Henning Makholm
henning@makholm.net

Region-Based Memory Management in Prolog

MASTER'S THESIS
MARCH 2000
REVISED AUGUST 2000

Technical Report DIKU-TR-00/09
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK–2100 København Ø
DENMARK
CR Subject Classification: D.3.4

Abstract

This thesis investigates whether region-based memory management can successfully be applied to Prolog programs. The answer is affirmative.

It is shown how region-based memory management can be extended to work with backtracking and logical variables. Experiments with a prototype region-based Prolog implementation show that the time efficiency of the extended region-based model compares favorably with garbage collection and is not prohibitively worse than a purely stack-based Prolog implementation.

The thesis also describes a method for translating a subset of Prolog into C programs that use the extended region-based memory manager. The translation involves several typed and untyped intermediate languages with rigorously defined semantics. Informal arguments for the correctness of most of the individual transformation phases are provided; they are meant to be expandable into fully rigorous proofs.

Contents

Preface

This Master's Thesis (Danish: *speciale*) was submitted to the University of Copenhagen on February 1, 2000, as part of the author's work towards the M.Sc. (Danish: *cand.scient.*) degree in Computer Science. The project carries a nominal workload of 13/24 of one year's full-time study. The project advisor was Professor Neil D. Jones.

This report is accompanied by an electronically available prototype implementation which appears in the list of references as [Makholm 2000].

Audience

The intended audience of the report is computer scientists and computer science students with a background in programming languages. I have done my best not to assume any prior knowledge of regions and assume no particular knowledge of Prolog implementation techniques. I do, however, assume that readers know the absolute basics of Prolog programming.

Readers whose programming-languages background have a formal slant will perhaps feel dissatisfied by the lack of rigorous proofs of the soundness of the various transformations I propose. That lack is intentional—I preferred to use the available time on producing a working implementation that could be used in actual experiments rather than on writing down detailed proofs. The reasoning is that a proof is not worth much if the technique it describes turns out to be hopelessly inefficient in practise, whereas an implementation that seems to work fine in practise but has not been proven rigorously correct is at least a proof of concept.

It should also be noted that I have deliberately chosen the of the various semantics and other applicable definitions so that they support the construction of rigorous proofs.

Thanks

I want to thank Neil Jones for proposing the project and for his support, proofreading, critique, and helpful suggestions throughout the work.

Thanks to Leo Schou-Jensen at Thomas Linder Puls at Prolog Development Center for a couple of inspiring discussions in the beginning of the project. It was originally intended that the project would involve a higher amount of cooperation with PDC; that it did not turn out that way is entirely my fault.

Various subscribers to the Usenet newsgroup comp.lang.prolog patiently answered my naïve questions about traditional Prolog implementation techniques.

Henning Niss, Fritz Henglein, and Morten Voetmann Christensen discussed my early design with me and asked many enlightening questions.

Peter Makholm helped with proofreading.

A lot of people, most of whose names I don't even know, created the excellent free software I used for conducting experiments and writing the report, including Moscow ML, the GNU C compiler, GNU Emacs, TEX, LATEX, XY-pic, and dvips.

Language and notation

I refer to myself, from this point on, as "we". It may seem arrogant to use "majestic plural" in this manner, but we feel that using "I" would be more disturbing, drawing undue attention to ourselves by continually forcing the reader to adjust for whether the text she is reading was written by a single person or a collective.

We also assume that any anonymous person mentioned in the text is a "she". The intention was to provide a counterweight to the traditional tendency to use "he" in that position, but some proofreaders thought it was "sexist" to imply that women are always uninteresting and anonymous. Oh, well.

Following standard mathematical notation, we use $\mathbb{Z}$ for the set of integers and $\mathbb{N}$ for the set of natural numbers.

When we use lists as abstract semantical objects, we use ML notation, that is, we write $x :: \vec{x}$ for the "cons" operation that is notated as $[x|\vec{x}]$ in Prolog.

Revised version

In this revised version, published as a DIKU technical report in August 2000, a number of typographical and spelling errors have been corrected. A few changes of notation have also been made, but the text is otherwise unchanged.

