaded into $\phi_{ret.}$. This must be recorded in μ :

$$\mu = \text{wipe } \phi_{\text{ret.}} \mu$$

If the parameter that e_1 evaluates to is put in a register that e_2 destroys, then that register must be saved across e_2 . Likewise if e_3 destroys the result from e_1 or e_2 , and so on. We deal with this the standard way: The registers containing parameters are temporaries, and $\llbracket \cdot \rrbracket_{\text{ra-arg}}$ records this in μ . After all arguments have been translated, we must check for each parameter in a register whether this register should be saved across the rest of the arguments. We introduce *kill-arg* to do this and give a preserver, p_i , for each parameter:

$$\begin{aligned} (\mu, p_{k+n}) &= \text{kill-arg } \pi_{k+n} \mu \\ &\vdots \\ (\mu, p_1) &= \text{kill-arg } \pi_1 \mu \\ (\mu, p_{\text{clos.}}) &= \text{kill-arg } \phi_{\text{clos.}} \mu \end{aligned}$$

We now generate code to jump to the applied function:

$$(\mu, \kappa_{\text{goto}}) = \text{goto } \lambda \phi_{\text{clos.}} \{ \pi_1, \dots, \pi_{k+n}, \phi_{\text{clos.}}, \phi_{\text{ret.}} \} \omega' \mu,$$

where $\text{goto } \lambda \phi_{\text{clos.}} \hat{\phi} \omega' \mu$ returns code to jump to the function when the set of functions that may be applied is λ , the closure is passed in $\phi_{\text{clos.}}$, and $\hat{\phi}$ is a set of registers that must not be destroyed by the code to jump.

We are now at the point where the applied function returns. We must *wipe* the registers in μ that have been destroyed by the callee, i.e., the set *destroys* $\lambda \mu$. Moreover, if the application is potentially recursive, we must record in μ that all values have been evicted from their registers, such that all values that are used after the application must be reloaded (p. 102). In other words, when $r = \circ$ we must *wipe* the set $\{ \phi \mid \mu^d \phi \neq \square \}$:

$$\begin{aligned} \mu &= \text{wipe } (\text{destroys } \lambda \mu \cup \\ &\quad \text{if } r = \circ \text{ then } \{ \phi \mid \mu^d \phi \neq \square \} \text{ else } \emptyset) \mu. \end{aligned}$$

The space that must be reserved on the stack for the frame is simply the number of integers among the parameter conventions:

$$j = |\{ \pi_1, \dots, \pi_{k+n} \} \cap I|.$$

Finally, the code is generated:

$$\begin{aligned} \beta &= \lambda \phi. \lambda \varsigma. \langle \phi_{\text{sp}} := \phi_{\text{sp}} + \llbracket j \rrbracket_{I \rightarrow I} \rangle ; \\ &\quad (\beta_{\text{clos.}} \phi_{\text{clos.}} ; \\ &\quad \quad p_{\text{clos.}} (\theta_1 \varsigma^{\text{p}} ; \\ &\quad \quad \quad p_1 (\theta_2 \varsigma^{\text{p}} ; \\ &\quad \quad \quad \quad \vdots \\ &\quad \quad \quad \quad p_{k+n+1} (\theta_{k+n} \varsigma^{\text{p}}) \dots)) \\ &\quad \quad \quad (\varsigma^{\text{e}}, \varsigma^{\text{p}} + j) ; \\ &\quad \phi_{\text{ret.}} := \iota ; \kappa_{\text{goto}} ; \\ &\quad \iota : \langle \phi := \phi_{\text{res.}} \rangle. \end{aligned}$$

$\langle \phi_{\text{sp}} := \phi_{\text{sp}} + \llbracket j \rrbracket_{I \rightarrow I} \rangle$ reserves memory on the stack for for the frame. $\beta_{\text{clos.}} \phi_{\text{clos.}}$ evaluates the closure and puts it in $\phi_{\text{clos.}}$. $p_{\text{clos.}}$ preserves $\phi_{\text{clos.}}$ across all the following code. $\theta_1 \zeta^{\text{P}}$ evaluates the first argument, and p_1 preserves it across all the following code; and so on. The stack offset of the frame, ζ^{P} , is passed to all θ_i 's. The ζ resulting from all this is applied to the stack shape $(\zeta^{\text{e}}, \zeta^{\text{P}} + j)$, reflecting that a frame of j words has been put on the stack.

Here *app* is in its entirety:

$$\begin{aligned}
\text{app } g \quad \langle e_1, \dots, e_n \rangle [\rho_1, \dots, \rho_k] \quad r \lambda \\
(\phi_{\text{clos.}}, (\pi_1, \dots, \pi_n, \pi_{1+n}, \dots, \pi_{k+n}), \phi_{\text{ret.}}) \phi_{\text{res.}} \mu &= \\
\text{let } (\mu, \dot{\phi}_{\text{clos.}}, \beta_{\text{clos.}}) = \llbracket g \rrbracket_{\text{ra-clos}} \phi_{\text{clos.}} \omega \mu & \\
\mu = \text{move } \dot{\phi}_{\text{clos.}} \phi_{\text{clos.}} \mu & \\
\mu = \mu \uparrow_t \phi_{\text{clos.}} & \\
(\mu, \theta_1) = \llbracket e_1 \rrbracket_{\text{ra-arg}} \pi_1 \quad \omega_1' \mu & \\
\vdots & \\
(\mu, \theta_n) = \llbracket e_n \rrbracket_{\text{ra-arg}} \pi_n \quad \omega_n' \mu & \\
(\mu, \theta_{1+n}) = \llbracket \rho_1 \rrbracket_{\text{ra-arg}} \pi_{1+n} \omega' \mu & \\
\vdots & \\
(\mu, \theta_{k+n}) = \llbracket \rho_k \rrbracket_{\text{ra-arg}} \pi_{k+n} \omega' \mu & \\
\mu &= \text{wipe } \phi_{\text{ret.}} \quad \mu \\
(\mu, p_{k+n}) = \text{kill-arg } \pi_{k+n} \quad \mu & \\
\vdots & \\
(\mu, p_1) = \text{kill-arg } \pi_1 \quad \mu & \\
(\mu, p_{\text{clos.}}) = \text{kill-arg } \phi_{\text{clos.}} \quad \mu & \\
(\mu, \kappa_{\text{goto}}) = \text{goto } \lambda \phi_{\text{clos.}} \{ \pi_1, \dots, \pi_{k+n}, \phi_{\text{clos.}}, \phi_{\text{ret.}} \} \omega' \mu & \\
\mu = \text{wipe } (\text{destroys } \lambda \mu \cup \text{if } r = \circ \text{ then } \{ \phi \mid \mu^d \phi \neq \square \} \text{ else } \emptyset) \mu & \\
j = |\{ \pi_1, \dots, \pi_{k+n} \} \cap I| & \\
\beta = \lambda \phi. \lambda (\zeta^{\text{e}}, \zeta^{\text{P}}). & \\
\langle \phi_{\text{sp}} := \phi_{\text{sp}} + \llbracket j \rrbracket_{I \rightarrow I} \rangle ; & \\
(\beta_{\text{clos.}} \phi_{\text{clos.}} ; p_{\text{clos.}} (\theta_1 \zeta^{\text{P}} ; p_1 (\theta_2 \zeta^{\text{P}} ; \dots p_{k+n} \perp 1 (\theta_{k+n} \zeta^{\text{P}}) \dots)) & \\
(\zeta^{\text{e}}, \zeta^{\text{P}} + j) ; & \\
\phi_{\text{ret.}} := \iota ; \kappa_{\text{goto}} ; & \\
\iota : \langle \phi := \phi_{\text{res.}} \rangle & \\
\text{in } (\mu, \phi_{\text{res.}}, \beta). &
\end{aligned}$$

We will now give the definitions of the auxiliary functions used above.

The function $\llbracket \cdot \rrbracket_{\text{ra-clos}}$ takes $g \in E \cup F$ as its argument. If g is an expression e , $\llbracket \cdot \rrbracket_{\text{ra-clos}}$ simply uses $\llbracket \cdot \rrbracket_{\text{ra}}$ to translate it:

$$\llbracket e \rrbracket_{\text{ra-clos}} \phi_{\text{clos.}} \omega \mu = \llbracket e \rrbracket_{\text{ra}} \mu.$$

If g is a sibling name, f , the closure is obtained by accessing f :

$$\llbracket f \rrbracket_{\text{ra-clos}} \phi_{\text{clos.}} \omega \mu = \llbracket f \rrbracket_{\text{use}} \emptyset \phi_{\text{clos.}} \omega \mu.$$

The function $\llbracket \cdot \rrbracket_{\text{ra-arg}}$ translates an argument $\alpha \in E \cup P$ to a θ . It uses $\llbracket \cdot \rrbracket_{\text{ra-arg-0}}$ to translate α to a β that evaluates α . If the parameter convention for α passed to $\llbracket \cdot \rrbracket_{\text{ra-arg}}$ is an integer $\pi = i$, code must be generated to place the result in the frame at offset i :

$$\begin{aligned} \llbracket \alpha \rrbracket_{\text{ra-arg}} i \omega \mu &= \text{let } (\mu, \dot{\phi}_\alpha, \beta_\alpha) = \llbracket \alpha \rrbracket_{\text{ra-arg-0}} \text{ -register } \omega \mu \\ &\quad (\mu, \phi_\alpha) = \text{tmp-tmp } \oslash \dot{\phi}_\alpha \omega \mu \\ &\quad \theta = \lambda i_{\text{frame}} \cdot \lambda(\zeta^e, \zeta^p). \\ &\quad \beta_\alpha \phi_\alpha(\zeta^e, \zeta^p) ; \text{m}[\phi_{\text{sp}} - \llbracket \zeta^p - i_{\text{frame}} - i \rrbracket_{I \rightarrow I}] := \phi_\alpha \\ &\text{in } (\mu, \theta), \end{aligned}$$

where i_{frame} is the position of the frame on the stack. Otherwise, code is generated to place the result in the supplied register $\pi = \phi_\alpha$:

$$\begin{aligned} \llbracket \alpha \rrbracket_{\text{ra-arg}} \phi_\alpha \omega \mu &= \text{let } (\mu, \dot{\phi}_\alpha, \beta_\alpha) = \llbracket \alpha \rrbracket_{\text{ra-arg-0}} \phi_\alpha \omega \mu \\ &\quad \mu = \text{move } \dot{\phi}_\alpha \phi_\alpha \mu \\ &\quad \mu = \mu +_i \phi_\alpha \\ &\quad \theta = \lambda i_{\text{frame}} \cdot \beta_\alpha \phi_\alpha \\ &\text{in } (\mu, \theta). \end{aligned}$$

The function $\llbracket \alpha \rrbracket_{\text{ra-arg-0}}$ works differently according to the form of α :

$$\begin{aligned} \llbracket \rho \rrbracket_{\text{ra-arg-0}} \dot{\phi} \omega \mu &= \llbracket \rho \rrbracket_{\text{use}} \oslash \dot{\phi} \omega \mu \\ \llbracket e_1 \rrbracket_{\text{ra-arg-0}} \dot{\phi} \omega \mu &= \llbracket e_1 \rrbracket_{\text{ra}} \mu. \end{aligned}$$

The function *kill-arg* takes a parameter convention π as its argument. If π is a register ϕ , *kill-arg* $\pi \mu$ works like *kill-tmp* μ and returns a preserver p of ϕ :

$$\text{kill-arg } \phi \mu = \text{kill-tmp } \mu.$$

If π is some i , i.e., the parameter is passed in the frame, there is no register to preserve, and *kill-arg* $\pi \mu$ yields the preserver that does nothing (p. 138):

$$\text{kill-arg } i \mu = (\mu, \text{don't}).$$

Finally, *goto* $\lambda \phi_{\text{clos}} \hat{\phi} \omega' \mu$ generates the code to jump to the function being applied. If only one λ may be applied, i.e. if λ is the singleton $\{\lambda\}$, the code to jump is simply a goto to the label for that λ :

$$\text{goto } \{\lambda\} \phi_{\text{clos}} \hat{\phi} \omega' \mu = (\mu, \text{goto } \llbracket \lambda \rrbracket_{\Lambda \rightarrow I}),$$

where $\llbracket \cdot \rrbracket_{\Lambda \rightarrow I} \in \Lambda \rightarrow I$ is a function to give a unique label to a λ . (λ 's can always be distinguished from each other because a λ binds a variable and all binding occurrences of variables are distinct (p. 19).)

If more than one λ may be applied, i.e., λ is not a singleton, we must, at run-time, fetch the destination label from the closure, and a register ϕ_t is needed to hold it temporarily:

$$\text{goto } \lambda\phi_{\text{clos.}}\hat{\phi}\omega'\mu = \text{let } (\mu, \phi_t) = \text{tmp-tmp } \hat{\phi}\text{-register } \omega'\mu \\ \text{in } (\mu, \phi_t := \mathbf{m}[\phi_{\text{clos.}} + 0] ; \text{goto } \phi_t).$$

The register ϕ_t must not be chosen amongst any registers used for parameters. This is why the $\hat{\phi}$ -argument to *goto* is needed. It places a restriction on the number of registers that are used for passing parameters: One register must be left for ϕ_t . This is the reason we only used N registers instead of simply all, when deciding the calling convention above (p. 159).

8.10 Exceptions

Concrete code for `raise e_1 , e_1 handle $a \Rightarrow e_2$, and $a \Rightarrow e_2$`

What the code to raise and handle exceptions should be was decided in section 4.8. That code did not specify how the global variable \mathbf{h} should be implemented. This we decide now, and then we can give the concrete code for the constructs `raise e_1` and `e_1 handle $a \Rightarrow e_2$` and for a handler `$a \Rightarrow e_2$` . After that, we discuss how to make the register allocation for these constructs. The main problem is how to deal with the rather complicated control flow that exceptions entail.

The code to raise an exception is (p. 50)

$$\phi := \boxed{\text{code to evaluate } \mathbf{raise } e_1} =$$

$\phi_{\text{raised}} :=$	<code>code to evaluate e_1</code>	;	
	<code>endregions \mathbf{h}</code>		}
	<code>$\phi_{\text{sp}} := \mathbf{h}$</code>		
	<code>pop \mathbf{h}</code>		}
	<code>pop ϕ'</code>		
	<code>goto ϕ'</code>		

Allocating some register to hold the current handler \mathbf{h} would reduce the number of available registers, and \mathbf{h} will probably not be used often in many programs. Therefore we prefer to keep \mathbf{h} in a memory cell.

Each global variable in memory is kept at a specific offset from a reserved register ϕ_{dp} . (Using a register for this purpose is necessary in our language K, for addresses can only be given relative to a register. We could have included in the language instructions “ $\phi := \mathbf{m}[\iota]$ ” and “ $\mathbf{m}[\iota] := \phi$ ” to load from and store into an absolute address. Actually, in our concrete architecture, PA-RISC, all addresses are also given relative to registers, and the operating system demands that a reserved register always points to a global data space.) Assuming \mathbf{h} 's offset from ϕ_{dp} is $\iota_{\mathbf{h}}$, the code is

$$\begin{array}{l}
\phi := \boxed{\text{code to evaluate raise } e_1} = \\
\begin{array}{l}
\phi_{\text{raised}} := \boxed{\text{code to evaluate } e_1} ; \\
\phi_{h1} := m[\phi_{dp} + \iota_h] ; \text{endregions } \phi_{h1} ; \\
\phi_{sp} := m[\phi_{dp} + \iota_h] ; \\
\text{pop } \phi_{h2} ; m[\phi_{dp} + \iota_h] := \phi_{h2} ; \\
\text{pop } \phi' ; \\
\text{goto } \phi'
\end{array}
\left| \begin{array}{l}
\text{endregions } h \\
\phi_{sp} := h \\
\text{pop } h \\
\text{pop label of handler} \\
\text{jump.}
\end{array} \right.
\end{array}$$

In that code, the same actual register can be used for both ϕ_{h1} , ϕ_{h2} , and ϕ' , as long as it is different from ϕ_{raised} . If we always use some fixed register, ϕ_{raise} , we can define the instruction `raise` that raises an exception when the exception value is in ϕ_{raised} :

$$\begin{array}{l}
\text{raise} = \begin{array}{l}
\phi_{\text{raise}} := m[\phi_{dp} + \iota_h] ; \text{endregions } \phi_{\text{raise}} ; \\
\phi_{sp} := m[\phi_{dp} + \iota_h] ; \\
\text{pop } \phi_{\text{raise}} ; m[\phi_{dp} + \iota_h] := \phi_{\text{raise}} ; \\
\text{pop } \phi_{\text{raise}} ; \\
\text{goto } \phi_{\text{raise}}
\end{array}
\left| \begin{array}{l}
\text{endregions } h \\
\phi_{sp} := h \\
\text{pop } h \\
\text{pop label of handler} \\
\text{jump}
\end{array} \right.
\end{array}$$

where $\phi_{\text{raise}} \neq \phi_{\text{raised}}$ (i.e., `raise` is an instruction whose semantics is specified by defining it in terms of other instructions, as, e.g., `push ϕ` .)

Then

$$\begin{array}{l}
\phi := \boxed{\text{code to evaluate raise } e_1} = \\
\phi_{\text{raised}} := \boxed{\text{code to evaluate } e_1} ; \text{raise.}
\end{array}$$

The instruction `raise` includes (the equivalent of) an `endregions`-instruction and destroys ϕ_{raise} , and hence, the set of registers destroyed by `raise` is $\hat{\phi}_{\text{raise}} = \{\phi_{\text{raise}}\} \cup \hat{\phi}_{\text{endregions}}$.

With the decision to keep the global variable h in a memory cell, the code for a `handle`-expression is (p. 50):

$$\begin{array}{l}
\phi := \boxed{\text{code to evaluate } e_1 \text{ handle } a \Rightarrow e_2} = \\
\begin{array}{l}
\phi' := \iota_{a \Rightarrow e_2} ; \text{push } \phi' ; \\
\phi_{h1} := m[\phi_{dp} + \iota_h] ; \text{push } \phi_{h1} ; \\
m[\phi_{dp} + \iota_h] := \phi_{sp} ; \\
\phi := \boxed{\text{code to evaluate } e_1} ; \\
\text{pop } \phi_{h2} ; m[\phi_{dp} + \iota_h] := \phi_{h2} ; \\
\text{pop}
\end{array}
\left| \begin{array}{l}
\text{push label of handler} \\
\text{save } h \\
h := \phi_{sp} \\
\text{restore } h \\
\text{discard label of handler.}
\end{array} \right.
\end{array}$$

The code for a handler is (p. 53)

$$\begin{aligned}
 \boxed{\text{code for } a \Rightarrow e_2} &= \\
 \phi_a &:= \boxed{\text{code to access } a} ; \\
 \phi_{a?} &:= m[\phi_{\text{raised}} + 0] ; \\
 \text{if } \phi_a = \phi_{a?} &\text{ then } \iota \text{ else } \bar{\iota} ; \quad \iota: \phi := \boxed{\text{code to evaluate } e_2} ; \text{ goto } \check{\iota} ; \\
 &\quad \bar{\iota}: \text{ raise ,}
 \end{aligned}$$

where ϕ is the ϕ in $\phi := \boxed{\text{code to evaluate } e_1 \text{ handle } a \Rightarrow e_2}$, and $\check{\iota}$ is the meeting label of the code for e_1 and the code for the handler, i.e., $\check{\iota}$ labels the code after the code for $e_1 \text{ handle } a \Rightarrow e_2$.

Register allocation of `raise e_1 and $e_1 \text{ handle } a \Rightarrow e_2$`

Now the register allocation of the exception constructs is discussed. The problem is the flow of the descriptors, δ , in the function $\llbracket \cdot \rrbracket_{\text{ra}}$ in the presence of the irregular control flow caused by exceptions. Hitherto, the flow of δ 's has not been a problem because the possible control flow has been so simple: either linear, or forking and then meeting again in `if`-expressions.

With exceptions, control flow is less structured and there is no single, clear answer to how δ 's should flow. In particular,

- what should the δ flowing out of `$e_1 \text{ handle } a \Rightarrow e_2$` be? At run-time, control may simply flow straight through the code for e_1 , but an exception may also be raised, causing control to flow through the code for the handler $a \Rightarrow e_2$.
- What should the δ flowing into the handler be? An exception may be raised from different places in e_1 , and it may even be raised in another function called by e_1 .

The following is somewhat involved and specific to our use of δ 's. Still, the discussions touch upon issues that may be of general interest, e.g., how to deal with exception control flow, and what can be assumed at different program points when exception control flow is in the picture.

We require that the claims made by a δ at some point in the translation are always correct at run-time at the corresponding program point; e.g., if a δ says that ϕ contains v , then ϕ must always contain v at the corresponding program point in the code no matter how control flowed to that program point. Giving a formal definition, let alone proving that the claims made by the δ 's of our algorithm are always correct, is beyond the scope of this report.

It has not before been necessary to discuss the requirements on the δ 's, as they were automatically satisfied because the flow of δ 's mimicked the control flow.

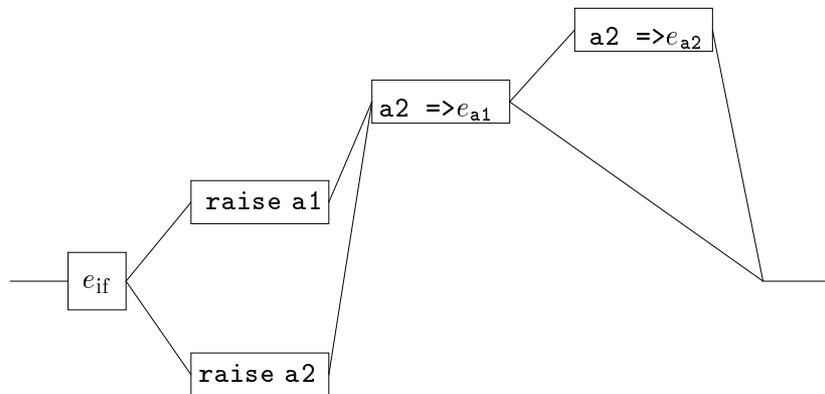
Control flow with exceptions

Ideally, the δ 's should flow as control flows. This would ensure that the δ 's always contained correct information. This section discusses exception control flow; the next section discusses how to make the corresponding flow of δ 's.

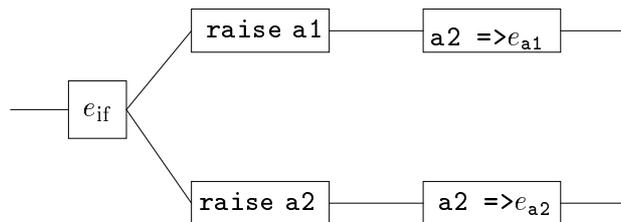
It is generally undecidable what handler will *handle* a raise from a given **raise-expression**, but remember that it will always be the handler on the top of the stack of handlers that *deals with* a raised exception (p. 45). In the following sub-sections we shall refer to the construct e_1 **handle** $a \Rightarrow e_2$. The handler $a \Rightarrow e_2$ will deal with all raises from **raise-expressions** in e_1 that are not within some other **handle-expression** in e_1 (and it will also deal with the implicit raise that is in any **handle-expression** within e_1). I.e., in

```
((if eif then raise a1 at r1
  else raise a2 at r2) handle a1 => ea1)
  handle a2 => ea2,
```

the inner handler, $a1 \Rightarrow e_{a1}$, deals with raises from both **raise a1 at r1** and **raise a2 at r2**, although it can only handle **a1**. The outer handler, $a2 \Rightarrow e_{a2}$, only deals with the raise implicit in the inner **handle-expression**, i.e., the re-raise done by the inner handler when it discovers that it cannot handle **a2**. Hence the control flow is



and not



Raises do not come solely from **raise-expressions** within e_1 , however; functions applied within e_1 may also raise an exception:

```

let rejs = λu.raise (a at r3) at r3
in
  (1+rejs 2+4) handle a => 5.

```

This we deal with by conservatively regarding every application as a potential raise. Alternatively, a more elaborate, inter-procedural analysis could be employed, but it would probably not be worth the effort and added complexity.

Mimicking the control flow with the flow of δ 's

A first try at computing the in-flowing δ to the register allocation of the handler $a \Rightarrow e_2$ could be to collect all out-flowing δ 's from the register allocation of expressions that may raise an exception (i.e. **raise**-expressions, applications, and other **handle**-expressions) in e_1 that are not within any other **handle**-expression in e_1 , and take the meet (\sqcap_δ , p. 150) of them. E.g., in the example above, if the δ 's returned by $\llbracket \text{raise } a1 \text{ at } r1 \rrbracket_{ra}$ and $\llbracket \text{raise } a2 \text{ at } r2 \rrbracket_{ra}$ are δ_1 and δ_2 , respectively, the in-flowing δ to the register allocation of the handler $a1 \Rightarrow e_{a1}$ would be $\delta_1 \sqcap_\delta \delta_2$. The in-flowing δ to the outer handler, $a2 \Rightarrow e_{a2}$, should be the δ flowing out from the inner handler, since the inner handler implicitly contains a raise which will be dealt with by the outer handler.

The resulting δ flowing into the handler will likely be very conservative because every application is regarded as a raise. Our main reason for not choosing this scheme is, however, that making the flow of δ 's more like the actual control flow when there are exceptions will complicate the algorithm for all constructs and not just the exception constructs.

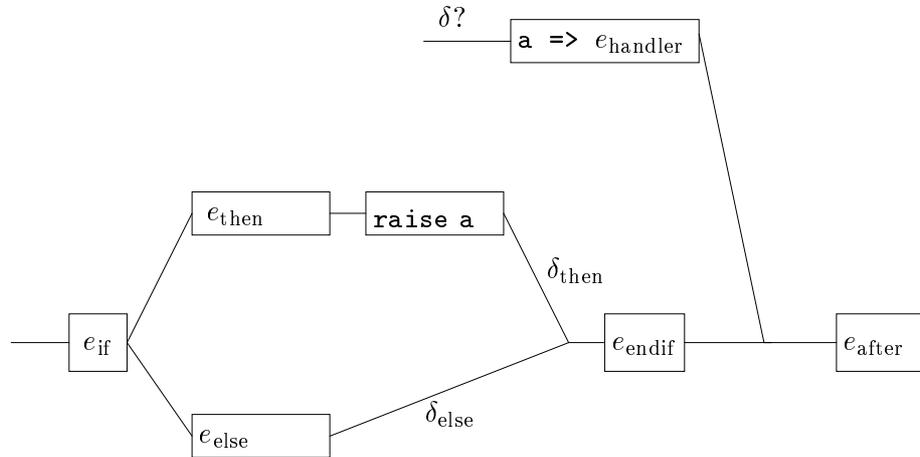
The second try at computing the δ flowing into the handler is more of a hack. It does not attempt to follow the exceptional control flow: We let the δ 's flow as if raises did not change control flow. The method is illustrated by this example

```

(((if eif then ethen + raise a
   else eelse) + eendif)
  handle a => ehandler) + eafter

```

where δ 's flow as follows



A δ flows from e_{then} to e_{endif} , although control could never flow that way. In effect, we have incorporated the normal and the exceptional control flow in the same δ , which, of course, makes the information in δ less precise. For instance, the δ flowing into e_{endif} , i.e., $\delta_{\text{then}} \sqcap \delta_{\text{else}}$, is a worse approximation than the δ_{then} it should have been if the flow of δ 's reflected the normal control flow better.

Also the implicit re-raise in a handler is ignored; thus only one δ flows out from e_1 `handle` $a \Rightarrow e_2$.

When exceptions are not raised, this flow of δ 's will give correct (although less accurate) δ 's.

What δ should flow into the register allocation of the handler $a \Rightarrow e_2$? Observe that the raise will come from somewhere within the code for e_1 . We can make a correct δ flow into $a \Rightarrow e_2$, by taking the δ flowing out from e_1 and making the registers destroyed by e_1 undefined (with *wipe*) in that δ . This gives a correct δ flowing into $a \Rightarrow e_2$ because at the entry to the handler code, a register will actually contain the value this δ says it contains no matter how control flowed to the handler code, for control can only flow from raises within e_1 , and they can only have destroyed registers that are also destroyed by the whole code of e_1 since they are part of e_1 .

Notice, however, that this only holds because expressions are *selfish* when saving registers: an expression saves a register only if it contains a value the expression needs and the register is destroyed by a sub-expression; an expression never saves a register for the benefit of its context.

If expressions were not selfish, an expression might save a register that was destroyed by a sub-expression, and the context of that expression would (rightly) believe that the register would not be destroyed. But with the method sketched above, the nearest enclosing handler would also believe that the register would not be destroyed, and this would be wrong, for an exception may be raised from within the sub-expression, where the register has been destroyed.

Notice that if we did not use the producer-saves placement of spill code strategy, or if we had functions with callee-save registers, the register saving would not be selfish.

A price is paid in this scheme for mixing the δ 's of the normal control flow and the exceptional control flow: The code for e_1 may be less efficient, because the δ 's give less accurate information. It is, regrettably, a price paid also when exceptions are not raised. Nevertheless, we have chosen this solution to avoid the complication of the first solution.

Furthermore, the δ flowing into the handler $a \Rightarrow e_2$ is less accurate: The values in all registers that are destroyed by e_1 are assumed to be undefined at the entry to the code for $a \Rightarrow e_2$. This means that values used by $a \Rightarrow e_2$ may have to be reloaded unnecessarily inside $a \Rightarrow e_2$. That is quite acceptable, as it will only occur when an exception is raised. Worse, the values that are thus reloaded unnecessarily may as a consequence have to be spilled, and this unnecessary spilling happens regardless of whether an exception is raised or not. This may, however, still be one of the minor problems, as handlers often use few values.

Finally, if the δ flowing into $a \Rightarrow e_2$ is unnecessarily conservative, the δ flowing out from $a \Rightarrow e_2$, and thus, the δ flowing out from e_1 **handle** $a \Rightarrow e_2$, will be unnecessarily conservative. Consequently, unnecessary saving of values may occur around e_1 **handle** $a \Rightarrow e_2$.

The δ flowing out from e_1 **handle** $a \Rightarrow e_2$ should be the meet of the δ flowing out from e_1 and the δ flowing out from $a \Rightarrow e_2$, corresponding to the meeting of the control flow from the code for e_1 and the code for $a \Rightarrow e_2$. In other words, in the δ flowing out from a **handle**-expression, it is conservatively assumed that the code for the handler was executed. It would be more in line with our intention to make the code for a **handle**-expression efficient at the price of making a raise (i.e., the handler code) less efficient if we assumed the opposite: that the handler code was not executed, and then imposed on the handler the duty of saving the registers it destroys. However, this strategy does not mesh nicely with the general selfish principle of the register allocation: that the code for each expression destroys whatever registers that suit it without saving them. To keep things simple, we therefore choose not to do it.

$$\begin{aligned}
\llbracket e_1 \text{ handle } a \Rightarrow e_2 \rrbracket_{\text{ra}}(\delta, \varepsilon) &= \\
\text{let } ((\delta, \varepsilon), \phi') &= \text{tmp-tmp } \emptyset \text{ -register } \omega_1(\delta, \varepsilon) \\
((\delta, \varepsilon), \dot{\phi}_1, \beta_1) &= \llbracket e_1 \rrbracket_{\text{ra}}(\delta, \varepsilon) \\
((\delta, \varepsilon), \phi_1) &= \text{tmp-tmp } \emptyset \dot{\phi}_1 \omega'_1(\delta, \varepsilon) \\
((\delta_1, \varepsilon), \phi'') &= \text{tmp-tmp } \{\phi_1\} \text{ -register } \omega'_1(\delta, \varepsilon) \\
(\delta, \varepsilon) &= \text{wipe } \varepsilon \dot{\phi}(\delta_1, \varepsilon) \\
((\delta, \varepsilon), \phi_a, \beta_a) &= \llbracket a \rrbracket_{\text{use}} \{\phi_{\text{raised}}\} \text{ -register } \omega_2(\delta, \varepsilon) \\
((\delta, \varepsilon), \phi_{a?}) &= \text{tmp-tmp } \{\phi_{\text{raised}}, \phi_a\} \text{ -register } \omega_2(\delta, \varepsilon) \\
(\delta_\iota, \dot{\phi}_2, \beta_2) &= \llbracket e_2 \rrbracket_{\text{ra}}(\delta, \varepsilon) \\
(\delta_{\bar{\iota}}, \varepsilon) &= \text{wipe } \dot{\phi}_{\text{raise}}(\delta, \varepsilon) \\
\delta_2 &= \delta_\iota \sqcap_\delta \delta_{\bar{\iota}} \\
\beta &= \lambda\phi. \lambda\varsigma. \\
&\phi' := \iota_{a \Rightarrow e_2} ; \text{push } \phi' ; \\
&\phi' := \mathbf{m}[\phi_{\text{dp}} + \iota_{\text{h}}] ; \text{push } \phi' ; \\
&\mathbf{m}[\phi_{\text{dp}} + \iota_{\text{h}}] := \phi_{\text{sp}} ; \\
&\beta_1 \phi_1(\varsigma^e, \varsigma^p + 2) ; \\
&\text{pop } \phi'' ; \mathbf{m}[\phi_{\text{dp}} + \iota_{\text{h}}] := \phi'' ; \\
&\text{pop} ; \langle \phi := \phi_1 \rangle ; \text{goto } \checkmark ; \\
&\iota_{a \Rightarrow e_2} : \beta_a \phi_a \varsigma ; \phi_{a?} := \mathbf{m}[\phi_{\text{raised}} + 0] ; \\
&\quad \text{if } \phi_a = \phi_{a?} \text{ then } \iota \text{ else } \bar{\iota} ; \quad \iota : \beta_2 \phi \varsigma ; \text{goto } \checkmark ; \\
&\quad \quad \quad \bar{\iota} : \text{raise} ; \\
&\checkmark : \epsilon \\
&\text{in } (\delta_1 \sqcap_\delta \delta_2, \phi_1, \beta).
\end{aligned}$$

where $\iota_{a \Rightarrow e_2}$ labels the handler code, and \checkmark is the meeting label of the code for e_1 and the code for the handler, i.e., \checkmark labels the code after the code for $e_1 \text{ handle } a \Rightarrow e_2$.

Notice that the stack for the code for e_1 is $\varsigma^p + 2$, reflecting that the handler has been pushed on the stack before the code for e_1 . Contrastingly, the stack for the code for e_2 is ς^p , because the handler element has been removed by `raise` before the handler code is called.

In the register allocation for the handler, ϕ_a and $\phi_{a?}$ must not be the same register as ϕ_{raised} , for then the handler would destroy the raised exception value.

Notice that it is ensured that the code for e_1 and the code for the handler put the result in the same register, ϕ .

The translation of a raise is:

$$\begin{aligned}
\llbracket \mathbf{raise} \ e_1 \rrbracket_{\text{ra}}(\delta, \varepsilon) &= \text{let } ((\delta, \varepsilon), \hat{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}}(\delta, \varepsilon) \\
&\quad (\delta, \varepsilon) = \mathit{move} \ \hat{\phi}_1 \phi_{\text{raised}}(\delta, \varepsilon) \\
&\quad (\delta, \varepsilon) = \mathit{wipe} \ \hat{\phi}_{\text{raise}}(\delta, \varepsilon) \\
&\quad \beta = \lambda\phi. \lambda\varsigma. \beta_1 \phi_{\text{raised}} \varsigma ; \mathbf{raise} \\
&\text{in } ((\delta, \varepsilon), \text{-register}, \beta).
\end{aligned}$$

Notice the δ which is returned incorporates the information that we want the δ flowing to the enclosing handler to receive.

Comparison

We have previously compared our way of implementing exceptions with that of the existing intermediate-code generator, COMPILE-LAMBDA (p. 53). Now we can compare the two with respect to register allocation aspects. COMPILE-LAMBDA pushes all registers containing live values just before a **handle**-expression. In comparison, we only save those registers that some approximation says may be destroyed by the **handle**-expression. As this approximation has its limitations, this might not make much difference. Our producer-saves saving strategy means that a value is only saved once and it is only reloaded when it is actually needed. (This is, of course, a feature of our producer-saves saving strategy and not something that has specifically to do with our register allocation of exceptions.) COMPILE-LAMBDA may push and pop the same value many times across **handle**-expressions although it is only used once.

Unlike COMPILE-LAMBDA, we do not build a closure for the handler every time e_1 **handle** $a \Rightarrow e_2$ is evaluated. On the other hand, in our scheme, if some value used by the handler resides in one of the registers that are destroyed by e_1 , its use in the handler will cause it to be spilled, and it will be saved on the stack by its producer and it will be fetched by the handler. This is similar to COMPILE-LAMBDA's use of a closure.

In practice, these differences are likely unimportant. This way of implementing exceptions has been chosen neither because we expected it to give a big gain in speed compared to COMPILE-LAMBDA nor because we wanted to squeeze out the last clock cycle (then we would have chosen a more sophisticated solution); mainly, it has been chosen because it was the nicest way we could see to assimilate exceptions into our register allocation algorithm.

8.11 Processing a call graph node, λ

This section discusses how to process a node, λ , in the call graph, i.e., it develops the function $\llbracket \cdot \rrbracket_{\text{donode}}$ (p. 64). From now on, λ_{cur} refers to the function (node) in the call graph currently being processed (i.e., the argument of $\llbracket \cdot \rrbracket_{\text{donode}}$) and e_0 refers to its body. Processing λ_{cur} chiefly consists of doing register allocation and code generation ($\llbracket \cdot \rrbracket_{\text{ra}}$) for the body of λ_{cur} .

The main problems are: First, the linking convention for λ_{cur} may or may not be decided yet. If it is not, we must decide it now, for we need it to generate code for λ_{cur} . (In technical terms, we need to know the calling convention to be able to set up the initial descriptor δ and stack shape ζ , and we need to know the returning convention, as it is the register that the code β for λ_{cur} should be applied to.) Second, it may be necessary to store some of the parameters of λ_{cur} , because they may be loaded in e_0 . In this respect, the parameters can be treated as if they were **let**-bound values. For example, e_0 of $f[\rho_1, \dots, \rho_k] \langle y_1, \dots, y_n \rangle = \mathcal{K}_{e_0}$ is treated approximately as if it were

```

let clos = “something” in
  let  $y_1$  = “something else” in
    ..
    let  $y_n$  = “something third” in
      let  $\rho_1$  = “something” in
        ..
        let  $\rho_k$  = “something” in
          let ret = “something”
            in  $e_0$ .

```

If, e.g., y_1 is loaded in e_0 , $\llbracket \lambda_{\text{cur}} \rrbracket_{\text{donode}}$ must ensure that it is accessible on the stack. If y_1 is passed to λ_{cur} on the stack, it will be accessible on the stack when it is loaded by the code for e_0 ; otherwise—if y_1 is passed in a register—the code for λ_{cur} must itself push y_1 .

The code for λ_{cur} has the form

```

 $\llbracket \lambda_{\text{cur}} \rrbracket_{\Lambda \rightarrow I}$  : push the parameters  $\{v_1, \dots, v_m\}$  that are
                           loaded in  $e_0$  and not passed on the stack ;
 $\phi :=$  code to evaluate  $e_0$  ;
 $\phi_{\text{ret}} :=$  code to access ret ;
                           remove the parameters from the stack ;
                           goto  $\phi_{\text{ret}}$ .

```

where $\llbracket \lambda_{\text{cur}} \rrbracket_{\Lambda \rightarrow I}$ is the unique label for λ_{cur} ; ϕ is the register λ_{cur} returns its result in according to the returning convention; and ϕ_{ret} is a temporarily used register. Before returning, the parameters on the stack are removed, including those that were passed on the stack.

The algorithm to process a function, λ_{cur} , is thus

0. perform the ω -analysis (section 8.1) on λ_{cur} ;
1. establish entry conditions:
 - \mathcal{C} = the calling convention (if it is undecided as yet, decide it now);
 - δ_0 = the initial descriptor according to \mathcal{C} (e.g., if **clos** is passed in register ϕ_1 , $\delta_0^d \phi_1 = \mathbf{clos}$);

- ς_0 = the initial stack shape according to \mathcal{C} (e.g., if y is passed on the stack at frame offset 2, $\varsigma_0^e y = 2$);
2. perform the register allocation of ε_0 ;
 3. establish the exit condition: ϕ = the returning convention (if it is undecided as yet, decide it now);
 4. perform the register allocation of a use of **ret**;
 5. generate the code sketched above.

More specifically, $\llbracket \lambda_{\text{cur}} \rrbracket_{\text{donode}}$ is as follows. Explanations are below.

$$\begin{aligned}
\llbracket \lambda_{\text{cur}} \rrbracket_{\text{donode}} \nu &= \\
\text{let } \lambda_{\text{cur}}^* &= \llbracket \lambda_{\text{cur}} \rrbracket_{\text{oa}} \nu^\eta && 0. \\
\varepsilon_0 &= (\nu, \lambda_{\text{cur}}^*) && 1. \\
((\delta_0, \varepsilon), \varsigma_0, \mathcal{C}) &= \text{entry } \varepsilon_0 \\
((\delta, \varepsilon), \check{\phi}, \beta) &= \llbracket \check{e}_0 \rrbracket_{\text{ra}} (\delta_0, \varepsilon) && 2. \\
(\phi_{\text{res.}}, \varepsilon) &= \text{decide-rc } \check{\phi} (\delta, \varepsilon) && 3. \\
((\delta, \varepsilon), \phi_{\text{ret}}, \beta_{\text{ret}}) &= \llbracket \text{ret} \rrbracket_{\text{use}} \{\phi_{\text{res.}}\} -_{\text{register}} \omega'_0 (\delta, \varepsilon) && 4. \\
\{v_1, \dots, v_m\} &= \delta^v \setminus \text{Dm } \varsigma_0^e && 5. \\
\kappa &= \llbracket \lambda_{\text{cur}} \rrbracket_{\Lambda \rightarrow \text{I}} : \\
&\quad \llbracket v_1 \rrbracket_{\text{push-arg.}} \mathcal{C} (\\
&\quad \quad \quad \vdots \\
&\quad \quad \quad \llbracket v_m \rrbracket_{\text{push-arg.}} \mathcal{C} (\\
&\quad \quad \quad \quad \llbracket \beta \phi_{\text{res.}} ; \beta_{\text{ret}} \phi_{\text{ret}} \rrbracket_{\text{zap}}) \cdots) \varsigma_0 ; \\
&\quad \text{goto } \phi_{\text{ret}} \\
&\text{in } (\kappa, \varepsilon^\nu).
\end{aligned}$$

Remember that the environment ν has the form $(\eta, \check{\phi})$, where η is the inter-procedural environment and $\check{\phi}$ is the set of registers destroyed by the current strongly connected component. The ω -analysis $\llbracket \cdot \rrbracket_{\text{oa}}$ needs the inter-procedural environment η .

The per-function environment ε , used during register allocation, consists of the environment ν and the current function: $\varepsilon = (\nu, \lambda_{\text{cur}})$.

The auxiliary function *entry* establishes the entry conditions by finding the calling convention \mathcal{C} , the initial descriptor δ_0 , and the initial stack shape ς_0 . If it was necessary to decide the calling convention, ε is updated to record the decision. This is described in detail below.

\check{e}_0 is the body of the ω -annotated version, λ_{cur}^* , of λ_{cur} .

In analogy with *entry*, the auxiliary *decide-rc* establishes the returning convention. This is described in detail below.

The return label is accessed with $\llbracket \text{ret} \rrbracket_{\text{use}} \{\phi_{\text{res.}}\} -_{\text{register}} \omega'_0 (\delta, \varepsilon)$. The register used for the return label must not be the same as the result register, $\phi_{\text{res.}}$; so $\phi_{\text{res.}}$ is forbidden. The ω after \check{e}_0 is ω'_0 .

After the register allocation, δ holds the set δ^v of values that will be loaded in the body. Of these values, those that are already on the stack

because they are passed on the stack (i.e., the parameters $\text{Dm } \zeta_0^e$ in the initial stack shape) need not be pushed. Hence, the values that must be pushed are $\{v_1, \dots, v_m\} = \delta^v \setminus \text{Dm } \zeta_0^e$.

The calling convention $\mathcal{C} \in V \xrightarrow{\perp} \Phi \cup I$ maps each parameter to its parameter convention: if $\mathcal{C}v = \phi$, v is passed in ϕ ; if $\mathcal{C}v = i$, v is passed on the stack at frame offset i . (When discussing how to translate an application, we used a different definition of calling convention; in this situation the present definition is more convenient. Clearly, it is possible to convert the two kinds of calling conventions to each other.)