It can now be told that the thesis received the maximum grade of the Danish ten-step "13-scale" at the oral defence on March 9th. An article that summarises the most important results of the thesis will appear at the ACM-SIGPLAN–sponsored International Symposium on Memory Management in October 2000.

Chapter 1

Introduction

This thesis is about how to use region-based memory management in implementations of the programming language Prolog.

We trust most computer scientists to have at least a passing knowledge of Prolog. No computer science curriculum would be complete without a basic introduction to logic programming, and Prolog is the unchallenged *lingua franca* of logic programming.

Most people in this report's intended audience (which is computer scientists with a background in programming language definition, processing, or implementation) also have a good mental model of what memory management is about and which services a memory manager provides.

The "region-based" bit, however, may be new to many readers. Therefore we begin this chapter with a quick introduction to what region-based memory management is, compared to other ways of doing memory management.

1.1 A quick tour of memory-management paradigms

The basic problems of memory management are when to deallocate memory[1] and how to efficiently reuse deallocated memory.

1.1.1 Explicit individual deallocation

Explicit individual deallocation is the simplest strategy for deciding when to deallocate memory. Here, the programmer must request deallocation of each individual memory block. Examples are the `malloc/free` discipline of C, or C++'s `new` and `delete` operators.

[1] When to *allocate* memory is less of a problem. In functional programming languages it is common that memory allocation is implicit in the construction of compound values. Where this is not the case, for example in procedural languages with explicit pointers, it is generally considered acceptable that the programmer is supposed to ask for memory explicitly—even though it is known to be error-prone to rely on the programmer for telling when to *deallocate* things.

In theory, explicit individual deallocation gives the programmer excellent chances of trimming the space usage of programs. The problem is that it can be complex to determine when it is safe to deallocate what, even for a programmer who understands the logic of her program. If she specifies too early deallocation for some memory blocks, the resulting misbehaviour is very subtle and hard to trace. It is a common experience that errors of this kind account for a significant fraction of the bugs experienced in software.

Additionally, in many applications it requires significant algorithmic overhead to keep track of convenient deallocation opportunities. The programmer may decide it is not worth the trouble and never deallocate anything at all. When this happens, explicit deallocation becomes a very bad strategy.

1.1.2 Garbage collection

A better choice is **garbage collection**, which is widely used for functional languages and also for some procedural languages (notably Java). Here, the programmer is not involved in deallocation at all. Instead, a run-time component called the garbage collector periodically deallocates memory blocks whose addresses are not known to the running program anymore. This eliminates the danger of programmer error.

The cost, however, is the time overhead that is used at run time keeping the garbage collector informed about which memory blocks the main program has direct pointers to, the so-called **root set**. The active phase of the garbage collector also takes substantial—and unpredictable—time to complete. This can be a problem in real-time applications[2] where unexpected, unpredictable delays are not acceptable.

Worse yet, there is a time/space tradeoff involved: if one has more memory available the garbage collector needs to run less often and hence uses less time. Some garbage collectors prefer to postpone collection until there is no fresh memory to get from the operating system; a tactic which may not be ideal in multiprogrammed environments.

1.1.3 Reference counting

Reference counting is a lightweight garbage collection scheme where memory blocks are deallocated as soon as no more pointers point to them. The main problem with it is that it fails when circular data structures are created. Languages that do not allow circular data structures (such as Prolog with "occurs-check" unification) might use reference counting to get rid of the time/space tradeoff of garbage collection. Still, it takes a lot of time to maintain the reference counters, so the performance of such systems may still be poor.

[2] As well as interactive applications where one wants to guarantee a timely response to the user's actions.

1.1.4 Stack allocation

Finally, in **stack allocation**, the allocation operation pre-schedules the deallocation of a memory block to happen at some specific point in the future execution history of the program. Stack allocation is very efficient regarding running time as well as space usage; in practise it seldom gives rise to problems with too early deallocation even in languages that allow pointers to stack-allocated data. However, stack allocation is not suited for all kinds of data, as it requires the lifetimes (which here means the time from the block is allocated to it is deallocated) of all the stack-allocated memory block to be properly nested.