The function $\llbracket v \rrbracket_{\text{push-arg.}} \mathcal{C} \in \mathbf{Z} \rightarrow \mathbf{Z}$ prepends to its argument, ζ , code to push v :

$$\llbracket v \rrbracket_{\text{push-arg.}} \mathcal{C} \zeta(\zeta^e, \zeta^p) = \text{push } \mathcal{C}v ; \zeta(\zeta^e + \{v \mapsto \zeta^p\}, \zeta^p + 1).$$

The function $\llbracket \cdot \rrbracket_{\text{zap}} \in \mathbf{Z} \rightarrow \mathbf{Z}$ appends after its argument, ζ , code to reset the stack pointer:

$$\llbracket \zeta \rrbracket_{\text{zap}} \varsigma = \zeta \varsigma ; \langle \phi_{\text{sp}} := \phi_{\text{sp}} - \llbracket \zeta^p \rrbracket_{I \rightarrow I} \rangle.$$

$\llbracket \lambda_{\text{cur.}} \rrbracket_{\text{donode}}$ must return a ν ; this is extracted from ε .

The details

Now *entry* and *decide-rc* are described.

$$\begin{aligned} \text{entry } \varepsilon &= \text{let } \delta_0^d = \lambda\phi. \text{if } \phi \in \varepsilon^{\check{\phi}} \text{ then } -_d \text{ else } \square \\ &\quad \delta_0 = (\emptyset, (), \delta_0^d) \\ &\quad (\mathcal{C}, \varepsilon) = \text{decide-cc}(\delta_0, \varepsilon) \\ &\quad \delta_0 = (\emptyset, (), \delta_0^d + \mathcal{C}^{\perp 1} |_{\Phi}) \\ &\quad \zeta_0^e = \mathcal{C} |_{\{v \mid \mathcal{C}v \in I\}} \\ &\quad \zeta_0^p = |\{ \mathcal{C}v \mid \mathcal{C}v \in I \}| \\ &\quad \varsigma_0 = (\zeta_0^e, \zeta_0^p) \\ &\quad \text{in } ((\delta_0, \varepsilon), \varsigma_0, \mathcal{C}). \end{aligned}$$

where ω_0 is the ω before e_0 .

This function first sets up the initial descriptor δ_0 . Remember ε contains an approximation $\varepsilon^{\check{\phi}}$ of the set of registers that will be destroyed anyway by the current strongly connected component (p. 72). We let the initial register descriptor, δ_0^d , map these registers to $-_d$, thereby encouraging the heuristic that chooses registers to use them first (p. 90). Other registers are mapped to \square . The two other components in δ_0 are initially: $\delta_0^v = \emptyset$ and $\delta_0^t = ()$ (“()” is an empty stack of temporaries).

Then *entry* calls *decide-cc* (described below) to either get the calling convention \mathcal{C} (from ε) or, if it is not decided yet, decide it (and update ε).

Then the register descriptor, δ_0^d , is updated according to the calling convention: if $\mathcal{C}v$ is ϕ , then $\delta_0^d \phi$ should be v . The initial stack environment ζ_0^e

is set to map the parameters that are passed on the stack to their offsets: if $\mathcal{C}v$ is i , then $\varsigma_0^e v$ should be i . The initial stack pointer, ς_0^p , is the number of parameters passed on the stack.

$$\begin{aligned}
\mathit{decide-cc}(\delta, \varepsilon) &= \text{if } cc \varepsilon^{\lambda_{\text{cur}}}. \varepsilon \neq -_{cc} \text{ then } (cc \varepsilon^{\lambda_{\text{cur}}}. \varepsilon, \varepsilon) \text{ else} \\
&\text{let } (v_1, \dots, v_j) = \mathit{params} \varepsilon^{\lambda_{\text{cur}}}. \\
&((\delta, \varepsilon), \phi_1) = \llbracket v_1 \rrbracket_{\text{def}} \emptyset -_{\text{register}} \omega_0(\delta, \varepsilon) \\
&((\delta, \varepsilon), \phi_2) = \llbracket v_2 \rrbracket_{\text{def}} \{\phi_1\} -_{\text{register}} \omega_0(\delta, \varepsilon) \\
&\quad \vdots \\
&((\delta, \varepsilon), \phi_N) = \llbracket v_N \rrbracket_{\text{def}} \{\phi_1, \dots, \phi_{N-1}\} -_{\text{register}} \omega_0(\delta, \varepsilon) \\
\mathcal{C} &= \{ v_1 \mapsto \phi_1, \dots, v_N \mapsto \phi_N, \\
&\quad v_{N+1} \mapsto 0, \dots, v_j \mapsto j - (N + 1) \} \\
&\text{in } (\mathcal{C}, \mathit{set-cc} \varepsilon^{\lambda_{\text{cur}}}. \mathcal{C} \varepsilon).
\end{aligned}$$

The expression $cc \lambda \varepsilon$ gives the calling convention for λ . The parameters of λ_{cur} are obtained with $\mathit{params} \lambda_{\text{cur}}$:

$$\begin{aligned}
\mathit{params}(\lambda \langle y_1, \dots, y_n \rangle. \mathcal{K} \dot{e}_0 \text{ at } \rho) &= (\mathbf{clos}, \mathbf{ret}, y_1, \dots, y_n) \\
\mathit{params}(f[\dot{\rho}_1, \dots, \dot{\rho}_k] \langle y_1, \dots, y_n \rangle = \mathcal{K} \dot{e}_0) \\
&= (\mathbf{clos}, \mathbf{ret}, y_1, \dots, y_n, \dot{\rho}_1, \dots, \dot{\rho}_k).
\end{aligned}$$

In case the calling convention is not decided, $\mathit{decide-cc}$ chooses a register in which to pass each parameter v_i using

$$\llbracket v_i \rrbracket_{\text{def}} \{\phi_1, \dots, \phi_{i-1}\} -_{\text{register}} \omega_0(\delta, \varepsilon),$$

which yields a register different from $\phi_1, \dots, \phi_{i-1}$ given ω_0 , the ω before e_0 (i.e., the ω describing the situation at the entry to λ_{cur}). Only the first N parameters get to be passed in a register, where N is approximately the number of registers (at least one register is needed by the code to call a function, p. 164). Parameters that are not passed in registers are passed on the stack at offsets 0 through $j - (N + 1)$.

The expression $\mathit{set-cc} \lambda_{\text{cur}}. \mathcal{C} \varepsilon$ yields a per-function environment ε' that records that the calling convention for λ_{cur} is \mathcal{C} . (It is, of course, the inter-procedural environment component, ε^η , of ε which is updated.)

Notice that $\mathit{decide-cc}$ can use $\llbracket \cdot \rrbracket_{\text{def}}$ which was introduced to take care of values declared by **let**-expressions etc.

The function $\mathit{decide-rc}$ is analogous to $\mathit{decide-cc}$: If the returning convention is decided, just return it, otherwise, decide what it should be. If possible, choose the natural destination register $\dot{\phi}$ of e_0 :

$$\begin{aligned}
\mathit{decide-rc} \dot{\phi}(\delta, \varepsilon) &= \text{if } rc \varepsilon^{\lambda_{\text{cur}}}. \varepsilon \neq -_{rc} \text{ then } (rc \varepsilon^{\lambda_{\text{cur}}}. \varepsilon, \varepsilon) \\
&\text{else let } \phi_{\text{res.}} = \mathit{choose} \dot{\phi} \emptyset \omega'_0 \delta \text{ in} \\
&\quad (\phi_{\text{res.}}, \mathit{set-rc} \varepsilon^{\lambda_{\text{cur}}}. \phi_{\text{res.}} \varepsilon).
\end{aligned}$$

where rc is analogous to cc and $\mathit{set-rc}$ to $\mathit{set-cc}$, and ω'_0 is the ω -information after the body of λ_{cur} .

8.12 The remaining constructs

The code for an integer constant, i , moves the constant to the destination register. There is no natural destination register:

$$\llbracket i \rrbracket_{\text{ra}} \mu = (\mu, \text{-register}, \lambda\phi. \lambda\zeta. \phi := \llbracket i \rrbracket_{I \rightarrow I}).$$

The code for $u e_2$ is analogous to that for $e_1 o e_2$ (p. 134):

$$\phi_2 := \boxed{\text{code to evaluate } e_2} ; \llbracket u \rrbracket_{\text{u-prim}} \phi_2 \phi,$$

where $\llbracket u \rrbracket_{\text{u-prim}} \phi_2 \phi$ translates the unary operator u to code that computes the result from ϕ_2 and puts it in ϕ :

$$\llbracket \#i \rrbracket_{\text{u-prim}} \phi_2 \phi = \phi := \mathbf{m}[\phi_2 + \llbracket i \rrbracket_{I \rightarrow I}]$$

$$\llbracket ! \rrbracket_{\text{u-prim}} \phi_2 \phi = \llbracket \#0 \rrbracket_{\text{u-prim}} \phi_2 \phi = \phi := \mathbf{m}[\phi_2 + 0].$$

Then

$$\begin{aligned} \llbracket u e_2 \rrbracket_{\text{ra}} \mu &= \text{let } (\mu, \dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \mu \\ &\quad (\mu, \phi_2) = \text{tmp-tmp } \phi_2 \dot{\phi}_2 \omega'_2 \mu \\ &\quad \beta = \lambda\phi. \lambda\zeta. \beta_2 \phi_2 \zeta ; \llbracket u \rrbracket_{\text{u-prim}} \phi_2 \phi \\ &\quad \text{in } (\mu, \text{-register}, \beta), \end{aligned}$$

where ω'_2 is the ω -information after e_2 .

The translation function for $\dot{c}_1 e_2 \text{ at } \rho$ is very similar to that for a pair. Compare with the definition of $\llbracket (e_1, \dots, e_n) \text{ at } \rho \rrbracket_{\text{ra}}$ (p. 146).

$$\begin{aligned} \llbracket \dot{c}_1 e_2 \text{ at } \rho \rrbracket_{\text{ra}} \mu &= \\ &\text{let } (\mu, \phi_*, \zeta, o) = \llbracket \rho \rrbracket_{\text{ra-at } 2} (\llbracket e_2 \rrbracket_{\text{da}} \mu^n) \omega_2 \mu \\ &\quad (\mu, \phi_1) = \text{tmp-tmp } \{\phi_*\} \text{-register } \omega_2 \mu \\ &\quad (\mu, \dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \mu \\ &\quad (\mu, \phi_2) = \text{tmp-tmp } \{\phi_*\} \dot{\phi}_2 \omega'_2 \mu \\ &\quad (\mu, p_*) = \text{kill-tmp } \mu \\ &\quad \beta = \lambda\phi. \lambda\zeta. \zeta \zeta ; \phi_1 := \llbracket \dot{c} \rrbracket_{C \rightarrow I} ; \mathbf{m}[\phi_* + o\zeta 0] := \phi_1 ; \\ &\quad \quad \quad p_*(\beta_2 \phi_2) \zeta ; \mathbf{m}[\phi_* + o\zeta 1] := \phi_2 ; \\ &\quad \quad \quad \langle \phi := \phi_* + o\zeta 0 \rangle \\ &\quad \dot{\phi} = \text{if } \phi_* = \phi_{\text{sp}} \text{ then } \text{-register} \text{ else } \phi_* \\ &\quad \text{in } (\mu, \dot{\phi}, \beta). \end{aligned}$$

where ω_1 and ω'_2 is the ω -information before e_1 and after e_2 , respectively. The “2” is the number of words allocated for the constructor and its argument.

The translation function for an application of a nullary constructor is similar. There is no sub-expression, and we only allocate one word:

$$\begin{aligned}
\llbracket \dot{c} \text{ at } \rho \rrbracket_{\text{ra}} \mu &= \text{let } (\mu, \phi_*, \zeta, o) = \llbracket \rho \rrbracket_{\text{ra-at}} 1 \mathcal{O} \omega \mu \\
&\quad (\mu, \phi_1) = \text{tmp-tmp } \{\phi_*\} \text{-register } \omega' \mu \\
&\quad (\mu, p_*) = \text{kill-tmp } \mu \\
&\quad \beta = \lambda \phi. \lambda \zeta. \\
&\quad \quad \zeta \zeta ; \phi_1 := \llbracket \dot{c} \rrbracket_{C \rightarrow I} ; \mathbf{m}[\phi_* + o\zeta 0] := \phi_1 ; \\
&\quad \quad \quad \langle \phi := \phi_* + o\zeta 0 \rangle \\
&\quad \dot{\phi} = \text{if } \phi_* = \phi_{\text{sp}} \text{ then } \text{-register} \text{ else } \phi_* \\
&\text{in } (\mu, \dot{\phi}, \beta).
\end{aligned}$$

Notice p_* is not used.

The translation function for `ref e_1 at ρ` is much like that for `$\dot{c}_1 e_2$ at ρ` (p. 177):

$$\begin{aligned}
\llbracket \text{ref } e_1 \text{ at } \rho \rrbracket_{\text{ra}} \mu &= \text{let } (\mu, \phi_*, \zeta, o) = \llbracket \rho \rrbracket_{\text{ra-at}} 1 (\llbracket e_1 \rrbracket_{\text{da}} \mu^\eta) \omega_1 \mu \\
&\quad (\mu, \dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \mu \\
&\quad (\mu, \phi_1) = \text{tmp-tmp } \{\phi_*\} \dot{\phi}_1 \omega'_1 \mu \\
&\quad (\mu, p_*) = \text{kill-tmp } \mu \\
&\quad \beta = \lambda \phi. \lambda \zeta. \\
&\quad \quad \zeta \zeta ; p_*(\beta_1 \phi_1) \zeta ; \mathbf{m}[\phi_* + o\zeta 0] := \phi_1 ; \\
&\quad \quad \quad \langle \phi := \phi_* + o\zeta 0 \rangle \\
&\quad \dot{\phi} = \text{if } \phi_* = \phi_{\text{sp}} \text{ then } \text{-register} \text{ else } \phi_* \\
&\text{in } (\mu, \dot{\phi}, \beta).
\end{aligned}$$

The code to declare an exception constructor is (p. 51)

$$\begin{aligned}
\phi := \boxed{\text{code to evaluate exception } a \text{ in } e_2} &= \\
&\quad \mathbf{n} := \mathbf{n} + 1 ; \\
&\quad \text{bind } a \text{ to } \mathbf{n} \text{ in the environment ;} \\
&\quad \phi := \boxed{\text{code to evaluate } e_2}.
\end{aligned}$$

Allocating some register to hold \mathbf{n} is unreasonable, as this would reduce the number of available registers, and \mathbf{n} will probably not be used often in the vast majority of programs. Therefore, we choose to keep \mathbf{n} in some memory cell.

Assuming \mathbf{n} 's offset from ϕ_{dp} is ι_n , the code is

$$\begin{aligned}
\phi := \boxed{\text{code to evaluate exception } a \text{ in } e_2} &= \\
&\quad \phi_a := \mathbf{m}[\phi_{\text{dp}} + \iota_n] ; \phi_a := \phi_a + 1 ; \mathbf{m}[\phi_{\text{dp}} + \iota_n] := \phi_a ; \\
&\quad \phi := \boxed{\text{code to evaluate } e_2}.
\end{aligned}$$

This is quite similar to the code for a **let**-expression, and the definition of $\llbracket \text{exception } a \text{ in } e_2 \rrbracket_{\text{ra}}$ is almost identical to that of $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$ (compare with p. 104):

$$\begin{aligned}
\llbracket \text{exception } a \text{ in } e_2 \rrbracket_{\text{ra}} \mu &= \text{let } (\mu, \phi_a) = \llbracket a \rrbracket_{\text{def}} \emptyset -_{\text{register}} \omega_2 \mu \\
&\quad (\mu, \overset{\dagger}{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \mu \\
&\quad (\mu, p_a) = \llbracket a \rrbracket_{\text{kill}} \phi_a \mu \\
&\quad \beta = \lambda \phi. \lambda \varsigma. \phi_a := \mathfrak{m}[\phi_{\text{dp}} + \iota_n] ; \\
&\quad \quad \phi_a := \phi_a + 1 ; \\
&\quad \quad \mathfrak{m}[\phi_{\text{dp}} + \iota_n] := \phi_a ; \\
&\quad \quad p_a(\beta_2 \phi) \varsigma \\
&\text{in } (\mu, \overset{\dagger}{\phi}_2, \beta).
\end{aligned}$$

The definitions of $\llbracket \overset{\circ}{a}_1 e_2 \text{ at } \rho \rrbracket_{\text{ra}}$ and $\llbracket \overset{\circ}{a} \text{ at } \rho \rrbracket_{\text{ra}}$ are quite similar to the definitions of $\llbracket \overset{\circ}{c}_1 e_2 \text{ at } \rho \rrbracket_{\text{ra}}$ and $\llbracket \overset{\circ}{c} \text{ at } \rho \rrbracket_{\text{ra}}$, respectively. See the code sketches and discussion on p. 52. The only difference is that the exception constructs represent a use of the exception constructor. We will only give the definition of $\llbracket \overset{\circ}{a}_1 e_2 \text{ at } \rho \rrbracket_{\text{ra}}$; compare with $\llbracket \overset{\circ}{c}_1 e_2 \text{ at } \rho \rrbracket_{\text{ra}}$ (p. 177):

$$\begin{aligned}
\llbracket \overset{\circ}{a}_1 e_2 \text{ at } \rho \rrbracket_{\text{ra}} \mu &= \\
&\text{let } (\mu, \phi_*, \zeta, o) = \llbracket \rho \rrbracket_{\text{ra-at}} 2 (\llbracket e_2 \rrbracket_{\text{da}} \mu^n) \omega_2 \mu \\
&\quad (\mu, \phi_{\overset{\circ}{a}}, \beta_{\overset{\circ}{a}}) = \llbracket \overset{\circ}{a} \rrbracket_{\text{use}} \{\phi_*\} -_{\text{register}} \omega_2 \mu \\
&\quad (\mu, \overset{\dagger}{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \mu \\
&\quad (\mu, \phi_2) = \text{tmp-tmp } \{\phi_*\} \overset{\dagger}{\phi}_2 \omega'_2 \mu \\
&\quad (\mu, p_*) = \text{kill-tmp } \mu \\
&\quad \beta = \lambda \phi. \lambda \varsigma. \zeta \varsigma ; \quad \beta_{\overset{\circ}{a}} \phi_{\overset{\circ}{a}} \varsigma ; \quad \mathfrak{m}[\phi_* + o\varsigma 0] := \phi_{\overset{\circ}{a}} \\
&\quad \quad p_*(\beta_2 \phi_2) \varsigma ; \quad \mathfrak{m}[\phi_* + o\varsigma 1] := \phi_2 ; \\
&\quad \quad \langle \phi := \phi_* + o\varsigma 0 \rangle \\
&\quad \overset{\dagger}{\phi} = \text{if } \phi_* = \phi_{\text{sp}} \text{ then } -_{\text{register}} \text{ else } \phi_* \\
&\text{in } (\mu, \overset{\dagger}{\phi}, \beta).
\end{aligned}$$

A λ -abstraction $\lambda \vec{y}.^{\mathcal{K}} e_0 \text{ at } \rho$ has annotated a closure representation $\mathcal{K} \in Z \xrightarrow{\perp} I$, which maps the free variables of $\lambda \vec{y}.^{\mathcal{K}} e_0 \text{ at } \rho$ to their offsets in the closure.

The code for **letrec** ^{\mathcal{K}} $b_1 \dots b_m \text{ at } \rho \text{ in } e_{m+1}$ builds a shared closure (p. 43), and this is much like building a closure for a λ -abstraction. We factor the common parts into a function: *build-closure* $\mathcal{K} \rho \tilde{\omega} \mu$ returns the usual triple $(\mu', \overset{\dagger}{\phi}, \beta)$, where β is the code to build a closure for the closure representation \mathcal{K} in region ρ and with code pointer $\tilde{\iota} \in I \cup \{-\mathbf{I}\}$. If $\tilde{\iota} = -\mathbf{I}$, no code pointer is stored into the closure. The ω -information is ω , and $\overset{\dagger}{\phi}$ is the natural destination register of the code to build the expression. Now we can write $\llbracket \lambda \vec{y}.^{\mathcal{K}} e_0 \text{ at } \rho \rrbracket_{\text{ra}}$ using this auxiliary function:

$$\llbracket \lambda \vec{y}.^{\mathcal{K}} e_0 \text{ at } \rho \rrbracket_{\text{ra}} \mu = \text{build-closure } \mathcal{K} \rho \llbracket \lambda \vec{y}.^{\mathcal{K}} e_0 \text{ at } \rho \rrbracket_{\Lambda \rightarrow \mathbf{I}} \omega \mu,$$

where $\llbracket \lambda_{\text{cur.}} \rrbracket_{\Lambda \rightarrow \mathbb{I}}$ is the unique label for $\lambda_{\text{cur.}}$, and ω is the ω -information before $\lambda \vec{y}.^{\mathcal{K}} e_0$ at ρ .

The code to build a closure (p. 40) is very similar to that for building a tuple. Instead of storing the values of the n sub-expressions at offsets 0 through $n - 1$, the label of the code for the λ is stored at offset 0, and the free variables v_1, \dots, v_n are stored at offsets $\mathcal{K}v_1, \dots, \mathcal{K}v_n$. Accordingly, the register allocation part of *build-closure* is quite similar to that of $\llbracket (e_1, \dots, e_n) \text{ at } \rho \rrbracket_{\text{ra}}$ (p. 146):

$$\begin{aligned}
\text{build-closure } \mathcal{K}\tilde{\rho}\tilde{\omega}\mu &= \\
\text{let } \{z_1, \dots, z_n\} &= \text{Dm } \mathcal{K} \\
(\mu, \phi_*, \zeta, o) &= \llbracket \rho \rrbracket_{\text{ra-at}} (n + \text{if } \tilde{t} = -\mathbb{I} \text{ then } 0 \text{ else } 1) \oslash \omega \mu \\
(\mu, \phi_0) &= \text{tmp-tmp } \{\phi_*\} \text{ -register } \omega \mu \\
(\mu, \phi_1, \beta_1) &= \llbracket z_1 \rrbracket_{\text{use}} \{\phi_*\} \text{ -register } \omega \mu \\
&\vdots \\
(\mu, \phi_n, \beta_n) &= \llbracket z_n \rrbracket_{\text{use}} \{\phi_*\} \text{ -register } \omega \mu \\
(\mu, p_*) &= \text{kill-tmp } \mu \\
\beta &= \lambda \phi. \lambda \varsigma. \zeta \varsigma ; \\
&\quad \text{if } \tilde{t} = -\mathbb{I} \text{ then } \epsilon \\
&\quad \quad \text{else } \phi_0 := \tilde{t} ; \mathbf{m}[\phi_* + o\varsigma 0] := \phi_0 ; \\
&\quad \beta_1 \phi_1 \varsigma ; \mathbf{m}[\phi_* + o\varsigma(\mathcal{K}z_1)] := \phi_1 ; \\
&\quad \vdots \\
&\quad \beta_n \phi_n \varsigma ; \mathbf{m}[\phi_* + o\varsigma(\mathcal{K}z_n)] := \phi_n ; \\
&\quad \langle \phi := \phi_* + o\varsigma 0 \rangle \\
\dot{\phi} &= \text{if } \phi_* = \phi_{\text{sp}} \text{ then } \text{-register} \text{ else } \phi_* \\
\text{in } (\mu, \dot{\phi}, \beta).
\end{aligned}$$

Besides building a closure, the construct

$$\text{letrec } f_1 \vec{\rho}_1 y_1 = e_1 \cdots f_m \vec{\rho}_m y_m = e_m \text{ at } \rho \text{ in } e_{m+1}$$

declares a sibling name $\mathbf{f} = \{f_1, \dots, f_m\}$ (p. 117) in e_{m+1} , just as $\text{let } x = e_1 \text{ in } e_2$ declares x in e_2 . Accordingly, $\llbracket \text{letrec }^{\mathcal{K}} b_1 \cdots b_m \text{ at } \rho \text{ in } e_{m+1} \rrbracket_{\text{ra}}$ is like $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$, except it uses *build-closure* $\mathcal{K}\rho_1 - \mathbb{I} \omega_{m+1} \mu$ instead of $\llbracket e_1 \rrbracket_{\text{ra}}$, and \mathbf{f} instead of x :

$$\begin{aligned}
& \llbracket \text{letrec } f_1 \vec{\rho}_1 y_1 = e_1 \cdots f_m \vec{\rho}_m y_m = e_m \text{ at } \rho \text{ in } e_{m+1} \rrbracket_{\text{ra}} \mu = \\
& \quad \text{let } \mathbf{f} = \{f_1, \dots, f_m\} \\
& \quad (\mu, \dot{\phi}_1, \beta_1) = \text{build-closure } \mathcal{K} \rho -_{\mathbf{I}} \omega_{m+1} \mu \\
& \quad (\mu, \dot{\phi}_{\mathbf{f}}) = \llbracket \mathbf{f} \rrbracket_{\text{def}} \mathcal{O} \dot{\phi}_1 \omega_{m+1} \mu \\
& \quad (\mu, \dot{\phi}_{m+1}, \beta_{m+1}) = \llbracket e_{m+1} \rrbracket_{\text{ra}} \mu \\
& \quad (\mu, p_{\mathbf{f}}) = \llbracket \mathbf{f} \rrbracket_{\text{kill}} \dot{\phi}_{\mathbf{f}} \mu \\
& \quad \beta = \lambda \phi. \beta_1 \phi_{\mathbf{f}} ; p_{\mathbf{f}} (\beta_{m+1} \phi) \\
& \quad \text{in } (\mu, \dot{\phi}_{m+1}, \beta),
\end{aligned}$$

where ω_{m+1} is the ω -information before e_{m+1} .

9 Target code generation

This chapter describes the last phases in the back end:

$$K \xrightarrow{\llbracket \cdot \rrbracket_{\text{bbs}}} \mathcal{PB} \xrightarrow{\text{lin}} \hat{K} \xrightarrow{\llbracket \cdot \rrbracket_{\text{pa}}} \mathfrak{P} \xrightarrow{\llbracket \cdot \rrbracket_{\text{sched.}}} \mathfrak{P}.$$

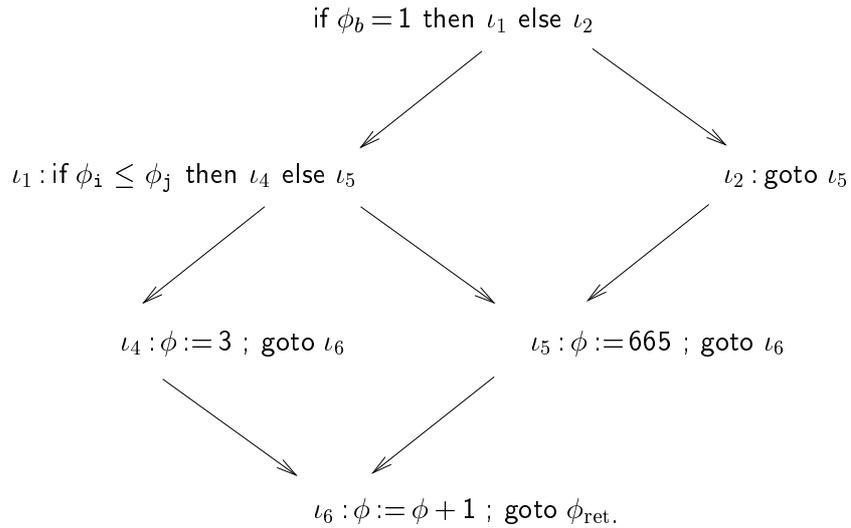
9.1 Linearising the code

Viewing the code as basic blocks

One can view K as a control flow graph language. For instance, the code generated by $\llbracket \cdot \rrbracket_{\text{ra}}$ for the body of the function

`λ<b,i,j>. (if b andalso i<=j then 3 else 665)+1 at r299`

is something like



The notion of a control flow graph representation of the code is formalised by regarding the code as a set of basic blocks.

A *basic block* $b \in B$ has the form

$$B ::= I : \hat{K} ; \check{K}$$

where \hat{K} is the set of jump instructions in K , and \check{K} is the set of instructions in K which do not contain jump or label instructions:

$$\begin{aligned} \hat{K} &::= \text{goto } \Upsilon \quad | \quad \text{if } X \text{ then } I \text{ else } I \\ \check{K} &::= \Phi := \Phi \pm \Upsilon \quad | \quad \Phi := \Upsilon \\ &\quad | \quad \Phi := m[\Phi \pm I] \quad | \quad m[\Phi \pm I] := \Phi \\ &\quad | \quad \check{K} ; \check{K} \quad | \quad \epsilon \\ \Upsilon &::= \Phi \quad | \quad I. \end{aligned}$$

(Remember X is the set of conditions (p. 20). As in K , the other instructions (e.g., pop) can be defined as macros in this language.)

We assume some function, $\llbracket \cdot \rrbracket_{\text{bbs}} \in K \rightarrow \mathcal{PB}$, can convert the code generated by $\llbracket \cdot \rrbracket_{\text{ra}}$ to a set of basic blocks.

Linear code

The control flow graph representation of the code is convenient when generating code: pieces of code can be glued together using jumping and labelling instructions. But the assembly language has one sequence of instructions, and instead of the symmetric if χ then ι else $\bar{\iota}$ it has a conditional jump instruction if χ then ι which falls through to the following instruction if the condition χ is false. Therefore, we define *linear code*, \hat{K} , to be like K , except that the if χ then ι -construct replaces if χ then ι else $\bar{\iota}$:

$$\hat{K} ::= I : \hat{K} \mid \dots \mid \text{goto } \Upsilon \mid \text{if } X \text{ then } I.$$

We must translate the set of basic blocks to linear code. Since we do not want to pay a price at run-time for the convenience of using jumps to glue together pieces of code, the translation from basic blocks to linear code should try to generate as few jumps as possible.

For instance, the linear code for the control flow graph above could be:

```

        if  $\phi_b = 1$  then  $\iota_{\neg b}$  ;
 $\iota_b$  :  $\phi := 665$  ;
         $\phi := \phi + 1$  ;
        goto  $\phi_{\text{ret.}}$  ;
 $\iota_{\neg b}$  : if  $\neg\phi_i \leq \phi_j$  then  $\iota_b$  ;
         $\phi := 3$  ;
         $\phi := \phi + 1$  ;
        goto  $\phi_{\text{ret.}}$  .

```

This linear code was obtained by way of the following optimisations:

1. Avoid jumps to jumps. (This avoided $\iota_2 : \text{goto } \iota_5$.)
2. A jump can be avoided by placing the jumpee right after the jumper. (This avoided jumps to $\phi := 665$ and $\phi := 3$.)
3. If the jumpee has already been put somewhere, the jump can still be avoided by duplicating the jumpee. (The code $\phi := \phi + 1 ; \text{goto } \phi_{\text{ret.}}$ was duplicated to avoid a jump instruction in the branches.) To avoid an explosion in code size, code should only be duplicated when it is sufficiently small; and restrictions must also be imposed to ensure termination.

The first optimisation is a special case of the third if we assume a jump will be sufficiently small to be duplicated.

Making linear code from basic blocks

In the following, we explain the function $lin \in \mathcal{PB} \rightarrow \hat{K}$ which linearises a set of basic blocks by translating it to a $\hat{\kappa} \in \hat{K}$.

During the translation of the set of basic blocks, we keep track of the status of each basic block in an environment, i , which maps each label in the program to the status of the corresponding basic block:

$$i \in I = I \xrightarrow{\perp} B \cup \{\mathbf{underway}\} \cup \hat{K}$$

If $i \iota = \mathbf{b}$, the basic block corresponding to ι is \mathbf{b} and has not yet been processed; if $i \iota = \mathbf{underway}$, the basic block corresponding to ι is currently being processed; and if $i \iota = \hat{\kappa}$, the processing of the basic block corresponding to ι has been finished and resulted in the linear code $\hat{\kappa}$.

The function $\llbracket \cdot \rrbracket_{lin-1} \in B \rightarrow I \rightarrow I$ processes a \mathbf{b} and returns an updated i that records the result of processing \mathbf{b} , i.e., the resulting code, $\hat{\kappa}$. It first records in i that \mathbf{b} is underway, and then calls $\llbracket \cdot \rrbracket_{lin-2} \in B \rightarrow I \rightarrow I$:

$$\llbracket \iota : \hat{\kappa} ; \hat{\kappa} \rrbracket_{lin-1} i = \llbracket \iota : \hat{\kappa} ; \hat{\kappa} \rrbracket_{lin-2} (i + \{ \iota \mapsto \mathbf{underway} \})$$

The actual work is done by $\llbracket \mathbf{b} \rrbracket_{lin-2} i$: If \mathbf{b} has the form $\iota : \hat{\kappa} ; \mathbf{goto} \iota'$, we try to eliminate the $\mathbf{goto} \iota'$ by putting the basic block labelled ι' (call it \mathbf{b}') right after \mathbf{b} . The following situations can arise: 1. If \mathbf{b}' has not been processed yet (i.e., $i \iota' = \mathbf{b}'$), we can process it now, and glue the resulting code onto \mathbf{b} . 2. If \mathbf{b}' has been processed (i.e., $i \iota'$ is some $\hat{\kappa}'$), we cannot eliminate $\mathbf{goto} \iota'$, except if the code ($\hat{\kappa}'$) for \mathbf{b}' is so small that it is permissible to duplicate it to avoid the jump. 3. If \mathbf{b}' is underway (i.e., $i \iota' = \mathbf{underway}$), we do not eliminate $\mathbf{goto} \iota'$:

$$\begin{aligned} \llbracket \iota : \hat{\kappa} ; \mathbf{goto} \iota' \rrbracket_{lin-2} i &= \text{if } i \iota' = \mathbf{b}' \text{ then let } i = \llbracket \mathbf{b} \rrbracket_{lin-1} i \\ &\quad \text{in } i + \{ \iota \mapsto \iota : \hat{\kappa} ; i \iota' \} \\ &\quad \text{else } i + \{ \iota \mapsto \iota : \hat{\kappa} ; \text{if } i \iota' = \hat{\kappa}' \wedge |\hat{\kappa}'| < 42 \\ &\quad \quad \text{then } \llbracket \hat{\kappa}' \rrbracket_{relabel} \\ &\quad \quad \text{else } \mathbf{goto} \iota' \}, \end{aligned}$$

where “ $|\hat{\kappa}'| < 42$ ” means that $\hat{\kappa}'$ is sufficiently small to be duplicated, and $\llbracket \hat{\kappa}' \rrbracket_{relabel}$ is a $\hat{\kappa}$ where the labels bound in $\hat{\kappa}'$ have been replaced by labels that are not used anywhere else.

Notice that in the case where an already processed $\hat{\kappa}'$ is glued onto the current basic block, we do not continue and try to eliminate the jump instruction in the end of $\hat{\kappa}'$, for if it could be eliminated, it would already have been when $\hat{\kappa}'$ was processed.

By keeping track of what basic blocks are **underway** and not eliminating jumps to them, we ensure that the algorithm does not loop infinitely when there are loops among the basic blocks.

Jumps to registers cannot be eliminated:

$$\llbracket \iota : \hat{\kappa} ; \text{goto } \phi \rrbracket_{\text{lin-2}} i = i + \{ \iota \mapsto \iota : \hat{\kappa} ; \text{goto } \phi \}$$

The conditional jump instruction $\text{if } \chi \text{ then } \iota' \text{ else } \iota''$ can be translated to the linear code $\text{if } \chi \text{ then } \iota' ; \text{goto } \iota''$, but as above, we try to avoid the jump instruction by putting the basic block labelled ι'' right after the current basic block. If this is not possible, we pretend the instruction is $\text{if } \neg\chi \text{ then } \iota'' \text{ else } \iota'$ and try to put the basic block labelled ι' right after the current basic block. If neither attempt succeeds, we try to duplicate the code for one of the basic blocks as above:

$$\begin{aligned} \llbracket \iota : \hat{\kappa} ; \text{if } \chi \text{ then } \iota' \text{ else } \iota'' \rrbracket_{\text{lin-2}} i &= \\ \text{if } i \iota'' = \mathbf{b}'' \text{ then let } i &= \llbracket \mathbf{b}'' \rrbracket_{\text{lin-1}} i \\ &\quad \text{in } i + \{ \iota \mapsto \iota : \hat{\kappa} ; \text{if } \chi \text{ then } \iota' ; i \iota'' \} \quad \text{else} \\ \text{if } i \iota' = \mathbf{b}' \text{ then let } i &= \llbracket \mathbf{b}' \rrbracket_{\text{lin-1}} i \\ &\quad \text{in } i + \{ \iota \mapsto \iota : \hat{\kappa} ; \text{if } \neg\chi \text{ then } \iota'' ; i \iota' \} \quad \text{else} \\ \text{if } i \iota' = \hat{\kappa}'' \wedge |\hat{\kappa}''| < 42 \text{ then} & \\ &\quad i + \{ \iota \mapsto \iota : \hat{\kappa} ; \text{if } \chi \text{ then } \iota' ; \llbracket \hat{\kappa}'' \rrbracket_{\text{relabel}} \} \quad \text{else} \\ \text{if } i \iota' = \hat{\kappa}' \wedge |\hat{\kappa}'| < 42 \text{ then} & \\ &\quad i + \{ \iota \mapsto \iota : \hat{\kappa} ; \text{if } \neg\chi \text{ then } \iota'' ; \llbracket \hat{\kappa}' \rrbracket_{\text{relabel}} \} \quad \text{else} \\ i + \{ \iota \mapsto \iota : \hat{\kappa} ; \text{if } \chi \text{ then } \iota' ; \text{goto } \iota'' \}. & \end{aligned}$$

The basic blocks of the program are processed by $\llbracket \cdot \rrbracket_{\text{lin-0}} \in \mathcal{B} \rightarrow I \rightarrow \hat{\mathbf{K}} \rightarrow \hat{\mathbf{K}}$, which repeatedly calls $\llbracket \cdot \rrbracket_{\text{lin-1}}$ until there are no more unprocessed basic blocks in the environment i :

$$\begin{aligned} \llbracket \iota : \hat{\kappa} ; \hat{\kappa} \rrbracket_{\text{lin-0}} i \hat{\kappa}_{\text{so-far}} &= \\ \text{let } i &= \llbracket \iota : \hat{\kappa} ; \hat{\kappa} \rrbracket_{\text{lin-1}} i \\ \hat{\kappa}_{\text{so-far}} &= \hat{\kappa}_{\text{so-far}} ; i \iota \\ \text{in if } \exists \iota : i \iota = \mathbf{b} \text{ then } &\llbracket \mathbf{b} \rrbracket_{\text{lin-0}} i \hat{\kappa}_{\text{so-far}} \text{ else } \hat{\kappa}_{\text{so-far}} \end{aligned}$$

The code for the different basic blocks is accumulated in $\hat{\kappa}_{\text{so-far}}$ which is returned when all basic blocks have been processed.

All $\text{lin} \in \mathcal{PB} \rightarrow \hat{\mathbf{K}}$ does is to set up the initial i and $\hat{\kappa}_{\text{so-far}}$, and then call $\llbracket \cdot \rrbracket_{\text{lin-0}}$ with the first basic block \mathbf{b} :

$$\text{lin}(\mathbf{b} \cup \{\mathbf{b}\}) = \llbracket \mathbf{b} \rrbracket_{\text{lin-0}} \{ \iota \mapsto \iota : \hat{\kappa} ; \hat{\kappa} \mid \iota : \hat{\kappa} ; \hat{\kappa} \in \mathbf{b} \} \epsilon.$$

Discussion

Shorter jumps are more efficient on many architectures. The algorithm could be extended to process the basic blocks in an order such that basic blocks that jump to each other are placed close to each other.

The heuristic for deciding when code can be duplicated could be extended to take into account the number of jumps to the code and not just its size.

$$\begin{aligned} \llbracket m[\phi_1 + \iota] := \phi_2 \rrbracket_{\text{pa}} &= \\ &\text{if } 2^{13} \leq 4 \cdot \iota \wedge 4 \cdot \iota < 2^{13} \text{ then } \mathbf{stw} \ \% \phi_2, 4 \cdot \iota (\% \mathbf{sr0}, \phi_1) \\ &\quad \text{else } \mathbf{addil} \ \% 4 \cdot \iota - \$ \mathbf{global} \$, \phi_1 \\ &\quad \quad \mathbf{stw} \ \% \phi_2, r \% 4 \cdot \iota (\% \mathbf{sr0}, \% \mathbf{r1}). \end{aligned}$$

The function $\llbracket \cdot \rrbracket_{\text{pa}}$ we have almost directly copied from the existing back end (Elsman and Hallenberg, 1995).

The PA-RISC has 31 registers:

$$\Phi = \{\mathbf{r1}, \dots, \mathbf{r31}\}.$$

The stack pointer is $\phi_{\text{sp}} = \mathbf{r30}$; the global data space pointer (p. 164) is $\phi_{\text{dp}} = \mathbf{r27}$; and since $\mathbf{r1}$ is used, as above, when a constant is too large to fit in an instruction, we prohibit its use in the register allocation. This leaves 28 registers at the disposal of the register allocator.

The heavy-weight instructions $\phi_1 := \text{at } \phi_2 : \iota$, $\phi := \text{letregion}$, etc. are implemented through a combination of translating them to simpler \hat{K} instructions, calls to sub-routines, and calls to the run-time system. The register allocator must know the set of registers destroyed by these instructions. E.g., in one configuration of the compiler, $\phi_1 := \text{at } \phi_2 : \iota$ and $\phi := \text{letregion}$ are coded completely in \hat{K} instructions, and

$$\begin{aligned} \hat{\phi}_{\text{at}} &= \{\mathbf{r1}, \mathbf{r20}, \mathbf{r21}, \mathbf{r23}, \dots, \mathbf{r26}\} \\ \hat{\phi}_{\text{letregion}} &= \{\mathbf{r1}, \mathbf{r20}, \mathbf{r21}\}, \end{aligned}$$

whereas $\text{endregions } \phi$ is implemented as a call to the run-time system, and thus $\hat{\phi}_{\text{endregion}}$ is the registers destroyed according to the PA-RISC calling convention (Hewlett-Packard, 1991b):

$$\hat{\phi}_{\text{endregion}} = \{\mathbf{r1}, \mathbf{r2}, \mathbf{r19}, \dots, \mathbf{r26}, \mathbf{r28}, \mathbf{r29}, \mathbf{r31}\}.$$

Similarly, we must define the natural destination registers and preferred argument registers of the heavy-weight instructions. For instance, $\phi_{\text{letregion}} = \mathbf{r28}$ when $\phi := \text{letregion}$ is implemented as a call to the run-time system.

9.3 Instruction scheduling

On pipelined processors *pipeline interlocks* can occur. This happens, for example, when a value is loaded from memory to a register and the instruction immediately after the load instruction uses this register. For example:

```
p1: ldw 0(%sr0, %r1), %r3
p2: add %r2, %r3, %r4
p3: add %r5, %r6, %r7
```

On the PA-RISC, the time needed to transfer a value between cache and a register is one cycle, so **r3** is not ready by the time **p₂** needs it. This causes the processor to wait for a cycle until the contents of the memory location pointed to by **r1** has reached **r3**. Since **p₃** does not use **r3**, there is no interlock if the order of the instructions is changed to **p₁p₃p₂**. This does not change the semantics of the program, as **p₃** does not depend on **p₂**. (Since **p₃** does not depend on **p₁** either, the sequence **p₃p₁p₂** would also have the same semantics, but an interlock would occur.)

The purpose of instruction scheduling is to try to avoid interlocks where possible, by rearranging instructions to produce a (hopefully) faster program. In the following, we first discuss a general algorithm for scheduling. Then specifics concerning the PA-RISC are covered. The main inspiration for the algorithm stems from (Gibbons and Muchnick, 1986).

The scheduling algorithm

To reorder the instructions in the original program without changing the semantics of it, it must be recorded how instructions depend on each other, and a strategy must be chosen to decide on the reordering. The main issue that should be addressed to solve the first problem is: When is it legal to swap two instructions?

The scheduling algorithm is used only on basic blocks. This is because it is not always possible at compile-time to determine where control will flow at jumps. Completely different dependencies could arise between the instructions, depending on the flow of control.

In the example above, it is clear that **p₃** may be executed at any point. This is because **p₃** does not use any registers that are destroyed by **p₁** and **p₂**, and it does not destroy any registers that are used by **p₁** and **p₂**.

Consider a basic block consisting of instructions **p₁, . . . , p_j, . . . , p_k, . . . , p_n** ($1 \leq j < k \leq n$). (In this section, a basic block is a sequence of \mathfrak{P} -instructions.) The instruction dependencies are:

1. If **p_j** destroys a register that is used by **p_k**, then **p_j** must be executed before **p_k**.
2. If **p_j** uses a register that is destroyed by **p_k**, then **p_j** must be executed before **p_k**.
3. If **p_j** destroys a register that **p_k** also destroys, then **p_j** must be executed before **p_k**. This rule is redundant if there is at least one instruction **p_l** between **p_j** and **p_k** that uses the register destroyed by **p_j**, because then, by rule 1, **p_j** must come before **p_l**, and by rule 2, **p_l** must come before **p_k**. If there are no uses of the register that **p_j** destroys before **p_k** is reached, then **p_j** could have been safely removed by an optimisation phase. Since this does not happen in the current implementation, these instructions must not be swapped, and this rule is necessary. We will also see other reasons for this rule below.