Stack allocation also performs excellently with respect to the other big memory management problem: how to reuse deallocated memory efficiently. Explicit deallocation, some garbage collection schemes, and reference counting all suffer from potential fragmentation of the heap. With stack allocation, the unused memory is always contiguous and no fragmentation occurs.

The efficiency of stack allocation makes it attractive enough that most procedural programming languages use it in the default case and require the programmer to specify explicitly when she wants to allocate memory whose lifetime does not correspond to the stack model. And many implementations of non-procedural languages use sophisticated analyses to select stack allocation for as much of the program's data as possible.

1.1.5 Stack-of-regions allocation

Now enter the **stack-of-regions** allocation paradigm, proposed by Tofte and Talpin [1993, 1994, 1997] as a generalisation of normal stack allocation. This paradigm relaxes stack allocation's requirement that the lifetimes of values must be properly nested by introducing a mediator concept called a **region**.

Regions can be thought of as abstractions of lifetimes, or as the "handles" of pre-scheduled deallocation points. They are run-time objects that live on a **region stack**; hence, the lifetimes of *regions* must be properly nested. Normal memory blocks can be allocated at any point in time; they are always allocated *in* a particular region and are deallocated at the end of that region's lifetime. Figure 1.1 illustrates this.

Figure 1.1 makes it apparent that the memory blocks themselves cannot live on the region stack, because all of the regions that are alive at a given point in time need to be able to grow. In the ML Kit [Tofte et al. 1997] the memory blocks in a region are allocated from fixed-size **pages**. The pages used for each region are tied together in a linked list; when the region is deallocated the entire page list is appended to a list of free pages, making it available for use in other regions. This scheme eliminates fragmentation of the heap and has low administrative overhead because there is no need to keep track of how the used part of a page is divided into individual memory blocks.

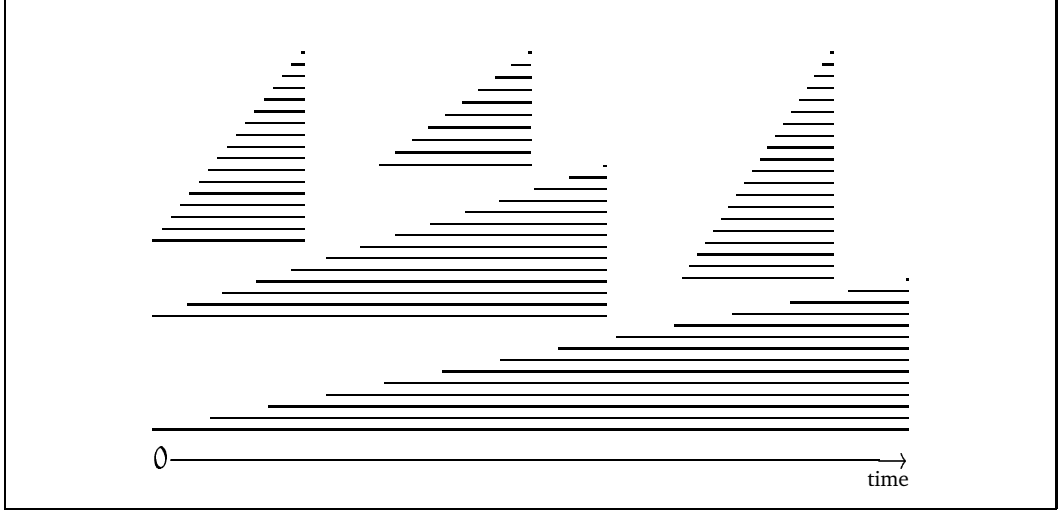


Figure 1.1: Memory-block lifetimes in the stack-of-regions model. Each horizontal line corresponds to the lifetime of one memory block. The triangular collections of blocks are regions.

A compile-time analysis called **region inference** is used to determine when in the program regions are created and destroyed, and which of the existing regions each allocation operation uses. The programmer does not include any explicit any memory-management directives in her code at all; they are provided by the region inference.