The different kinds of dependencies are illustrated by this basic block:

```

p4 :  add  %r1,%r2,%r3
p5 :  add  %r3,%r3,%r4
p6 :  add  %r0,%r0,%r3

```

These instructions cannot be reordered: By rule 1, p_4 must be executed before p_5 because p_4 destroys $r3$, which is used by p_5 . By rule 2, p_5 must come before p_6 because p_6 destroys $r3$, which is used by p_5 . By rule 3, p_4 must come before p_6 , but that is already ensured by the two other rules.

It is computationally unwise to consider the $n!$ possible rearrangements of a given basic block with n instructions to find the optimal scheduling. Therefore we use a heuristic. We construct a *dependency graph* $(\mathbf{p}, \mathbf{d}) \in \mathfrak{P} \times \mathcal{P}(\mathfrak{P} \times \mathfrak{P})$, which is a directed graph that reflects how instructions depend on each other according to the rules above: p_2 depends on p_1 according to (\mathbf{p}, \mathbf{d}) iff $p_1 \mathbf{d}^* p_2$. Any topological sort of (\mathbf{p}, \mathbf{d}) will result in a reordered basic block that is semantically equivalent to the original basic block.² A heuristic that seeks to minimise the number of interlocks is used to choose a node among those nodes with no parents at each step in the topological sort. It is possible to construct pathological heuristics that will produce worse scheduling than the original program.

The algorithm in detail

The function $\llbracket \cdot \rrbracket_{\text{sched.}} \in \mathfrak{P} \rightarrow \mathfrak{P}$ schedules a program by converting the program to basic blocks, scheduling each basic block, and putting the basic blocks back together:

$$\begin{aligned}
\llbracket \mathbf{p} \rrbracket_{\text{sched.}} &= \\
&\text{let } (b_1, \dots, b_m) = \llbracket \mathbf{p} \rrbracket_{\text{bbs}'} \\
&\quad b'_1 = \llbracket b_1 \rrbracket_{\text{sched}0} \\
&\quad \vdots \\
&\quad b'_m = \llbracket b_m \rrbracket_{\text{sched}0} \\
&\text{in} \\
&\quad b'_1 \bullet \dots \bullet b'_m
\end{aligned}$$

where $b_1 \bullet b_2$ is defined by:

$$(p_1, \dots, p_n) \bullet (p_{n+1}, \dots, p_m) = (p_1, \dots, p_m).$$

The function $\llbracket \cdot \rrbracket_{\text{bbs}'}$ is trivial and will not be described further.

The function $\llbracket \cdot \rrbracket_{\text{sched}0}$ processes a basic block by making a dependency graph, (\mathbf{p}, \mathbf{d}) , and then applying the heuristic to produce a sequence of instructions with as few interlocks as possible from this graph:

²Topological sort: Choose a node with no parents, and delete it. Continue till the graph is empty

```

[[b]]sched0 =
  let (p, d) = dependencies b
  while (p, d) is not empty
    let the candidate set pc =
      the set of instructions with no predecessors in (p, d);
    apply the heuristic to pc, yielding a choice p;
    remove p from (p, d);

```

The next sub-sections explain *dependencies* and the heuristic.

Building the dependency graph

The set of registers that is used and destroyed by an instruction set is processor dependent. This information is recorded in the functions $[[\cdot]]_{\text{used}}$ and $[[\cdot]]_{\text{defd}}$:

$$\begin{aligned} [[\cdot]]_{\text{used}} &\in \mathfrak{P} \xrightarrow{\perp} \mathcal{P}\Phi \\ [[\cdot]]_{\text{defd}} &\in \mathfrak{P} \xrightarrow{\perp} \mathcal{P}\Phi \end{aligned}$$

The function $[[\cdot]]_{\text{used}}$ computes which registers an instruction uses. Similarly, $[[\cdot]]_{\text{defd}}$ computes which registers an instruction destroys. For example:

$$\begin{aligned} [[\text{add}\%r1,\%r2,\%r3]]_{\text{used}} &= \{\mathbf{r1}, \mathbf{r2}\}, \\ [[\text{add}\%r1,\%r2,\%r3]]_{\text{defd}} &= \{\mathbf{r3}\}. \end{aligned}$$

During the processing of a basic block, the values necessary to build the dependency graph are maintained by maps \mathcal{D} and \mathcal{U} ,

$$\begin{aligned} \mathcal{D} &\in \Phi \rightarrow \mathfrak{P} \\ \mathcal{U} &\in \Phi \rightarrow \mathcal{P}\mathfrak{P}. \end{aligned}$$

The last instruction that destroys ϕ is given by $\mathcal{D}\phi$. Similarly, $\mathcal{U}\phi$ is the set of instructions that use ϕ . The maps \mathcal{D} and \mathcal{U} are updated on-the-fly after each instruction in the basic block is processed. Thus, they only record information from the beginning of the current basic block to the instruction currently being processed. When an instruction destroys a register, all later uses will depend on this definition (rule 1). Therefore the use set, $\mathcal{U}\phi$, of ϕ may be set to \emptyset when ϕ is destroyed.

The functions *dependencies* and *dependencies₀* build the dependency graph. *dependencies₀* p \mathcal{U} \mathcal{D} is a set \mathbf{d} consisting of edges representing the instructions that p depends on, and *dependencies* produces the dependency graph for the whole basic block.

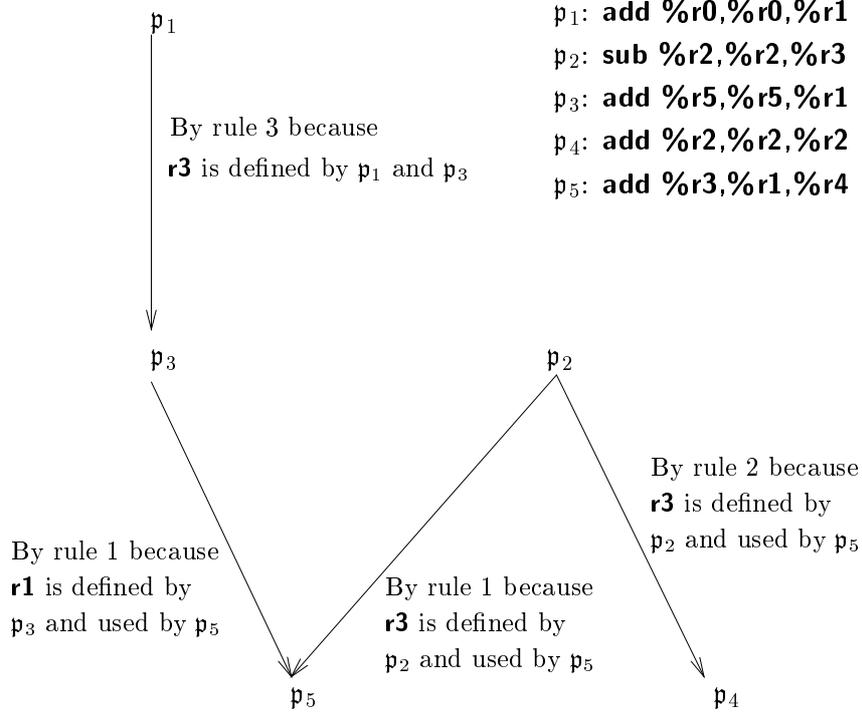


Fig. 41. *The dependency graph for a program.* On each edge is an explanation of the rule used to insert the edge into the graph.

$$\begin{aligned}
 & \text{dependencies}_0 \mathbf{p} \mathcal{U} \mathcal{D} = \\
 & \text{let } \mathbf{d} = \{ (p', p) \mid p' \in \{ \mathcal{D} \phi \mid \phi \in \llbracket p \rrbracket_{\text{used}} \} \cup \quad (\text{rule 1}) \\
 & \quad \quad \quad \{ \mathcal{U} \phi \mid \phi \in \llbracket p \rrbracket_{\text{defd}} \} \cup \quad (\text{rule 2}) \\
 & \quad \quad \quad \{ \mathcal{D} \phi \mid \phi \in \llbracket p \rrbracket_{\text{defd}} \} \} \quad (\text{rule 3}) \\
 & \mathcal{U} = \mathcal{U} + \{ \phi \mapsto (\mathcal{U} \phi \cup \{ p \}) \mid \phi \in \llbracket p \rrbracket_{\text{used}} \} \\
 & \mathcal{D} = \mathcal{D} + \{ \phi \mapsto p \mid \phi \in \llbracket p \rrbracket_{\text{defd}} \} \\
 & \mathcal{U} = \mathcal{U} + \{ \phi \mapsto \emptyset \mid \phi \in \llbracket p \rrbracket_{\text{defd}} \} \\
 & \text{in} \\
 & (\mathcal{U}, \mathcal{D}, \mathbf{d}) \\
 & \text{dependencies} (p_1, \dots, p_n) = \\
 & \text{let } \mathcal{U} = \emptyset \\
 & \quad \mathcal{D} = \emptyset \\
 & \quad (\mathcal{U}, \mathcal{D}, \mathbf{d}_1) = \text{dependencies}_0 p_1 \mathcal{U} \mathcal{D} \\
 & \quad \quad \vdots \\
 & \quad (\mathcal{U}, \mathcal{D}, \mathbf{d}_n) = \text{dependencies}_0 p_n \mathcal{U} \mathcal{D} \\
 & \text{in} \\
 & (\{ p_1, \dots, p_n \}, \mathbf{d}_1 \cup \dots \cup \mathbf{d}_n)
 \end{aligned}$$

As an example of how the algorithm works, consider p_5 in the figure above. The instructions that p_5 depends on according to rule 1 are:

$$\{\mathcal{D}\phi \mid \phi \in \llbracket p_5 \rrbracket_{\text{used}}\} = \{\mathcal{D}(\mathbf{r3}), \mathcal{D}(\mathbf{r1})\} = \{p_2, p_3\}.$$

The instructions that p_5 depends on according to rule 2 are:

$$\{\mathcal{U}\phi \mid \phi \in \llbracket p_5 \rrbracket_{\text{defd}}\} = \{\mathcal{U}(\mathbf{r4})\} = \emptyset.$$

The instructions that p_5 depends on according to rule 3 are:

$$\{\mathcal{D}\phi \mid \phi \in \llbracket p_5 \rrbracket_{\text{defd}}\} = \{\mathcal{D}(\mathbf{r4})\} = \emptyset.$$

Thus the edges returned by $dependencies_0(p_5, \mathcal{U}, \mathcal{D})$ are $\{(p_2, p_5), (p_3, p_5)\}$.

Loads and stores

There are other dependencies than register dependencies. It must be considered when it is legal to swap loads and stores. Obviously, it is unsafe in general to swap a load and a store instruction, since they might reference the same memory location. It is also unsafe to swap two store instructions because one store may overwrite the other causing later loads from this memory location to result in the wrong value. Thus it is only safe to rearrange loads, subject to the same restrictions as there are on all other instructions. This is elegantly handled by the scheduling algorithm by considering (all of) the memory as a pseudo-register \mathbf{m} . This pseudo-register is destroyed by a store, and used by a load. E.g.,

$$\llbracket \text{ldw } 0(\%sr0, \%r1), \%r3 \rrbracket_{\text{used}} = \{\mathbf{r1}, \mathbf{m}\}$$

$$\llbracket \text{stw } \%r24, 4(\%sr0, \%r30) \rrbracket_{\text{defd}} = \{\mathbf{m}\}.$$

Two loads can be swapped if the rules permit, but a load and a store can never be swapped because of rules 1 and 2. Rule 3 will ensure that all stores in the scheduled program will appear in the same order as in the original program.

Some instructions may set processor flags besides defining a register. For example, an add instruction may set an overflow flag, on which succeeding instructions depend. This can be handled by regarding flags as pseudo-registers. Thus, an instruction may destroy several registers.

Designing a heuristic for the PA-RISC

This and the following section covers the specifics concerning scheduling PA-RISC code.

The basic principle for the scheduling heuristic is to try to schedule instructions that may cause interlocks as early as possible. This gives a better chance of finding an instruction that will not interlock with the one just scheduled.

On the PA-RISC, interlocks occur when (Andersen, 1995):

- an instruction loads a register, and the next instruction uses that register as a source
- a load immediately succeeds a store
- a store immediately succeeds a store
- a load is followed by an arithmetic/logical instruction or a load/store with address modification.

This leads to the following heuristic for choosing the next instruction:

```

if the last instruction was a load then
    if there are non-interlocking candidates then
        choose a non-interlocking candidate, preferably a load
    else
        choose an interlocking candidate, preferably a load
else
    if the last instruction was a store then
        try to schedule a non-load/store instruction
    else
        choose any candidate, preferably a load or a store.

```

If there are several equally good candidates, the candidate with most successors in the dependency graph should be chosen.

Skipping instructions

On the PA-RISC, most instructions can conditionally skip the next instruction. Consider the program fragment

```

p1 : add,=  %r1,%r2,%r3
p2 : add    %r3,%r3,%r4.

```

Here p_1 skips p_2 if $r1$ contains the same value as $r2$. What registers are destroyed by this sequence? If p_2 is executed, $r4$ is destroyed. If p_2 is not executed, $r4$ is not destroyed. Also, the instruction scheduler must not move p_2 away from p_1 , as this could result in another instruction being skipped. We solve the problem by treating p_1 and p_2 as one compound instruction. The set of registers destroyed by a compound instruction is the union of the registers destroyed by each instruction. The same holds for set of registers used by a compound instruction.

There can be chains of skip instructions. These are handled by treating the whole chain as one compound instruction.

10 Implementation notes

Everything in the report has been implemented. The modules that we have written from scratch constitute approximately 13000 lines. This chapter gives some correspondences between the report and the code and lists some differences between the report and the code.

10.1 The correspondence between the report and the code

In some respects, the implementation is quite as the report. Here is the SML code for $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$ (p. 104)

```
| ra (LET(lvar,info_lvar,e1,e2,info)) d =
  let
    val (w2,w2') = get_w_annotated_on e2
    val (d,f1_opt,b1) = ra e1 d
    val (d,fx) = def f1_opt (Lvar lvar) d w2
    val (d,f2_opt,b2) = ra e2 d
    val (d,p) = spilled (Lvar lvar) d
    val b = fn f => b1 fx |||| preserve (Lvar lvar) fx p (b2 f)
  in
    (d,f2_opt,b)
  end
```

LET(lvar,info_lvar,e1,e2,info) corresponds to `let $x = e_1$ in e_2` . Every construct has an “info” field, which is a reference containing annotations. E.g., the ω -information ω_2 and ω'_2 annotated on e_2 can be accessed from the info field on **e2** with `get_w_annotated_on e2`. Phases that are presented in the report as translations from one language to another annotated language are often implemented as phases that update the info fields of the source expressions. The **f**-variables are ϕ 's; **d** is δ (the ε -environment is implemented with global references); **def** is $\llbracket \cdot \rrbracket_{\text{def}}$; **spilled** is $\llbracket \cdot \rrbracket_{\text{kill}}$; **b** is β ; **||||** is $;$; etc.

10.2 Some deviations from the report in the implementation

Polymorphic equality and tagging

Polymorphic equality allows the programmer to compare arbitrary data objects for equality if they do not contain functions. The function

$$\text{fn } a => \text{fn } b => a=b$$

could be applied to any type that allows polymorphic equality. If we want the same code to be able to work for any type, it is not enough to represent all types uniformly: Two integers are equal if their representation as words are equal, but to compare tuples one cannot compare their representation

as words, the actual representations in memory must be compared. Thus the code for the polymorphic equality function must be able to see whether it is integers or tuples it is comparing, i.e., data must be tagged in our implementation. Tags are normally also needed by the garbage collector; in our implementation, the sole reason for having them is polymorphic equality.

According to the report, `()`, which has type `unit`, need not be represented explicitly, because there is no built-in operation with `unit` in its input type. This is not the case in the implementation, for the built-in polymorphic equality function may compare values of type `unit`: `op = : unit * unit -> bool`. Still, this ought not to be a problem, for there is only one value of type `unit`, and hence, `=` will always return `true` when applied to `unit`-values. Therefore, it should be of no concern what the exact value of any word representing `()` is. Unhappily, this is not so; because `=` is polymorphic, it does not know whether it is applied to a `unit`-value, and it must always inspect the representations of the values that are compared. This implies that `()` must have a fixed representation. In this presentation, the code for `e1 := e2` just lets some arbitrary garbage remain in its destination register; in the implementation, it must explicitly put the value representing `()` into the destination register.

(If `unit` did not allow polymorphic equality, there would be less trouble. On the other hand, one single instruction suffices to ensure that expressions of type `unit` return the right value.)

Storage mode

The source language in the implementation has “storage mode” annotations (p. 12) (Birkedal et al., 1996). Sometimes, allocation can safely be made at the “bottom” of the region (i.e., previously allocated data in the region can be overwritten), and sometimes it must be made “at the top” of the region. To keep things simple, these annotations have not been described in the report, but we have implemented the proper translation of them. The storage modes complicate the function $\llbracket \cdot \rrbracket_{\text{ra-at}}$ (which makes the code to allocate in a region, section 8.6) and puts restrictions on the order allocations must be done in. Notably, it means that the code for unary constructor application as presented in the report (p. 177) would be wrong in the implementation (and at a point, it was): The memory for the constructed value that `c1 e2 at ρ` evaluates to must be allocated *after* `e2` has been evaluated, or else the storage mode annotations will be wrong. (To the reader who understands storage modes: In the expression `::(1, ::(2, nil))` the storage mode for `nil` may indicate that the region `nil` and the `::`'s are stored in can be overwritten. If the allocations for the `::`'s are performed before the allocation for `nil`, the latter will overwrite the former allocations.)

Exceptions

The representation of exceptions in the implementation also comprises the name of the exception constructor (i.e. a string). This is necessary to report

the name of an exception that escapes to the top level.

The region inference currently puts all exceptions in a *global region*, i.e., a region that will be allocated throughout the evaluation of the whole program. This means that memory allocated for exceptions will not be deallocated before the program terminates. To alleviate this problem, the code for `exception a in e2` and the code for `ā at ρ` differ in the implementation from the description in the report: The code for `exception a in e2` does the necessary allocation such that the code for `ā at ρ` does not have to allocate anything. This gives a reduction in memory consumption for most programs, because the first kind of expressions usually are evaluated only once, while the latter often are evaluated many times during execution. The code for `ā1 e2 at ρ` still allocates memory; hence programs using unary exceptions will still have bad memory behaviour.

Reals

On the PA-RISC, reals must be aligned when stored in memory. Elsmann and Hallenberg (1995) ensure that this is the case by checking addresses at run-time. This can be done at compile-time, and it should be especially easy in our compiler, as it already keeps track of the stack pointer. But to limit our job, we have not implemented reals. Extending the register allocation to reals might also be straightforward.

Etc.

In the implementation, `letregion` binds a sequence of region variables, and not just one. This is dealt with as if it were a series of nested `letregion`-expressions.

A peep-hole optimisation collapses adjacent $\phi := \phi \pm \iota$ -instructions. These can be generated when, e.g., more than one known-size regions are allocated in a row, or several spilled values are deallocated right after each other.

11 Assessment

We assess our compiler in two ways: we compare the standard configuration of our compiler with other configurations and with two other compilers: SML/NJ, the best readily available SML compiler known to us, and the existing ML Kit (which we will call `KAM`), i.e., the same compiler as ours except for the back end.

Section 11.3 compares the speed of the code generated by the three compilers. Section 11.4 tries to assess the significance of different ingredients in the inter-procedural register allocation. It examines the effect of implementing several-argument functions efficiently, and of using inter-procedural information in the register allocation. Section 11.5 considers the per-function part of the register allocation. Section 11.6 measures how important the number of registers is. Section 11.7 looks at the effect of duplicating code to avoid jumps. Section 11.8 evaluates the effect of instruction scheduling. Section 11.9 is a case study of the translation of the Fibonacci function `fib` that, among other things, illustrates a deficiency with the producer-saves store code placement strategy. Section 11.10 investigates memory consumption.

11.1 How the measurements were performed

`KAM` also translates to PA-RISC. It uses a graph-colouring register allocation that works on extended basic blocks; does copy propagation; and uses the same instruction scheduler as our compiler.³

We compare with version 0.93 of SML/NJ. There are more recent versions. Notably, the previously discussed (p. 36), successful closure representation analysis (Shao and Appel, 1994) is not part of version 0.93. This is perhaps fair, as we do not have a closure representation analysis either.⁴

In the tables, the first column is the baseline (always 1.00), and the other columns are normalised with respect to that column. E.g., in figure 42, our run-time of `kfb` normalised to SML/NJ is 0.80. Hence, if the execution time of `kfb` in SML/NJ is 10s, the execution time in our compiler is $0.80 \cdot 10\text{s} = 8\text{s}$. The first column of a table normally also gives the unnormalised data, e.g., the run-time in seconds. The final row in a table gives the geometric mean (geometric because we use normalised results (Fleming and Wallace, 1986)). In a column, \smile marks the best result and \smile the worst.⁵

³We use version 22s of the ML Kit except for the optimiser which is version 25s extended with an `uncurry` phase to enable more functions to be converted to functions of several arguments in our compiler. When we compare with `KAM`, we have used a version (28g) that should essentially be like version 22s but with the same optimiser as in our compiler.

⁴To increase SML/NJ's possibilities of optimising, we have put each benchmark in a structure and a `let`-expression and we have included the code for the built-in functions (e.g., `@`). A signature is imposed on the structure so that the only function visible from the outside of the structure is the benchmark function.

⁵The experiments were run on an unloaded HP 9000/C100 with 256MB RAM (called `freja`). Timing results is the minimum sum of the “user” and “system” time as measured by Unix `time` after running the benchmark thrice.

11.2 Benchmark programs

The following benchmark programs have been used in all experiments. Except for **k**kb****, and **life**, the benchmarks are toy programs. Benchmarks should be real programs to be worth much, but time forbids doing better.

1. *Real programs.*

kkb	Knuth-Bendix completion, improved for region inference by Mads Tofte, i.e., kkb has been written in certain ways that makes its memory behaviour better in a region inference based compiler (Tofte, 1995, Bertelsen and Sestoft, 1995). Profiling kkb (in SML/NJ), shows that it uses time in many different functions, which should increase its value as a benchmark. It uses exceptions. KAM will not compile kkb when the uncurrying optimisation is turned on, so all measurements with kkb using KAM have been obtained with this optimisation switched off. This probably gives our kkb a benefit.
life	The game of life implemented using lists and improved for region inference. Profiling life shows that it uses half its time in a single function.

2. *Non-sensical programs believed to benefit from inter-procedural register allocation.*

appel	Function applications in a row, and simple arithmetic, e.g.: <pre>fun f (g,h,u,v,w) = let val x = g(h,u,v) val y = g(h,x,w) val z = g(h,y,x) in x+y+z+v+1 end</pre>
bappel	More of that.
ip	A seven-functions-deep call graph within a loop.
plusdyb	A three-functions-deep call graph within a loop.

3. *Programs that make many calls.*

ack	The Ackermann function, a multiple-argument function.
tak	The Takeuchi function, a multiple-argument function: <pre>fun t(x,y,z) = if x<=y then z else t(t(x-1,y,z),t(y-1,z,x),t(z-1,x,y))</pre>
fib	The Fibonacci function.

4. *Programs designed to test specific things.*

bul	Should benefit from the short-circuit translation of Boolean expressions. Basically, it is a loop containing the expression <pre> if (x<y andalso (case z mod 4 of 0 => true 1 => false 2 => a mod 2 = 1 _ => a<n)) orelse ae<oe then a+1 else a-1 </pre>
fri	A function that uses its free variables many times. It should benefit because we allocate a free variable to a register once it has been fetched from the closure whereas KAM fetches it each time it is used.
handle	Introduce handlers often but raise exceptions only exceptionally.
raise	Raise a lot of exceptions.

5. *Miscellaneous.*

reynolds	Build a big balanced binary tree and traverse it. Designed to exhibit good behaviour with region inference (Birkedal et al., 1996).
ryenolds	As reynolds , but changed in a way that gives bad memory performance with region inference.
church	Convert integers to church numerals; multiply and take the power of them; and convert the result back to integers. Many function applications, and many fetches of free variables from a closure.
foldr	Build and fold a big constant list.
msort	Merge sort.
qsort	Quick sort.
iter	Apply the following function to different functions: <pre> fun iter(x,p,f) = let fun h(a,r) = if p(a,r) then a else h(f(a),a) in h(x,1) end </pre>

Since we want to assess the inter-procedural register allocation and not region inference, many of the benchmarks do not use much memory.

We have two major reservations concerning the experiments below: almost all benchmarks are toy programs, some of which are designed to make our register allocation work particularly well. Also, the timing of the benchmarks may be inaccurate; we have observed fairly large fluctuations between two runs of the same benchmarks. The number of decimals is not an indication of the accuracy of the measurements.

11.3 Speed

	(i)		(ii)	(iii)	(iv)	(v)
	SML/NJ		KAM	WE		KAM
k kb	20.41 s	1.00	1.49	0.80	1.00	~0.54
life	48.24 s	1.00	0.64	0.36	1.00	0.56
appel	16.11 s	1.00	0.78	0.64	1.00	0.83
bappel	27.76 s	1.00	0.83	0.67	1.00	0.81
ip	24.42 s	1.00	0.56	0.36	1.00	0.63
plusdyb	37.92 s	1.00	0.45	0.38	1.00	0.84
ack	16.87 s	1.00	0.76	0.59	1.00	0.78
fib	98.58 s	1.00	0.44	0.44	1.00	1.00
tak	31.63 s	1.00	0.96	0.64	1.00	0.67
bul	35.78 s	1.00	0.43	~0.33	1.00	0.78
fri	10.21 s	1.00	1.17	0.77	1.00	0.66
handle	60.13 s	1.00	0.55	0.46	1.00	0.85
raise	31.28 s	1.00	0.84	0.62	1.00	0.73
ryenolds	27.32 s	1.00	0.57	0.54	1.00	0.95
reynolds	28.04 s	1.00	~0.35	0.40	1.00	~1.14
church	39.79 s	1.00	0.95	0.85	1.00	0.90
foldr	47.14 s	1.00	0.67	0.51	1.00	0.77
msort	9.28 s	1.00	1.25	0.78	1.00	0.63
qsort	19.01 s	1.00	1.45	~1.05	1.00	0.73
iter	15.77 s	1.00	~1.67	0.94	1.00	0.56
<i>mean</i>		1.00	0.76	0.57	1.00	0.75

Fig. 42. Run-time of the compiled benchmarks with the standard configuration of the three compilers.

(i)–(iii) are normalised to SML/NJ. For SML/NJ, raw figures are also given. Comparing with SML/NJ, we do worse only on **qsort**. This is probably because SML/NJ’s garbage collection is better at handling this benchmark than region inference, for also KAM does worse than SML/NJ on **qsort**. On average, our benchmarks run in 0.57 the time of SML/NJ.

(iv)–(v) are normalised to KAM. Comparing with KAM, we do worse only on **reynolds**. On average, our benchmarks run in 0.75 the time of KAM. Strangely, it is not on the inter-procedural benchmarks that we see the greatest speed-up: the relative run-times of **appel**, **bappel**, and **plusdyb**, are all larger than the mean.

11.4 The importance of the different ingredients of the inter-procedural register allocation

	(i) WE, normal	(ii) no s.a.	(iii) no un- curry	(iv) no s.a., no un- curry	(v) uniform l.c.'s	(vi) uniform l.c.'s, caller saves	(vii) KAM
k k b	16.28 s 1.00	1.09	1.12	1.20	1.06	1.08	~1.86
l i f e	17.40 s 1.00	1.00	1.08	1.08	1.21	1.35	1.78
a p p e l	10.38 s 1.00	1.00	0.99	1.00	1.01	1.15	1.20
b a p p e l	18.64 s 1.00	1.03	1.00	1.03	1.00	1.14	1.23
i p	8.70 s 1.00	1.08	1.37	1.42	1.03	1.07	1.58
p l u s d y b	14.41 s 1.00	1.13	1.37	1.37	1.01	0.98	1.19
a c k	10.00 s 1.00	1.16	1.00	1.15	1.00	1.00	1.29
f i b	43.31 s 1.00	1.00	1.00	1.00	~0.90	~0.92	1.00
t a k	20.32 s 1.00	1.46	1.00	1.46	0.92	0.98	1.49
b u l	11.98 s 1.00	1.01	0.99	1.01	0.99	1.02	1.29
f r i	7.85 s 1.00	1.00	1.00	1.01	1.02	1.06	1.52
h a n d l e	27.93 s 1.00	1.11	~1.51	1.51	1.01	1.01	1.18
r a i s e	19.29 s 1.00	1.01	1.00	1.01	1.02	1.02	1.37
r y e n o l d s	14.86 s 1.00	1.13	1.02	1.16	1.01	1.01	1.05
r e y n o l d s	11.26 s 1.00	1.00	~0.96	~0.96	1.05	1.06	~0.87
c h u r c h	34.00 s 1.00	1.00	0.99	0.99	1.06	1.12	1.11
f o l d r	24.22 s 1.00	1.08	1.05	1.11	1.07	1.07	1.30
m s o r t	7.26 s 1.00	1.08	1.00	1.08	1.06	1.05	1.60
q s o r t	20.02 s 1.00	1.01	1.00	1.01	1.12	1.11	1.38
i t e r	14.76 s 1.00	~1.57	1.00	~1.57	~1.24	~1.40	1.78
<i>mean</i>	1.00	1.09	1.06	1.14	1.04	1.07	1.33

Fig. 43. Run-time with different parts of the register allocation disabled.

(i) The standard configuration of our compiler.

(ii) Disable several-argument functions. This increases the mean run-time to 1.09, and **k****k****b** confirms that.

(iii) Disable uncurrying. The uncurry phase is not part of our work; we try disabling it to see whether this inhibits the conversion to several-argument functions.

(iv) Disable both uncurrying and several-argument functions. Apparently there is little synergism between uncurrying and conversion to several-argument functions: (iv) is never more than the sum of (ii) and (iii).

(v) Use a uniform linking convention, i.e., use the same registers at all calls to pass parameters to functions. This increases the mean run-time to 1.04, and **k****k****b** confirms that.

(vi) Use a uniform linking convention and total caller-saves. There is a further increase to mean 1.07, which **k****k****b** confirms. Thus, using flexible linking conventions appears to be slightly more important than using flexible conventions for register saving. This is contrary to what we would expect, as the latter should avoid saves of registers while the former should only avoid moves (section 5.6).

Notice it is a bit unfair to compare our flexible sets of callee-save/caller-save registers with a total caller-saves convention, for an intra-procedural register allocator could probably benefit much from having both some caller-save and some callee-save registers. However, it was easiest to hack our compiler to use a total caller-save convention.

Offhand, several-argument functions (*ii*) seem more important than inter-procedural information (*vi*). But a part of the benefit from using several-argument functions may be indirect: several-argument functions should give the inter-procedural register allocation greater opportunity for doing well. I.e., several-argument functions might not be worth as much if the register allocation was not inter-procedural. It would be an interesting experiment to turn off both at the same time, i.e., combine (*ii*) and (*vi*): if this gives a run-time less than the “sum” 1.16 of 1.09 and 1.07, several-argument functions and inter-procedural information each increase the importance of the other. To some extent figure 44 (*iv*) carries out this experiment: there the two are turned off *and* our compiler is maimed in other ways, but the run-time only decreases to 1.15, which is less than 1.16. This suggests that there is synergism between the two.

(*vii*) KAM.

11.5 The per-function part of the register allocation

	(i)		(ii)	(iii)	(iv)	(v)
	normal <i>choose</i>		$\omega = \emptyset$	same always	WE, intra- procedural version	KAM
kkb	16.28 s	1.00	1.13	1.34	1.15	\sim 1.86
life	17.40 s	1.00	\sim 1.73	2.38	1.34	1.78
appel	10.38 s	1.00	1.29	1.70	1.19	1.20
bappel	18.64 s	1.00	1.25	1.62	1.15	1.23
ip	8.70 s	1.00	1.56	\sim 3.02	1.19	1.58
plusdyb	14.41 s	1.00	1.20	1.47	1.22	1.19
ack	10.00 s	1.00	1.00	1.90	1.16	1.29
fib	43.31 s	1.00	\sim 0.94	1.26	\sim 0.96	1.00
tak	20.32 s	1.00	1.05	1.49	1.61	1.49
bul	11.98 s	1.00	1.09	1.36	1.06	1.29
fri	7.85 s	1.00	1.32	2.88	1.06	1.52
handle	27.93 s	1.00	1.09	1.37	1.10	1.18
raise	19.29 s	1.00	1.12	1.10	1.01	1.37
ryenolds	14.86 s	1.00	1.03	1.21	0.99	1.05
reynolds	11.26 s	1.00	1.02	\sim 1.08	0.99	\sim 0.87
church	34.00 s	1.00	1.05	1.28	1.03	1.11
foldr	24.22 s	1.00	1.23	1.54	1.11	1.30
msort	7.26 s	1.00	1.31	1.57	1.11	1.60
qsort	20.02 s	1.00	1.37	1.70	1.04	1.38
iter	14.76 s	1.00	1.36	1.80	\sim 2.01	1.78
<i>mean</i>		1.00	1.19	1.59	1.15	1.33

Fig. 44. Run-time with different heuristics for choosing registers. (i) The normal heuristic for choosing registers (section 6.7). (ii) Same as (i), but ignoring the ω -information. (iii) Always choose the first register not in $\hat{\phi}$ according to some (arbitrary) given order. This (roughly) gives a lower bound on how bad a heuristic can do, as it does the worst possible: always choose the same register, except when two values *must* be in different registers (e.g., because they will be added together).

It seems the heuristic is quite good: (i) is much better than (ii), and it seems that (ii) is not unreasonably bad, for it is much better than the “lower bound”, (iii). Take care not to draw rash conclusions, though; setting $\omega = \emptyset$ might still not be a fair way to handicap the heuristic.

Comparing with graph colouring. (iv) Attempting to compare our per-function register allocation with graph colouring, we have tried giving WE the same conditions as KAM: uniform calling convention, total caller-saves convention, no several-argument functions, no instruction scheduling, no code duplication, and restricting the set of available registers to approximately what KAM uses. (v) KAM. We do better, and it is tempting to conclude that our per-function register allocation is better than graph colouring, but the experiment does not sustain a conclusion that radical, as there are other differences between the two compilers than the per-function register allocation algorithm (e.g., the treatment of exceptions, Boolean expressions, free variables, saving across function calls).

11.6 The importance of the number of registers

We have tried to give the register allocator fewer than the 28 registers available. One of the goals of inter-procedural register allocation is to utilise the many registers better. Thus, if our inter-procedural register allocation is worth anything, one should expect performance to degrade when there are fewer registers. With an *intra*-procedural register allocation, which cannot exploit the many registers as well, performance should degrade less.

We have decided what sets Φ of registers to try in the following way: The most frequently used heavy-weight instructions $\phi_1 := \text{at } \phi_2 : l$, $\phi := \text{letregion}$, and endregion destroy specific sets of registers, $\hat{\phi}_{\text{at}}$, $\hat{\phi}_{\text{letregion}}$, and $\hat{\phi}_{\text{endregion}}$, respectively. Since the registers in $\phi_{\text{heavy}} = \hat{\phi}_{\text{at}} \cup \hat{\phi}_{\text{letregion}} \cup \hat{\phi}_{\text{endregion}}$ are destroyed frequently (in programs that allocate) they will probably be used for values with short live ranges. Therefore, we divide the experiments in those that include ϕ_{heavy} , and those that do not. The set ϕ_{rest} is the set of registers not in ϕ_{heavy} .

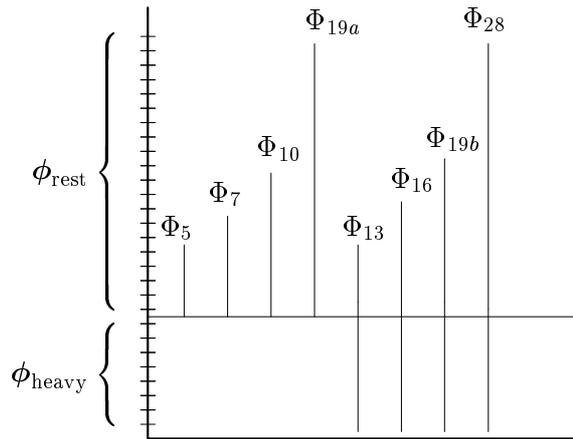


Fig. 45. Varying the set Φ of registers available to the register allocator. The markings on the vertical axis are the 28 available registers. Registers below the thin horizontal line are in ϕ_{heavy} ; those above the line are in ϕ_{rest} . The vertical lines are experiments. They are named after the number of registers they contain. There are two different experiments with 19 registers. For technical reasons, no experiment may have less than 5 registers from ϕ_{rest} .

	Φ_5	Φ_7	Φ_{10}	Φ_{19a}	Φ_{13}	Φ_{16}	Φ_{19b}	Φ_{28}	
kkb	1.28	1.22	1.11	0.99	1.10	1.04	1.03	1.00	16.28 s
life	~ 1.88	1.72	1.05	1.03	1.08	1.03	1.01	1.00	17.40 s
appel	1.43	1.32	1.10	1.01	1.11	1.05	1.04	1.00	10.38 s
bappel	1.37	1.29	1.10	1.02	1.11	1.06	1.05	1.00	18.64 s
ip	1.26	1.19	1.05	0.98	1.04	1.03	1.00	1.00	8.70 s
plusdyb	1.28	1.25	0.97	1.00	0.95	0.96	~ 0.99	1.00	14.41 s
ack	1.28	1.27	1.00	1.00	0.99	0.93	1.00	1.00	10.00 s
fib	1.23	1.27	0.93	0.95	~ 0.86	~ 0.90	1.00	1.00	43.31 s
tak	1.41	1.42	0.97	0.96	0.95	0.92	1.00	1.00	20.32 s
bul	1.26	1.17	1.14	~ 1.12	1.07	1.01	1.01	1.00	11.98 s
fri	1.27	1.14	1.05	1.02	1.05	1.02	1.01	1.00	7.85 s
handle	1.05	1.03	1.00	1.01	1.02	1.02	1.00	1.00	27.93 s
raise	1.10	1.09	1.05	1.02	1.07	1.04	1.03	1.00	19.29 s
ryenolds	~ 0.92	~ 0.92	~ 0.84	~ 0.87	1.03	1.00	1.00	1.00	14.86 s
reynolds	1.16	1.14	0.95	0.97	1.07	0.97	1.00	1.00	11.26 s
church	1.28	1.12	0.96	0.98	1.04	0.98	1.00	1.00	34.00 s
foldr	1.36	1.33	1.11	1.05	1.07	1.01	1.00	1.00	24.22 s
msort	1.44	1.30	1.07	1.01	1.13	1.13	1.03	1.00	7.26 s
qsort	1.66	1.52	~ 1.31	1.03	~ 2.73	1.21	1.08	1.00	20.02 s
iter	1.85	~ 1.88	1.16	0.99	1.38	~ 1.42	~ 1.17	1.00	14.76 s
<i>mean</i>	1.32	1.26	1.04	1.00	1.11	1.03	1.02	1.00	

Fig. 46. Run-time with the experiments from figure 45 normalised to Φ_{28} .

As is to be expected, the gain from adding an extra register to the set of available registers decreases as the number of registers increases.

The mean of experiment Φ_{19a} is 1.00, i.e., the benchmarks run just as well with ϕ_{heavy} added to the set of registers as without. This suggests the effort we have invested in making many registers participate on equal terms in the register allocation is somewhat wasted. We could simply have reserved ϕ_{heavy} for the run-time system and not allowed these registers to participate in the register allocation.

Since the registers in ϕ_{heavy} are destroyed by heavy-weight instructions, we expect that, e.g., experiment Φ_{19a} is faster than Φ_{19b} , even though the same number of registers are available. This is the case (1.00 compared to 1.02). The same effect can be seen by comparing Φ_{10} and Φ_{13} (1.04 compared to 1.11): experiment Φ_{13} has more registers, but many of them are destroyed by the heavy-weight instructions, and consequently, the benchmarks run faster in Φ_{10} . Naturally, it is the benchmarks that allocate, and thus use the heavy-weight instructions, that experience decreasing performance from Φ_{10} to Φ_{13} : e.g., **ryenolds**, **reynolds**, **iter**, and especially **qsort** (**kkb** is an exception, or perhaps it does not allocate much?). The benchmarks that do not allocate generally improve from Φ_{10} to Φ_{13} . In Φ_{16} , we get a mean performance which is the same as in Φ_{10} ; i.e., the numbers say: 10 registers that are not destroyed by the heavy-weight instructions is as good as 8 registers that are plus 8 that are not. In other words, the registers in ϕ_{heavy} are not worth much.

On the other hand, the difference between experiments Φ_5 and Φ_{13} is only that ϕ_{heavy} has been added, and this gives an improvement in normalised run-time from 1.32 to 1.11, which is hardly negligible. The extra registers for short live ranges are welcome when there are few registers in all.

In Φ_5 , the programs take 32% longer to run. The big performance increase is gained when going from 7 to 10 registers: already then, the benchmarks are only 4% slower than with all registers.

Surprisingly, **fb** seems to run fastest using 13 registers; **ryenolds** runs quite a bit better with 10 than with 28 registers; and it is even better with only 5 than with 28 registers. Surprises should be expected, since we are using heuristics.

Comparing experiments Φ_{19a} and Φ_{28} , we would expect the benchmarks **appel**, **bappel**, **ip**, and **plusdyb** that are designed to benefit from inter-procedural register allocation to be the ones that improve their performance the most when going from 19 to 28 registers. This is not the case: the improvement from Φ_{19a} to Φ_{28} on these benchmarks is small.

11.7 Linearising the code

The algorithm duplicates a basic block to avoid a jump if the number of instructions in the basic block is less than the *duplication limit*. We have tried setting the duplication limit to 0, 4, 10, 23, 42, and 1000. Heavy-weight instructions count as some fixed amount of light instructions; e.g., endregion counts as 9.

	(i)		(ii)	(iii)					
				0	4	10	23	42	1000
k kb	1324	1.00	0.43	0.00	0.03	0.12	0.22	0.31	0.45
life	459	1.00	0.34	0.00	0.03	0.10	0.20	0.25	~0.49
appel	21	1.00	0.43	0.00	~0.00	~0.00	~0.00	~0.00	~0.24
bappel	35	1.00	0.34	0.00	~0.00	0.06	0.06	0.06	0.43
ip	40	1.00	0.40	0.00	0.03	0.07	0.07	0.07	0.30
plusdyb	15	1.00	0.27	0.00	~0.00	0.07	0.20	0.20	0.27
ack	18	1.00	0.44	0.00	0.06	0.11	0.17	0.28	0.28
fib	16	1.00	0.44	0.00	~0.00	0.12	~0.25	0.25	0.25
tak	25	1.00	0.44	0.00	0.04	0.08	0.16	0.16	0.32
bul	41	1.00	~0.46	0.00	0.02	~0.17	0.22	0.29	0.39
fri	43	1.00	0.40	0.00	~0.00	0.05	0.14	0.28	0.40
handle	58	1.00	0.40	0.00	~0.00	0.14	0.19	0.21	0.38
raise	60	1.00	0.38	0.00	0.03	0.15	0.17	0.17	0.37
ryenolds	42	1.00	0.43	0.00	0.02	0.10	0.21	0.29	0.36
reynolds	47	1.00	0.40	0.00	0.02	0.11	0.21	~0.32	0.32
church	115	1.00	~0.17	0.00	0.01	0.07	0.20	0.28	0.33
foldr	44	1.00	0.39	0.00	~0.00	0.11	0.23	0.25	0.34
msort	196	1.00	0.42	0.00	0.04	0.12	0.20	0.30	0.46
qsort	80	1.00	0.44	0.00	0.04	0.09	0.14	0.23	0.40
iter	47	1.00	0.43	0.00	~0.09	0.15	0.19	0.26	0.28

Fig. 47. *Effect of ordering basic blocks and duplication on number of jumps.* (i) The number of basic blocks in the program.

(ii) The number of basic blocks that were put after a basic block that jumps to it, i.e., the number of jumps eliminated by ordering basic blocks. Ordering basic blocks succeeds in placing between 0.34 and 0.44 of all basic blocks after a basic block that jumps to it (except for three benchmarks).

(iii) The number of basic blocks that were duplicated to eliminate a jump, with the duplication limit set to 0, 4, 10, 23, 42, and 1000 instructions, respectively. With the duplication limit set to 4 only few extra jumps are avoided by duplication of basic blocks. With the duplication limit set to 10, up to 0.17 (**bul**) of the jumps at the end of basic blocks are avoided by duplicating basic blocks.