A downside of region-based memory management is that the lifetime of a memory block must still be decided when it is allocated. Imagine a program that allocates two blocks of memory and shortly thereafter asks the user which of them to use, after which the unwanted block is never referenced again. With region-based memory management both blocks must be allocated in a long-lived region, so the unwanted one cannot be deallocated early. Nevertheless, the experience with the ML Kit shows that with careful programming such problems can be eliminated or at least kept under control.

Region-based memory management was originally formulated for functional languages, and has been implemented for Standard ML in the ML Kit [Tofte et al. 1997]. Some work (but not nearly as developed as the ML Kit yet) has also been done on adapting region-based memory management to object-oriented languages [Velschow and Christensen 1998].

1.1.6 Non-stacked regions

As observed by Aiken et al. [1995][3], the attractiveness of the region concept is really not closely tied to the principle that the regions form a *stack*. The

[3]The observation is not explicitly present in the article but nevertheless is the fundamental intuitive principle behind Aiken et al.'s work. The "storage mode analysis" [Birkedal et al. 1996] that was part of even Tofte and Talpin's early prototypes can also be seen as building on this principle, but that was apparently not realised at the time.

fact that the lifetimes of regions must be properly nested is more a product of the structure of Tofte and Talpin's formal specification than of the region concept itself.

The important idea is that of a region as a "place" where memory can be allocated, such that all of the memory allocated from a region can later be deallocated in a single operation. The list-of-pages idea from the ML Kit provides an efficient implementation of these primitive operations; it is not an important part of the general idea that the way the ML Kit *uses* these primitives implies a stack discipline for regions.

(The reader may wonder why we introduce regions in this roundabout way, first claiming they need a region stack, then proceeding to say that stack is inessential. The reason is that much of the existing literature about region-based memory management, even after Aiken et al. [1995], has presented "the stack-of-regions model" as an integral whole. We feel that it is worth some energy to point out explicitly that it is not).

1.2 Prolog

We assume you know approximately what Prolog is about, at least well enough to get up to date by the following quick summary. If you are one of the readers we just offended on page 9 by claiming their education was incomplete[4] we suggest you consult a textbook such as Bratko [1990].

We would like to emphasise that the Prolog we talk about is Prolog the programming language—not Prolog the theorem prover nor Prolog the artificial-intelligence engine. We acknowledge that Prolog programs can be used for applications in logic and AI research, but what Prolog programs *do* is really not the perspective that interests us here. So if you expect to see phrases like "SLD resolution" or "knowledge representation" in what follows, then you're probably reading the wrong thesis. Consider yourself warned.

That being said, we also acknowledge that Prolog's origins in theorem proving have led to some concepts being called something different from the names they have in most other. We're not trying to invent a new vocabulary in addition to a new memory-management technique, so we use the normal Prolog words for Prolog's concepts. Most notably, a "predicate" is what other programming languages would call a "procedure", and a "functor" is what other programming languages would call a "(value) constructor". And a "variable" is something completely different from the thing other programming languages call "variables".

We view Prolog as a mostly normal, first-order, typeless programming language with a slightly unusual syntax and two principal features that make it stand out from other programming languages:

[4] Or if you don't have a computer science degree we can offend but would nevertheless like to understand this report.

1. **Backtracking**. Prolog's basic control-flow concept. Backtracking consists of going back to an earlier point (a **choice point**) in the execution history of the program, forgetting everything that happened since the first time we were there, and then continuing to execute some code that was saved as an alternative continuation at that time.

   Prolog does not have any primitive if–then–else construct, but backtracking can be invoked conditionally, which is enough to build a working control structure.

   In addition to being the only control-flow tool, for unfortunate historic reasons backtracking also doubles as Prolog's default reaction to many kinds of run-time errors which in other languages would simply abort execution with an error message. This makes life difficult for Prolog programmers who have a hard time finding out which of a multitude of possible errors made their program backtrack unexpectedly. In this report (which is concerned with implementing Prolog programs, correct or not) the problem is mainly one of terminology, namely that that the command to unconditionally initiate backtracking is spelled "fail" but used when the programmer deliberately wants the program to backtrack. We reluctantly adopt that usage and say "failure" about anything that causes the program to backtrack. When we talk about genuine error conditions that ought not to happen, we say "error" or "wrong".

2. **Logical variables** and **unification**. A logical variable[5] is a "hole" or a "placeholder" which can be inserted in a data structure to signify some data that is not there yet. When a logical variable is duplicated the two copies stay connected in the sense that if one of the copies are **instantiated** to some value (which may also be another variable) the same value automatically appears where the other copy of the variable used to be. In this thesis we try to use the word "variable" exclusively about logical variables, so we use the adjective "logical" only where we fear confusion.

   Unification (which some authors call **matching**) is how variables get instantiated. It is an operation which takes two terms and instantiates the variables it finds in the terms to whatever values are necessary to make the to terms completely identical[6]. If it is not possible to make the terms identical, backtracking is initiated.

The main challenge in adapting region-based memory management to Prolog is to make it work for these two features, because they are the ones that are not present in the languages region-based memory management has previously been used for.

[5] Which has nothing to do with a "boolean variable" in other programming languages.

[6] This description tacitly ignores an above-average number of fine points, but we suppose the readers who would appreciate them at this point already know what they are. The details are all in Section 3.6.

$$
\begin{array}{rcl}
\langle program \rangle & ::= & \langle clause \rangle ... \langle clause \rangle \\
\langle clause \rangle & ::= & \langle atomic \rangle \texttt{ :- } \langle goal \rangle, ..., \langle goal \rangle. \\
\langle goal \rangle & ::= & \langle atomic \rangle \\
& | & \langle term \rangle = \langle term \rangle \\
& | & \texttt{!} \\
& | & \texttt{fail} \\
\langle atomic \rangle & ::= & \mathtt{pr}(\langle term \rangle_1, ..., \langle term \rangle_n) \\
\langle term \rangle & ::= & \mathtt{f}(\langle term \rangle_1, ..., \langle term \rangle_n) \\
& | & ... \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid ... \\
& | & \mathtt{X}
\end{array}
$$

Figure 1.2: The "core" Prolog subset we work with.

Prolog, like many other programming languages, has developed into a variety of dialects—including a committee-created ISO standard—which all offer slightly different features. The differences are of little concern in our context, because we only work with a subset consisting of the "core" features that are common to all implementations.

The Prolog subset we work with are those programs that can be written in the syntax shown in Figure 1.2 and executed without any "built-in" predicates but primitives for rudimentary arithmetic and I/O. In Section 8.1 we list some convenient programming constructs that can be viewed as "syntactic sugar". Section 12.1.3 discusses common Prolog constructs that can *not* be expressed in our subset.

Our Prolog subset is almost what is known as the **pure** fragment of Prolog, except that we support the **cut**, "!", which is used to direct future backtracking operations to ignore certain otherwise valid destinations.[7] Our reason for including cut is twofold. First, very little serious Prolog programming is possible without it. Second, it provides unique implementation problems of its own, so our demonstration that region-based memory management is possible for Prolog would have little weight if we ignored cut.

1.3 Memory management in Prolog

We now turn to describe various ways of doing memory management in Prolog implementations.

[7] It is not easy to define the meaning of cut precisely with a few succinct sentences. If this quick resume does not ring a bell you may either consult a textbook or decide to let the matter rest until Section 2.5 where our formal semantics for GP defines rigorously what we take cut to mean.

1.3.1 The WAM

The classical reference model for Prolog implementations is the Warren Abstract Machine [Aït-Kaci 1991]. It essentially uses two stacks for its memory management. A **local stack** holds temporary results and is used to keep track of the control flow. Deallocation on the local stack happens often. The **heap** or **global stack** shrinks only at backtracking and is used to store terms and variables whose lifetime may be too long for the local stack. There are subtle (but local) rules for choosing which intermediate variables are known to be short-lived enough to be allocated on the local stack.