	0		4	10	23	42	1000
kkb	28068	1.00	1.00	1.03	~1.09	1.20	1.88
life	13951	1.00	1.00	1.00	1.04	1.08	2.39
appel	1026	1.00	1.00	1.00	1.00	1.00	1.57
bappel	1367	1.00	1.00	1.00	1.00	1.00	2.14
ip	1135	1.00	1.00	1.00	1.00	1.00	2.11
plusdyb	431	1.00	1.00	1.00	1.07	1.07	1.06
ack	519	1.00	1.00	1.01	1.04	1.15	1.15
fib	525	1.00	1.00	1.01	1.07	1.07	1.07
tak	599	1.00	1.00	1.01	1.03	1.03	1.31
bul	930	1.00	1.00	1.01	1.06	~1.30	1.81
fri	1366	1.00	1.00	1.00	1.05	1.19	~2.86
handle	2164	1.00	1.00	~1.04	1.05	1.06	1.53
raise	2441	1.00	1.00	1.02	1.02	1.02	1.54
ryenolds	1395	1.00	1.00	1.00	1.04	1.10	1.23
reynolds	1292	1.00	1.00	1.00	1.05	1.11	1.11
church	3271	1.00	1.00	1.01	~1.09	1.17	1.58
foldr	11032	1.00	1.00	1.00	1.01	1.01	~1.04
msort	4384	1.00	1.00	1.01	1.07	1.22	2.13
qsort	2197	1.00	1.00	1.00	1.03	1.13	2.48
iter	1119	1.00	1.00	1.00	1.03	1.08	1.16
<i>mean</i>		1.00	1.00	1.01	1.04	1.10	1.58

Fig. 48. *Effect of duplication on code size.* We use the number of lines in the `.s` file as a measure of the number of \mathfrak{B} -instructions. This should do, because we are studying the *increase* in code size. The duplication limit is set as in figure 47. Apparently, there is no code explosion; even with the duplication limit set unrealistically high, the code size maximally increases by a factor 2.86.

	0		4	10	23	42	1000
kkb	16.37s	1.00	1.00	0.97	0.98	0.96	0.96
life	17.91s	1.00	0.97	0.96	0.98	0.98	0.98
appel	10.18s	1.00	1.00	~1.01	~1.01	1.00	1.01
bappel	18.72s	1.00	1.00	1.00	1.00	1.00	0.99
ip	8.71s	1.00	1.00	1.00	1.00	1.00	0.97
plusdyb	14.00s	1.00	~1.01	0.99	0.99	0.99	0.95
ack	10.43s	1.00	~0.94	~0.89	~0.89	0.95	0.96
fib	41.37s	1.00	1.00	0.94	0.98	0.98	0.98
tak	20.23s	1.00	~0.94	0.92	0.91	~0.91	~0.93
bul	11.61s	1.00	1.00	1.00	~1.01	1.00	1.00
fri	7.91s	1.00	~1.01	1.00	1.00	1.02	1.00
handle	28.58s	1.00	~1.01	0.99	0.98	0.98	0.97
raise	19.49s	1.00	1.00	1.00	0.99	0.99	1.00
ryenolds	14.87s	1.00	1.00	1.00	0.97	1.03	1.02
reynolds	11.61s	1.00	1.00	~1.01	0.99	1.02	1.02
church	34.07s	1.00	~1.01	0.99	~1.01	1.00	1.00
foldr	24.46s	1.00	1.00	0.98	0.98	0.98	0.99
msort	7.30s	1.00	1.00	0.98	0.98	0.98	0.99
qsort	19.66s	1.00	0.99	~1.01	~1.01	~1.04	~1.05
iter	14.95s	1.00	1.00	0.99	0.97	0.97	0.97
<i>mean</i>		1.00	0.99	0.98	0.98	0.99	0.99

Fig. 49. *Effect of duplication on run-time.* Performance does not seem to degrade when the code size increases: e.g., at 42, the size of **kkb** is scaled with 1.20, at 1000 it is 1.88, but the performance is 0.96 in both cases.

The mean speed tops at the duplication limits 10 and 23 (it is 0.98). At these points, the code size (figure 48) has increased to a mere 1.01 and 1.04, respectively, and maximally increases to 1.09. (**kkb** tops at duplication limit 42, though, with the speed 0.96 and code size 1.20.)

11.8 Instruction scheduling

	WE, no sche- duling	WE
k k b	1.00	0.98
l i f e	1.00	0.99
a p p e l	1.00	0.99
b a p p e l	1.00	1.00
i p	1.00	1.00
p l u s d y b	1.00	0.93
a c k	1.00	0.91
f i b	1.00	~0.88
t a k	1.00	0.91
b u l	1.00	0.97
f r i	1.00	1.00
h a n d l e	1.00	~1.01
r a i s e	1.00	1.00
r y e n o l d s	1.00	1.00
r e y n o l d s	1.00	0.99
c h u r c h	1.00	0.99
f o l d r	1.00	0.99
m s o r t	1.00	0.99
q s o r t	1.00	1.00
i t e r	1.00	0.97
<i>mean</i>	1.00	0.97

Fig. 50. *Effect of instruction scheduling.* The scheduling gives an overall slight speedup. One reason that the speedup is relatively small may be that the generated code has a large percentage of load/store instructions. For example, approximately 65% of the instructions in the code for **k****k****b** are loads or stores.

11.9 An example

An example illustrates some of the problems with our register allocation. Consider **fib**:

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

<pre> fib: push ϕ_{24} ; push ϕ_{26} ; push ϕ_{25} ; if $\phi_{24}=0$ then fib3 else fib4 ; fib3: $\phi_{26} := 1$; goto fib7 ; fib4: if $\phi_{24}=1$ then fib5 else fib6 ; fib5: $\phi_{26} := 1$; goto fib7 ; fib6: $\phi_{23} := 1$; $\phi_{24} := \phi_{24} - \phi_{23}$; $\phi_{25} := \mathbf{fib1}$; goto fib ; fib1: push ϕ_{26} ; $\phi_{26} := m[\phi_{sp}-3]$; $\phi_{25} := m[\phi_{sp}-4]$; $\phi_{24} := 2$; $\phi_{24} := \phi_{25} - \phi_{24}$; $\phi_{25} := \mathbf{fib2}$; goto fib ; fib2: $\phi_{25} := \phi_{26}$; pop ϕ_{26} ; $\phi_{26} := \phi_{26} + \phi_{25}$; goto fib7 ; fib7: $\phi_{25} := m[\phi_{sp}-1]$; $\phi_{sp} := \phi_{sp} - 3$; goto ϕ_{25} </pre>	<pre> spill n spill clos spill ret 0Γ return 1 in ϕ_{26} 1Γ return 1 in ϕ_{26} fib(n-1) return to fib1 push result of fib(n-1) reload clos reload n fib(n-2) return to fib2 pop result of fib(n-1) return fib(n-1)+fib(n-2) in ϕ_{26} reload ret pop n, clos, and ret return. </pre>
---	---

Fig. 51. *The K-code for fib.* We use a so-called producer-saves strategy for placing store code: a spilled value is stored as soon as it has been produced (section 6.12). This is the earliest possible and has the disadvantage that values are sometimes stored unnecessarily. **fib** provides a pregnant example of this: the parameters to the function are live across the recursive calls and must hence be saved around these calls. According to the producer-saves strategy, this is done in the start of the code for **fib**. At leaf calls to the function (calls with argument 0 or 1), this storing is actually not necessary, and since half the calls at run-time are leaf calls, half the stores at run-time are superfluous. For **fib**, a callee-saves convention is better than our caller-saves convention, because so many of the recursive calls do not destroy registers. With earlier versions of our compiler, KAM did better on **fib** exactly because it always uses a callee-saves convention; apart from the placement of spill code, the code generated by the two compilers for **fib** was almost identical.

There should be no **clos** parameter for functions that have no free variables (e.g. **fib**). It should be straightforward to make this distinction in the calling convention.

Even though **clos** were needed, it need not be saved on the stack, for ϕ_{26} , the register holding it, is never destroyed by the code for **fib**. **clos** is nevertheless saved at each recursive call, as the compiler cannot know that ϕ_{26} will not be destroyed by the recursive calls to **fib**, because the processing of **fib** has not yet been completed. All conceivable solutions to this seem cumbersome.

The clumsy code labelled fib2 is caused by our way of saving temporary values.

fib	stwm	%r24, 4(%sr0, %r30)	spill n
	stwm	%r26, 4(%sr0, %r30)	spill clos
	stwm	%r25, 4(%sr0, %r30)	spill ret
	comib,	=,n 1, %r24, fib3	0Γ
(fib4)	comib,	=,n 3, %r24, fib5	1Γ
(fib6)	addil	l'fib1-\$global\$, %r27	put fib1 into %r25 (1 st part)
	ldi	3, %r23	fib(n-1)
	sub	%r24, %r23, %r24	
	ldo	r'fib1-\$global\$(%r1), %r25	put fib1 into %r25 (2 nd part)
	b	fib	
	ldo	1(%r24), %r24	
fib3	ldw	-4(%sr0, %r30), %r25	reload ret
(fib7)	ldo	-12(%r30), %r30	pop 3 words
	bv	%r0(%r25)	return
	ldi	3, %r26	... while returning 1 in %r26
fib5	ldw	-4(%sr0, %r30), %r25	reload ret
(fib7)	ldo	-12(%r30), %r30	pop 3 words
	bv	%r0(%r25)	return
	ldi	3, %r26	... while returning 1 in %r26
fib1	stwm	%r26, 4(%sr0, %r30)	push result of fib(n-1)
	ldi	5, %r24	fib(n-2)
	ldw	-16(%sr0, %r30), %r25	reload n
	ldw	-12(%sr0, %r30), %r26	reload clos
	sub	%r25, %r24, %r24	
	addil	l'fib2-\$global\$, %r27	put fib2 into %r25 (1 st part)
	ldo	r'fib2-\$global\$(%r1), %r25	put fib2 into %r25 (2 nd part)
	b	fib	
	ldo	1(%r24), %r24	
fib2	copy	%r26, %r25	pop result of fib(n-1)
	ldwm	-4(%sr0, %r30), %r26	+
	add	%r26, %r25, %r26	reload ret
	ldw	-4(%sr0, %r30), %r25	pop 3 words
(fib7)	ldo	-12(%r30), %r30	return
	bv	%r0(%r25)	... while returning result
	ldo	-1(%r26), %r26	

Fig. 52. Finally, the \mathfrak{P} -code for **fib**. Now integers are tagged: **1** represents 0, **3** represents 1, etc. Remember, $\phi_{sp} = \mathbf{r30}$. The basic blocks labelled **fib4** and **fib6** have been placed after a jump to them; and the one labelled **fib7** has been duplicated to avoid jumps to it.

Instruction scheduling has reordered instructions. E.g., the two **ldw**'s after **fib1** that reload **clos** and **n** have been exchanged (compare with figure 51) such that **n** will be ready when the **sub**-instruction needs it.

The instruction right after a jump instruction (**b** or **bv**) is executed “while” the jump is taken. An optimisation (made by Elsmann and Hallenberg (1995)) tries to take advantage of this. As can be seen, it has succeeded well on **fib**.

There are some obvious opportunities for peep-hole optimisations.

11.10 Memory consumption

	SML/NJ				KAM		WE	
	<i>tot.</i>	<i>res.</i>	<i>tot.</i>	<i>res.</i>	<i>tot.</i>	<i>res.</i>	<i>tot.</i>	<i>res.</i>
kkb	2244 k	2164 k	1.00	1.00	1.65	1.64	2.17	2.22
life	1824 k	1744 k	1.00	1.00	0.21	0.22	0.25	0.24
appel	1704 k	808 k	1.00	1.00	0.06	0.12	0.05	0.11
bappel	2228 k	1072 k	1.00	1.00	0.28	0.59	0.30	0.63
ip	1904 k	1032 k	1.00	1.00	0.25	0.46	0.40	0.75
plusdyb	1700 k	1248 k	1.00	1.00	0.05	0.07	0.05	0.07
ack	2220 k	2140 k	1.00	1.00	0.15	0.15	0.18	0.17
fib	1720 k	1640 k	1.00	1.00	0.05	0.05	0.05	0.06
tak	1708 k	1404 k	1.00	1.00	0.05	0.06	0.05	0.07
bul	1892 k	1156 k	1.00	1.00	0.05	0.09	0.05	0.08
fri	2136 k	1584 k	1.00	1.00	0.04	0.06	0.04	0.06
handle	1912 k	1832 k	1.00	1.00	0.05	0.06	0.05	0.05
raise	1728 k	1632 k	1.00	1.00	0.06	0.07	0.05	0.06
ryenolds	1660 k	1196 k	1.00	1.00	24.10	33.44	24.10	25.08
reynolds	1660 k	1296 k	1.00	1.00	0.06	0.08	0.05	0.07
church	1920 k	1840 k	1.00	1.00	7.29	7.61	7.29	7.07
foldr	1928 k	1848 k	1.00	1.00	0.88	0.92	0.86	0.87
msort	12000 k	12000 k	1.00	1.00	1.17	1.17	1.08	0.92
qsort	10000 k	10000 k	1.00	1.00	2.30	2.30	2.20	2.20
iter	1888 k	872 k	1.00	1.00	1.50	3.25	0.88	1.90
<i>mean</i>			1.00	1.00	0.36	0.36	0.27	0.35

Fig. 53. *Maximal memory consumption when the compiled program is run as seen by Unix **top**. There are two sizes (**man top**): *tot.*, “Total size of the process in kilobytes. This includes text, data, and stack.”, and: *res.*, “Resident size of the process in kilobytes. The resident size information is, at best, an approximate value.”*

We would expect the memory consumption of KAM and WE to be the same as both use region inference. For a discussion of memory behaviour with region inference, see (Birkedal et al., 1996).

Overall, our total memory consumption is slightly better than KAM’s (0.27 against 0.36 *tot.*). This is probably because we spill on the stack while KAM reserves a specific memory cell for each spilled local.

11.11 Conclusions

On average, our compiler compiles the (toy) benchmarks to code that runs in 0.57 of the time of the code generated by SML/NJ, and in 0.75 of the time of the code generated by another version of the ML Kit that uses graph-colouring intra-procedural register allocation.

What gives the speed up?

	decreases run-time by		
	<i>mean</i>	<i>max.</i>	<i>min.</i>
flexible linking conventions and caller-save/callee-save conventions (figure 43 (<i>vi</i>))	7%	29%	-9%
flexible linking conventions alone (figure 43 (<i>v</i>))	4%	20%	-11%
several-argument functions (figure 43 (<i>ii</i>))	8%	36%	0%
basic block duplication (figure 49)	2%	11%	-1%
instruction scheduling (figure 50)	3%	12%	-1%

If, e.g., basic block duplication is disabled and all other things are enabled, enabling basic block duplication as well decreases run-time by 2%. One cannot add figures to get the effect of the combination of two ingredients.

Conversion to several-argument functions and using inter-procedural information are clearly the most important ingredients. There are some indications that there is synergism between them. In comparison, Chow (1988) measures a reduction in the number of executed clock cycles when using inter-procedural information (and shrink wrapping) of about 3%. His experiment is, however, not directly comparable to ours (p. 73).

Basic block ordering will quite consistently place around 40% of the basic blocks after a basic block that jumps to it.

Basic block duplication is not the most important ingredient. On the other hand, it appears to be worth having, as there seems to be no problem with code explosion.

Instruction scheduling is slightly more important than basic block duplication.

The per-function part of our algorithm that uses the structure of the source program and not graph colouring to allocate registers in a function seems successful. The algorithm is a bit complicated, but at least we did succeed in inventing the register allocation for every construct in the source language E . It is a good sign that the method extends in a very nice way to encompass short-circuit translation of Boolean expressions. Comparisons of object code quality with a graph-colouring register allocator are in our favour (figure 44 (*iv*)), but this is not conclusive as the two register allocators are different in other aspects. There are also other hints that our non-graph-colouring heuristic is good at keeping values in registers (figure 44 (*ii*)). Graph colouring has completely conquered the world of register allocation

and it is a conceptually nice method, but it seems that other methods can compete in terms of efficiency.

11.12 Directions from here

It would be interesting to measure exactly how well a register allocator that is similar to ours except that it uses graph colouring as the per-function part of the algorithm would compete. The competitiveness of the inter-procedural part of our algorithm should be tested more carefully by comparing it with an intra-procedural register allocator that uses a split caller-save/callee-save convention instead of the total caller-saves convention that we have tried. An interesting experiment would be to see how much the inter-procedural register allocation would suffer from a less sophisticated closure analysis, e.g., one that only knows which function may be applied if it is an application of a specific, named function. Also, it would be interesting to see what could be won from using a smarter spill code placement strategy.

We have learned that it is very difficult to predict which optimisations will be effective. We guess the most important ones now are closure representation analysis (section 4.6) and data representation analysis (section 4.1): The register allocator cannot remove memory traffic due to accesses to free variables and accesses to the actual representations of data. Perhaps these optimisations should have been addressed before register allocation, thereby increasing the register pressure and thus the gain from register allocation. It should be feasible to integrate our register allocator with these two optimisations: closure representation analysis transforms free variables to arguments, and we already handle several-argument functions; and the data representation analysis will make functions with several results, which can probably be handled analogously to several arguments.

How to extend our work to separate/incremental compilation was touched upon in the end of section 7.2.

REFERENCES

- Aiken, Alexander, Manuel Fähndrich & Raph Levien (1995): Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **30**(6)). San Diego, California.
- Andersen, Finn Schiermer (1995): Re: prisen for et miss. E-mail Sep 19 13:00:28 1995.
- Appel, Andrew W. (1992): *Compiling with Continuations*. Cambridge.
- Asprey, Tom, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner & Jeff Yetter (1993): Performance features of the PA7100 microprocessor. *IEEE Micro* **6**, 22–35.
- Bertelsen, Peter & Peter Sestoft (1995): *Experience with the ML Kit and Region Inference*. Incomplete draft 1 of December 13.
- Birkedal, Lars (1994): *The ML Kit Compiler—Working Note*. Unpublished manuscript.
- Birkedal, Lars, Mads Tofte & Magnus Vejlstrup (1996): From region inference to von Neumann machines via region representation inference. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg Beach, Florida. 171–183.
- Birkedal, Lars & Morten Welinder (1993): *Partial Evaluation of Standard ML*. Master's thesis, Department of Computer Science, University of Copenhagen. (= Technical Report 93/22).
- Birkedal, Lars, Nick Rothwell, Mads Tofte & David N. Turner (1993): *The ML Kit Version 1*. Technical Report 93/14. Department of Computer Science, University of Copenhagen.
- Boquist, Urban (1995): Interprocedural register allocation for lazy functional languages. In *Conference Record of FPCA '95: SIGPLAN-SIGARCH-WG 2.8 Conference on Functional Programming Languages and Computer Architecture*. La Jolla, California.
- Briggs, Preston, Keith D. Cooper & Linda Torczon (1994): Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems* **16**(3), 428–455.
- Brooks, Rodney A., Richard P. Gabriel & Guy L. Steele, Jr. (1982): An optimizing compiler for lexically scoped Lisp. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (= *SIGPLAN Notices* **17**(6)). Boston, Massachusetts. 261–275.
- Burger, Robert, Oscar Waddell & R. Kent Dybvig (1995): Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **30**(6)). La Jolla, California. 130–138.
- Callahan, David & Brian Koblenz (1991): Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **26**(6)). 192–203.
- Cardelli, Luca (1984): Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. Austin, Texas. 208–217.

- Chaitin, Gregory J. (1982): Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (= *SIGPLAN Notices* **17**(6)). Boston, Massachusetts. 98–105.
- Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins & Peter W. Markstein (1981): Register allocation via coloring. *Computer Languages* **6**, 47–57.
- Chow, Fred C. (1988): Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **23**(7)). Atlanta, Georgia. 85–94.
- Chow, Fred C. & John L. Hennessy (1990): The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems* **12**(4), 501–536.
- Coutant, Deborah S., Carol L. Hammond & Jon W. Kelley (1986): Compilers for the new generation of Hewlett-Packard computers. *Hewlett-Packard Journal* **37**(1), 4–18.
- Damas, Luis & Robin Milner (1982): Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*. Albuquerque, New Mexico. 207–212.
- Elsman, Martin & Niels Hallenberg (1995): An optimizing backend for the ML Kit using a stack of regions. Student Project. Department of Computer Science, University of Copenhagen.
- Fleming, Philip & John J. Wallace (1986): How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM* **29**(3), 218–221.
- Garey, M. & D. Johnson (1979): *Computers and Intractability – A Guide to the Theory of NP-Completeness*. New York.
- George, Lal & Andrew W. Appel (1995): *Iterated Register Coalescing*. Technical Report CS-TR-498-95. Department of Computer Science, Princeton University.
- Gibbons, Phillip B. and Muchnick, Steven S. (1986): Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*. Palo Alto, California.
- Gupta, Rajiv, Mary Lou Soffa & Denise Ombres (1994): Efficient register allocation via coloring using clique separators. *ACM Transactions on Programming Languages and Systems* **16**(3), 370–386.
- Harper, Robert & Greg Morrisett (1995): Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California. 130–141.
- Henglein, Fritz (1992): *Simple Closure Analysis*. DIKU Semantics Report D-193. Department of Computer Science, University of Copenhagen.
- Henglein, Fritz & Jesper Jørgensen (1994): Formally optimal boxing. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon. 213–226.
- Hennessy, John L. & David A. Patterson (1990): *Computer Architecture: A Quantitative Approach*. San Mateo, California.
- Hewlett-Packard (1991 a): *Assembly Language Reference Manual*, 4th. edn. [Software Version 92453-03A.08.06].