The problem with the WAM strategy is that the heap shrinks too seldom for some programs. For "functional-style" programs that only backtrack as a way to do case analysis on terms, the heap tends to grow monotonically even though the amount of data actually used remains small throughout the computation. (Cuts are often used to improve the efficiency of such programs but do not solve this problem: a cut does not reclaim any memory in the heap in the WAM).

As a result, Prolog programmers who use an implementation with WAM-like memory management (such as Visual Prolog [Prolog Development Center 2000] or one of its predecessors) need to adopt a certain style of programming in order to avoid space problems. The `repeat-fail` loop is an excellent example. A `repeat-fail` loop causes good memory reuse on the WAM, but it also tends to make the program harder to understand, because imperative tricks are needed to pass information between the iterations.

1.3.2 WAM with a garbage collector

Due to the problems with unlimited growth of the global stack, many otherwise WAM-based Prolog implementation use a garbage collector to compact the stack when it grows too big.

Adding a garbage collector to the WAM is not entirely trivial, (see, for example, [Demoen et al. 1996] which describes the design of one such garbage collector)—especially because one wants to retain the WAM's efficient stack-based deallocation for programs that do backtrack. Nevertheless the idea seems to work reasonably well in practise.

However, "reasonably well" in this context does not necessarily mean much: garbage collection is presently the *only* practical alternative for those Prolog programs that need it.

1.3.3 Region-based memory management for Prolog

The purpose of this thesis is to try to adapt region-based memory management to logic programming languages, in particular Prolog, and to assess the utility of the paradigm in this context.

The latter part of this goal is not trivial: what is known about the performance of regions for ML is not directly transferable to Prolog. There are two main reasons for this.

First, for ML, the main "competition" is garbage collection, whose main perceived problem is execution speed. Region-based memory management could be declared a success for ML because it was faster than garbage collection and did not use unacceptably more memory. On the other hand the the WAM amounts for much of the "competition" on the Prolog scene: garbage collection is not always as big a burden on a Prolog program as it would be on an equivalent ML program. That may shift the performance balance in favour of existing techniques.

Second, the original region model cannot be used unchanged in Prolog, because it does not support backtracking. It is not known *a priori* how to adapt regions to support backtracking or how much that adaption will cost in terms of running time.

1.4 Our approach

A common problem with "apply technique X to programming language Y" projects is that the programming language is too rich and complex for an initial study of the technique's feasibility. The canonical way of solving it is to restrict one's attention to a subset of the programming language which is just big enough to exhibit its most prominent characteristics and allow nontrivial example programs to be written. In our case we did that with the definition of the "core Prolog" subset in Figure 1.2.

Having done that, we then find ourselves in the less common situation that the programming language is too tight, too orthogonal, and too "elegant" (in a minimalistic sense) to allow the theory to be applied to it. In particular, the unification operation is so versatile and powerful that Prolog programs use it for a multitude of purposes, some, but not all, of which require memory allocations. If we are to reason efficiently about the memory-use properties of Prolog programs, we first need to convert the program into a form where it is more explicit what happens at each point in the program. That is, the region inference must work at the level of intermediate code rather than source code.

This means that we must decide on a form of intermediate code to use. The only widely-known "intermediate language" for Prolog is WAM code [Aït-Kaci 1991], which does not distinguish precisely enough between allocations and non-allocations for our purposes. We have also looked as Visual Prolog [Prolog Development Center 2000] whose execution model has most of the properties we want, but it turns out that Visual Prolog does not use intermediate code at all; it generates machine code directly from Prolog.

This left us with defining our own intermediate code. The result is the little language P which we define in Chapters 2–3. ("P" stands for "Very Reduced Prolog"). We then imagine this architecture for a region-based prolog implementation:

- The Prolog source code is parsed and translated to P. A translator that does this is developed in Chapter 8.

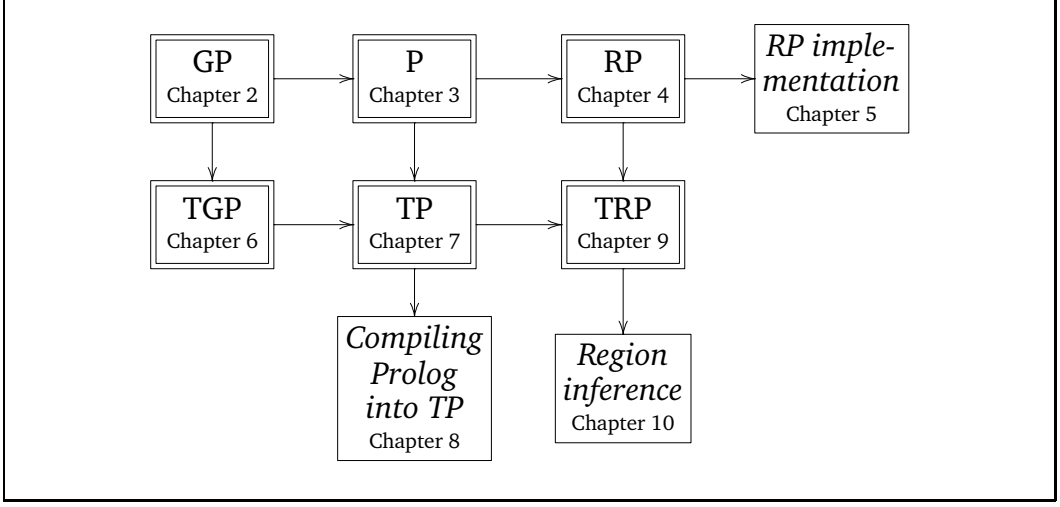


Figure 1.3: An overview of the central chapters of the thesis.

- A region inference algorithm adds explicit region annotations to the P program. The result is a program in a P-like language with explicit region-based memory management which we call RP. We describe RP in Chapter 4 and develop the region inference algorithm in Chapter 10.
- The RP program is translated to machine code which links to a run-time memory-management module which we develop in Chapter 5. In our prototype implementation we use C [Kernighan and Ritchie 1988] as an intermediate step, such that the RP program is first translated to C and then compiled to machine language with a standard compiler.

We have developed a prototype implementation of this architecture, available as [Makholm 2000].

1.4.1 Overview of the thesis

Figure 1.3 depicts the relations between the central chapters of the thesis, with the arrows corresponding roughly to reading-order constraints.

The chapters that are shown with with double frames present languages or type systems. All of these have names ending in "P"; the prefix letters from the following list indicate other features:

G – programs in the language can only manipulate ground terms (*i.e.*, terms with no uninstantiated variables in them).

R – the language has explicit region-based memory management.

T – the language has a type system.

The P and RP languages are directly used in our region-based Prolog implementation. The GP language is used as a stepping stone in the introduction of P.

The translation from Prolog to P uses the type system TP to help with reasoning about the program. TGP is used as a stepping stone in the introduction of TP.

Our region inference algorithm uses the region-based type system TRP to help guarantee that the resulting RP program is **region safe**, that is, it never tries to reference memory that has been deallocated.

Not shown on Figure 1.3 are Chapter 11, which reports on experimental results with our prototype RP implementation and compares its performance to other memory-management strategies, and Chapter 12, which concludes and suggests directions for further work, including improvements to the region model and the region inference that we did not have time to implement for this thesis.

Chapter 2

GP: a ground Prolog subset

The present chapter presents notation that we use in the following chapters. We define the language GP, whose entire purpose is later to be extended to form P (in Chapter 3). Purposefully, next to nothing is said about memory management here.

GP can be viewed as a first-order functional language with backtracking and cuts. It differs from P in that only ground terms are possible. This has the effect that we can define the language without needing to talk about storage and sharing. Thus, in the next chapter when we introduce the full P language, we will already be familiar with the mechanics of the language and our style of operational semantics. Then we can pay full attention to the important points about storage.