- Hewlett-Packard (1991b): *PA-RISC Procedure Calling Conventions Reference Manual*, 2nd. edn. [HP Part No. 09740-90015].
- Hewlett-Packard (1992): *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 2nd. edn. [HP Part No. 09740-90039].
- Jørgensen, Jesper (1995): *A Calculus for Boxing Analysis of Polymorphically Typed Languages*. Ph.D. thesis, Department of Computer Science, University of Copenhagen. Universitetsparken 1, DK 2100 Copenhagen Ø.
- Kannan, Sampath & Todd Proebsting (1995): Register allocation in structured programs. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. 360–368.
- Kernighan, Brian W. & Dennis M. Ritchie (1988): *The C Programming Language*, 2nd. edn. Englewood Cliffs, New Jersey.
- Kessler, R. R., J. C. Peterson, H. Carr, G. P. Duggan, J. Knell & J. J. Krohnfeldt (1986): EPIC – A retargetable, highly optimizing Lisp compiler. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction (= SIGPLAN Notices 21(7))*. Palo Alto, California. 118–130.
- Koch, Martin & Tommy Højfeldt Olesen (1996): *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.
- Kranz, David, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin & Norman Adams (1986): ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction (= SIGPLAN Notices 21(7))*. Palo Alto, California. 219–233.
- Landin, P. J. (1964): The mechanical evaluation of expressions. *Computer Journal* **6**(4), 308–320.
- Launchbury, John (1993): Lazy imperative programming. In *Proceedings of the ACM Workshop on State in Programming Languages*. Copenhagen.
- Lee, Ruby B. (1989): Precision Architecture. *Computer* **22**(1), 78–91.
- Leroy, Xavier (1992): Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Albuquerque, New Mexico. 177–188.
- Mahon, Michael J., Ruby Bei-Loh Lee, Terrence C. Miller, Jerome C. Huck & William R. Bryg (1986): Hewlett-Packard Precision Architecture: The Processor. *Hewlett-Packard Journal* **37**(8), 4–21.
- Milner, R. (1978): A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17**, 348–375.
- Milner, Robin & Mads Tofte (1991): *Commentary on Standard ML*. Cambridge, Massachusetts.
- Milner, Robin, Mads Tofte & Robert Harper (1990): *The Definition of Standard ML*. Cambridge, Massachusetts.
- Mueller, Frank & David B. Whalley (1992): Avoiding unconditional jumps by code replication. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (= SIGPLAN Notices 27(7))*. San Francisco, California. 322–330.
- Norris, Cindy & Lori L. Pollock (1994): Register allocation over the program dependence graph. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (= SIGPLAN Notices 29(6))*. 266–277.

- Paulson, Lawrence C. (1991): *ML for the Working Programmer*. Cambridge.
- Pettis, Karl W. & William B. Buzbee (1987): Hewlett-Packard Precision Architecture compiler performance. *Hewlett-Packard Journal* **38**(3), 29–35.
- Plasmeijer, Rinus & Marko van Eekelen (1993): *Functional Programming and Parallel Graph Rewriting*. Workingham.
- Reynolds, John C. (1995): Using functor categories to generate intermediate code. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California. 25–36.
- Santhanam, Vatsa & Daryl Odnert (1990): Register allocation across procedure and module boundaries. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **25**(6)). 28–39.
- Sestoft, Peter (1992): *Analysis and Efficient Implementation of Functional Programs*. Ph.D. thesis, Department of Computer Science, University of Copenhagen. (= Technical Report 92/6).
- Sethi, Ravi & J. D. Ullman (1970): The generation of optimal code for arithmetic expressions. *Journal of the ACM* **17**(4), 715–728.
- Shao, Zhong & Andrew W. Appel (1994): Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming* (= *LISP Pointers* **7**(3)). Orlando, Florida. 150–161.
- Shivers, Olin (1988): Control-flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **23**(7)). Atlanta, Georgia. 164–174.
- Steele, Jr., Guy Lewis (1977): *Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto*. Master's thesis, Artificial Intelligence Laboratory, MIT.
- Steenkiste, Peter A. (1991): Advanced register allocation. In Peter Lee (ed.): *Topics in Advanced Language Implementation*. Cambridge, Massachusetts. Chap. 2, 25–45.
- Steenkiste, Peter A. & John L. Hennessy (1989): A simple interprocedural register allocation algorithm and its effectiveness for Lisp. *ACM Transactions on Programming Languages and Systems* **11**(1), 1–32.
- Tarditi, David, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper & Peter Lee (1996): TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. Philadelphia, Pennsylvania. ?–?
- Thorup, Mikkel (1995): *Structured Programs have Small Tree-Width and Good Register Allocation*. Technical Report 95/18. Department of Computer Science, University of Copenhagen.
- Tofte, Mads (1995): *Region-Based Memory Management for the Typed Call-by-Value Lambda Calculus*. Submitted for publication.
- Tofte, Mads & Jean-Pierre Talpin (1993): *A Theory of Stack Allocation in Polymorphically Typed Languages*. Technical Report 93/15. Department of Computer Science, University of Copenhagen.
- Tofte, Mads & Jean-Pierre Talpin (1994): Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon. 188–201.

- Waite, W. M. (1974): Code generation. In F. Bauer & J. Eickel (eds.): *Compiler Construction—An Advanced Course* (= *Lecture Notes in Computer Science* **21**). Berlin. Chap. 3E, 302–332.
- Wall, David W. (1986): Global register allocation at link time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (= *SIGPLAN Notices* **21**(7)). Palo Alto, California. 264–275.
- Wand, Mitchell & Paul Steckler (1994): Selective and lightweight closure conversion. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon. 435–445.
- Welsh, Jim & John Elder (1982): *Introduction to PASCAL*, 2nd. edn. Englewood Cliffs, New Jersey.

C

$c \in C = \dot{C} \cup \overset{\circ}{C}$, constructor, p. 15
 $\dot{c} \in \dot{C}$, unary constructor, p. 15
 $\overset{\circ}{c} \in \overset{\circ}{C}$, nullary constructor, p. 15
 $\mathcal{C} \in \mathcal{C} = (\Phi \times \Pi^* \times \Phi) \cup \{-cc\}$,
 calling convention, p. 158
 $\llbracket c \rrbracket_{C \rightarrow I}$, target representation of c ,
 p. 24
 $\llbracket \cdot \rrbracket_{ca} \in E \rightarrow \hat{E}$, closure analysis,
 p. 113
 cc , get calling convention, p. 159
 $\llbracket \cdot \rrbracket_{cg} \in \hat{E} \rightarrow ?$, build the call
 graph, p. 63
children, find children in a graph,
 p. 64
choose, heuristic for choosing a
 register, p. 87
 $clos \in V$, closure, p. 84
 $coerce_{B \rightarrow \Sigma}$, convert β to σ , p. 155
 $coerce_{\Sigma \rightarrow B}$, convert σ to β , p. 155
 $\llbracket \cdot \rrbracket_{compile} \in E \rightarrow \mathfrak{P}$, pp. 110, 63, 67
 $\llbracket \cdot \rrbracket_{cr} \in \hat{E} \rightarrow \dot{E}$, annotate closure
 representation, p. 119

D

$d \in D$, description of register, p. 137
 $\delta = (\delta^v, \delta^t, \delta^d) \in \Delta$, descriptor,
 p. 137
 $\delta^d \in V \xrightarrow{\perp} D$, register descriptor,
 p. 137
 $\delta^t \in (\Phi \times (W \times P))^*$, stack of
 temporaries, p. 138
 $\delta^v \subseteq V$, values that are loaded,
 p. 103
 $\mathfrak{d} \subseteq \mathfrak{P} \times \mathfrak{P}$, edges in dependency
 graph, p. 189
 $\mathcal{D} \in \Phi \xrightarrow{\perp} \mathfrak{P}$, last instruction to
 define a register, p. 190
 $\llbracket e \rrbracket_{da}$, guess which registers e
 destroys, p. 128
 $\llbracket \lambda \rrbracket_{da-\Lambda}$, guess which registers λ
 destroys, p. 129
decide-cc, pp. 176, 175
decide-rc, pp. 176, 174
 $\llbracket \cdot \rrbracket_{def}$, translate definition of a

value, p. 88

$\llbracket p \rrbracket_{defd}$, registers defined by p ,
 p. 190
dependencies, build dependency
 graph for a basic block, p. 190
destroys, registers destroyed at
 application, p. 128
 Dm , domain of function, p. 13
 $\llbracket \cdot \rrbracket_{donode}$, process a call graph
 node, i.e. a λ , pp. 64, 72, 125
don't = $\lambda\zeta.\zeta$, preserver, p. 138
 $do-scc \in \mathcal{X}^\circ \rightarrow H \rightarrow K \times H$, process
 scc pp. 67, 125

E

$e \in E$, our source language, i.e.,
 region-annotated E , p. 10
 $\hat{e} \in \hat{E}$, lambda-annotated E , p. 62
 $\overset{\circ}{e} \in \overset{\circ}{E}$, sibling-annotated E , p. 117
 $\dot{e} \in \dot{E}$, closure-representation-
 annotated E , p. 119
 $\check{e} \in \check{E}$, several-argumented E , p. 121
 $\overset{\circ}{e} \in \overset{\circ}{E}$, recursiveness-annotated E ,
 p. 69
 $\overset{*}{e} \in \overset{*}{E}$, ω -annotated E , p. 86
 E , language before region analyses,
 p. 10
 $\epsilon \in K$, no-operation instruction,
 p. 20
 $\varepsilon = (\nu, \lambda_{cur.}) \in E$, per-function
 environment, p. 125
 ε^η , η -component of ε , p. 64
 $\eta = (\eta^l, \eta^d) \in H$, inter-procedural
 environment, p. 64
 $\eta^d \in \mathcal{P}(\mathcal{P}\Lambda) \xrightarrow{\perp} \Phi$, environment
 recording registers destroyed by
 scc of λ 's, p. 65
 $\eta^l \in \mathcal{P}(\mathcal{P}\Lambda) \rightarrow \mathcal{L}$, environment
 recording linking convention for
 equivalence class of λ 's, p. 65
 $\mathcal{E} \subseteq \Lambda \times \Lambda$, edges in the call graph,
 p. 63
endregion, p. 21
endregions, pp. 47, 21
entry, p. 174

F

$f \in F$, **letrec**-function name, p. 18
 $\mathbf{f} \in \mathbf{F} = \mathcal{P}F$, sibling name, p. 117
 \mathcal{F} , p. 117
 $\phi \in \Phi$, register, p. 20
 $\phi_{\text{descriptors}}$, p. 23
 ϕ_{dp} , data pointer, p. 164
 ϕ_{free} , p. 23
 $\phi_{\text{letregion}} \in \Phi$, natural destination of a letregion-instruction, p. 141
 ϕ_{raised} , handler argument register, p. 164
 ϕ_{sp} , stack pointer, p. 21
 $\hat{\phi} \in \hat{\Phi}_{\perp} = \Phi \cup \{-\text{register}\}$, register to preferably choose, natural destination register, p. 82
 $\hat{\phi} \subseteq \Phi$, registers that must not be chosen, p. 23
 $\hat{\phi}_{\text{at}}$ } registers
 $\hat{\phi}_{\text{endregion}}$ } destroyed by
 $\hat{\phi}_{\text{endregions}}$ } different
 $\hat{\phi}_{\text{letregion}}$ } heavy-weight
 $\hat{\phi}_{\text{raise}}$ } instructions
 $\phi_{\text{heavy}} = \hat{\phi}_{\text{at}} \cup \hat{\phi}_{\text{letregion}} \cup \hat{\phi}_{\text{endregion}}$, registers destroyed by heavy-weight instructions, p. 204
 $\phi_{\text{rest}} = \Phi \setminus \phi_{\text{heavy}}$, p. 204
 $\hat{\phi} \subseteq \Phi$, registers to avoid choosing, p. 87
 $\check{\phi} \subseteq \Phi$, registers to preferably choose, p. 73
 $\phi.\iota \in \mathbf{K}$, test bit ι of ϕ , p. 20
 $\varphi \subseteq \mathbf{P} \times \mathbf{P}$, region aliasing information, p. 113
find, p. 126
 $\llbracket e \rrbracket_{\text{fv}}$, free variables of e , p. 119

G

$g \in E \cup \mathbf{F}$, p. 159
 $\gamma = (\lambda^{\text{cg}}, \mathcal{E}, \lambda_{\text{main}}) \in ?$, call graph, p. 63
 $\gamma = (\lambda^{\text{os}}, \mathcal{S}, \lambda_{\text{main}}^{\text{c}}) \in \Gamma$, sccs graph, p. 67
goto, p. 161

H

\mathbf{h} , current handler (global variable), p. 48
 \mathbf{H} , inter-procedural environments, p. 64
 $\llbracket \cdot \rrbracket_{\text{has-been-loaded}}$, record in δ that some value is loaded, p. 103
heur, p. 90

I

$i \in I$, source integer, p. 14
 i_{frame} , p. 163
 $\iota \in I$, target integer, label, p. 20
 $\iota_{\mathbf{h}}$, label of \mathbf{h} , p. 164
 $\iota_{\mathbf{n}}$, label of \mathbf{n} , p. 178
 $\mathbf{i} \in I$, p. 184
 $\llbracket i \rrbracket_{I \rightarrow I}$, representation of i , p. 24

K

$\mathcal{K} \in \mathbf{K} = Z \xrightarrow{\perp} I$, closure representation, p. 118
 $\kappa \in \mathbf{K}$, intermediate language, p. 20
 $\acute{\kappa} \in \acute{\mathbf{K}} \subseteq \mathbf{K}$, jump instructions in \mathbf{K} , p. 182
 $\grave{\kappa} \in \grave{\mathbf{K}}$, non-jump instructions, p. 182
 $\hat{\kappa} \in \hat{\mathbf{K}}$, linear code, p. 183
 $\chi \in \mathbf{X}$, condition in \mathbf{K} , p. 20
 $\llbracket \cdot \rrbracket_{\text{kill}}$, give a preserver for a register if the value it contains is loaded, p. 104
kill-arg, pp. 163, 161
kill-tmp, mark end of live range of temporary; the dual of *new-tmp*, p. 136

L

$\lambda \in \Lambda$, function in E , p. 62
 $\lambda_{\text{cur.}} \in \Lambda$, the λ currently being processed, p. 72
 $\lambda_{\text{cur.}}^*$, ω -annotated $\lambda_{\text{cur.}}$, p. 174
 $\lambda_{\text{main}} = \lambda_{\text{y}_{\text{main}}} . e$ at $\mathbf{r}_{\text{main}} \in \Lambda$, whole program; root node in call graph, p. 63
 $\lambda \in \mathbf{\Lambda} = \mathcal{P}\Lambda$, p. 62
 $\lambda^{\text{cg}} \subseteq \Lambda$, nodes in callgraph; the λ 's of λ_{main} , p. 63
 $\lambda^{\equiv} \subseteq \Lambda$, equivalence class of λ 's having the same linking

convention, p. 65
 $\lambda^\circ \subseteq \Lambda$, scc, node in sccs graph,
 p. 67
 $\lambda_{\text{cur}}^\circ$, the scc currently being
 processed by *do-scc*, p. 72
 $\lambda_{\text{main}}^\circ = \{\lambda_{\text{main}}\}$, root node in sccs
 graph, p. 67
 $\lambda^{\text{=s}} \in \mathcal{P}(\mathcal{P}\Lambda)$, set of equivalence
 classes, p. 65
 $\lambda^{\text{os}} \in \mathcal{P}(\mathcal{P}\Lambda)$, set of sccs, p. 67
 $\mathcal{L} \in \mathcal{L} = \mathcal{C} \times \mathcal{R}$, linking
 convention, p. 158
 $\llbracket \lambda \rrbracket_{\Lambda \rightarrow \mathbb{I}}$, unique label for λ , p. 163
letregion, p. 17
letregion, p. 21
 $\text{lin} \in \mathcal{PB} \rightarrow \hat{K}$, linearise K basic
 blocks, p. 184
 $\llbracket \cdot \rrbracket_{\text{load}}$, translate load of value,
 p. 103

M

\mathbf{m} , memory pseudo-register, p. 192
 $\mathbf{m}[\cdot]$, memory access, p. 20
 $\mu = (\delta, \varepsilon) \in \mathbf{M}$, p. 134
 μ^η , η -component of μ , p. 64
move, record in δ that some value is
 copied to another register, p. 139

N

N , approximate number of registers,
 p. 159
 $n_{\text{r.d.}}$, size of region descriptor, p. 140
 $\nu = (\eta, \phi)$, per-scc environment,
 p. 73
 \mathbf{n} , last exception name (global
 variable), p. 51
new-tmp, get register for temporary,
 p. 135

O

$o \in O$, binary operator, p. 14
 $o \in O = S \rightarrow I \rightarrow \mathbb{I}$, p. 145
 $\llbracket \cdot \rrbracket_{\text{oa}}$, ω -analysis, p. 131
 $\llbracket o \rrbracket_{\text{o-prim}}$, code for o , p. 135

P

$p \in P = Z \rightarrow Z$, preserver, p. 138

P , region variables, p. 14
 $\pi \in \Pi = I \cup \Phi$, parameter
 convention, p. 158
 ϖ , p. 115
 $\mathbf{p} \in \mathfrak{P}$, PA-RISC assembler
 language, p. 10
 $\mathbf{p} \subseteq \mathfrak{P}$, nodes in dependency graph,
 p. 189
 $\mathbf{p}_c \in \mathcal{P}\mathfrak{P}$, candidates for scheduling,
 p. 190
 \mathcal{PM} , subsets of \mathcal{M} , p. 13
 $\llbracket \cdot \rrbracket_{\text{pa}} \in \hat{K} \rightarrow \mathfrak{P}$, p. 186
params, parameters of a λ , p. 176
preserve, give a preserver for a
 value, p. 138
preserve-tmp, give a preserver for a
 temporary, p. 138
 $\psi \in \Psi$, region size variable, p. 18
 $\llbracket \cdot \rrbracket_{\text{push-arg}}$, p. 175

R

$r \in R = \{\circ, \emptyset\}$, recursiveness,
 p. 127
 $\rho \in R$, region variable without size
 annotation, p. 18
 $\rho: \psi \in P$, variable-size ρ , p. 18
 $\rho: i \in P$, known-size ρ , p. 18
 $\rho: ? \in P$, unknown-size ρ , p. 18
 $\rho \in P = \acute{P} \cup \grave{P}$, region variable, p. 14
 $\acute{\rho} \in \acute{P}$, *letregion*-bound ρ , p. 18
 $\grave{\rho} \in \grave{P}$, *letrec*-bound ρ , p. 18
 $\vec{\rho} \in \vec{P}$, vector of ρ 's, p. 18
 $\vec{\rho} \in \vec{P}$, vector of ρ 's, i.e., the region
 arguments to a region
 polymorphic function, p. 18
 $\mathbf{r}_{\text{main}} \in P$, dummy region variable of
 λ_{main} , p. 63
 $\mathcal{R} \in \mathcal{R} = \Phi \cup \{-_{\text{rc}}\}$, returning
 convention, p. 158
ra, record that some value is put in
 some register, pp. 88, 139
 $\llbracket \cdot \rrbracket_{\text{ra}}$, pp. 81–82, 87–89, 102–104
 $\llbracket \cdot \rrbracket_{\text{ra-arg}}$, p. 160
 $\llbracket \cdot \rrbracket_{\text{ra-at}}$, translate allocation in
 region, p. 147
 $\llbracket \cdot \rrbracket_{\text{ra-clos}}$, p. 160

$\text{raise} \in \mathbf{K}$, p. 165
 $\text{rdfs} \in \Gamma \rightarrow \mathbf{K}$, reverse depth-first
traversal of the call graph, pp. 63,
65, 67, 124
 $\llbracket \cdot \rrbracket_{\text{relabel}} \in \hat{\mathbf{K}} \rightarrow \hat{\mathbf{K}}$, p. 184
 $\text{ret} \in V$, return label, p. 84

S

$\varsigma = (\varsigma^e, \varsigma^p) \in \mathbf{S}$, stack shape, p. 103
 $\varsigma^e \in V \xrightarrow{\perp} I$, compile-time stack
environment mapping values to
stack positions, p. 103
 $\varsigma^p \in I$, compile-time stack pointer,
p. 103
 $\sigma \in \Sigma = I \times I \rightarrow \mathbf{Z}$, selector, i.e.,
code abstracted over destination
labels, p. 152
 $\mathcal{S} \subseteq \mathcal{P}\Lambda \times \mathcal{P}\Lambda$, edges in the sccs
graph, p. 67
 $\llbracket \cdot \rrbracket_{\text{sa}} \in \dot{E} \rightarrow \dot{E}$, translate to
functions of several arguments,
p. 121
 $\text{sccs} \in ? \rightarrow \Gamma$, p. 124
 $\llbracket \cdot \rrbracket_{\text{sched.}} \in \mathfrak{P} \rightarrow \mathfrak{P}$, instruction
scheduling, p. 189
 set-cc , set calling convention, p. 160
 set-rc , set returning convention,
p. 160
 $\llbracket \cdot \rrbracket_{\text{sib}} \in \hat{E} \rightarrow \hat{E}$, convert f 's to
 F 's, p. 117

T

$t \in T$, source Boolean, p. 16
 $\tau \in \mathbf{T} = \Sigma \cup \mathbf{B}$, a σ or a β , i.e., code
abstracted over a destination,
p. 152
 $\theta \in I \rightarrow \mathbf{Z}$, p. 160
 ϑ , p. 114
 $\mathbf{t} = \phi \mapsto (w, p)$, temporary, p. 138
 $\llbracket t \rrbracket_{T \rightarrow I}$, target representation of t ,
p. 24
 tmp-tmp , get register for temporary
with short live range, p. 139

U

$u \in U$, unary operator, p. 14

$v \in \Upsilon = I \cup \Phi$, p. 20
 $\mathcal{U} \in \Phi \rightarrow \mathcal{P}\mathfrak{P}$, give instructions
that use a certain register, p. 190
 $\llbracket \cdot \rrbracket_{\text{uf}}$, find equivalence classes, $\lambda^{\equiv \text{s}}$,
p. 126

unantagonise , reset hostility
information in ω , p. 133

underway , p. 184

union , p. 126

$\llbracket u \rrbracket_{\text{u-prim}}$, code for u , p. 177

$\llbracket \cdot \rrbracket_{\text{use}}$, translate access to value,
p. 142

$\llbracket p \rrbracket_{\text{used}}$, registers used by p , p. 190

V

$v \in V ::= Z \mid \text{clos} \mid \text{ret}$, values, p. 84

W

$w \in W ::= V \mid -_d$, p. 137

wipe , record that some registers are
destroyed, p. 140

X

$x \in X$, let -bound variable, p. 14

$\xi \in \Xi = I \times I \cup \Phi$, destination
(register or labels), p. 153

Y

$y \in Y$, λ -bound variable, i.e.
argument, p. 14

$y_{\text{main}} \in Y$, dummy argument of
 λ_{main} , p. 63

$v \in \Upsilon = I \cup \Phi$, p. 20

Z

$z \in Z ::= X \mid Y \mid F \mid P \mid A$, variables
(before $\llbracket \cdot \rrbracket_{\text{sib}}$), p. 19

$\mathring{z} \in \mathring{Z} ::= X \mid Y \mid F \mid P \mid A$, variables
(after $\llbracket \cdot \rrbracket_{\text{sib}}$), p. 117

$\zeta \in \mathbf{Z} = \mathbf{S} \rightarrow \mathbf{K}$, code abstracted
over stack shape, p. 104

$\llbracket \cdot \rrbracket_{\text{zap}}$, p. 175

Æ

$\mathfrak{a} \in E \cup P$, p. 163

$\omega \in \Omega$, ω -information, p. 86