2.1 GP as a Prolog subset

GP arises as a syntactical subset of "core" Prolog by requiring that each goal (or predicate head) have one of the forms

$$\begin{array}{l} pr(X_1, ..., X_n) \\ X = f(X_1, ..., X_n) \\ X = X' \\ ! \\ \mathsf{fail} \end{array}$$

Furthermore,

- The parameters to each predicate are classified as either *input* or *output* parameters. When calling a predicate all of the input parameters must be instantiated to ground terms[1], and the output parameters must be fresh distinct variables. If the predicate succeeds it instantiates the output parameters to ground terms. We refer to this division as **parameter moding**.

[1] A **ground term** is a term that contains no variables, or at least no uninstantiated ones. In our storage model, a variable continues to matter after it has been instantiated; to prevent confusion we shall only use the word "ground" in informal contexts where it is clear that instantiated variables are not an issue.

- In a goal of the form $X = f(X_1, ..., X_n)$ either X or all of the X_is must be instantiated to ground terms in advance; if X is instantiated, no X_i may also be.
- In a goal of the form $X = X'$ at least one of X and X' must be instantiated in advance.
- fail only occurs as the last goal of a clause.

If the parameter modes for each predicate are given, it can be checked statically that these rules are obeyed. One simply scans through each clause from the beginning, keeping track of which variables have yet been instantiated.

The two upper sections of Figure 2.1 shows a naïve list reversal function written in Prolog and the GP subset. Imagine the cuts in the GP version to have been inserted by a Prolog-to-GP translator noting that no ground terms match both clauses for the predicate.

2.2 A better syntax for GP

It is not technically convenient to represent GP programs in traditional Prolog syntax. The $X = f(X_1, ..., X_n)$ goal form has two different roles: it serves to construct new terms as well as to inspect old ones. In connection with memory management this difference is quite important, because construction involves a memory allocation and inspection does not. Therefore, we want to be able to discriminate between these two roles syntactically.

Similarly, $X = Y$ can be used for simple assignment or for checking that two terms are identical.

The Prolog-like syntax also does not make the parameter modes explicit.

Therefore we notate GP programs in the somewhat less Prolog-like syntax shown in the lower pane of Figure 2.1. We now describe this syntax formally.

2.2.1 Lexical elements

GP programs consist of the following lexical elements:

punctuation – such as `:-` and $\rightarrow$.

keywords – enter, fail, exit, call, builtin, construct, destruct, alias, and unify.

predicate names pr – for example `append` and `rev`. Predicates roughly corresponds to what other languages call "procedures" or "functions".

We assume that each predicate name pr has associated a nonnegative **input arity** $\lceil pr \rceil$ and a nonnegative **output arity** $\lfloor pr \rfloor$.

functor names f – for example `cons` and `nil`. In classical Prolog the canonical spelling of these two are "`.`" and "`[]`", but we chose to use a more colloquial alphanumeric notation here. Functors correspond to value constructors in a typical functional language.

```
append([],B,B).
append([X|A],B,[X|C]) :- append(A,B,C).
rev([],[]).
rev([X|A],C) :- rev(A,Ar), append(Ar,[X],C)
```

```
% parameter modes: append(in,in,out)
append(P1,B,R3) :- P1=[], !, B=R3.
append(P1,B,R3) :- P1=[X|A], append(A,B,C), R3=[X|C].
% parameter modes: rev(in,out)
rev(P1,R2) :- P1=[], !, R2=[].
rev(P1,C) :- P1=[X|A], rev(A,Ar),
             T1=[], T2=[X|T1], append(Ar,T2,C).
```

```
append :- enter → (p1, b)
            & destruct p1 as nil → ( )
            & cut
            & alias b → r3
            & exit (r3);
          enter → (p1, b)
            & destruct p1 as cons → (x, a)
            & call append (a, b) → (c)
            & construct cons(x, c) → r3
            & exit (r3).
rev :- enter → (p1)
         & destruct p1 as nil → ( )
         & cut
         & construct nil( ) → r2
         & exit (r2);
       enter → (p1)
         & destruct p1 as cons → (x, a)
         & call rev (a) → (ar)
         & construct nil( ) → t1
         & construct cons(x, t1) → t2
         & call append (ar, t2) → (c)
         & exit (c).
```

Figure 2.1: The naïve reverse function in Prolog, in the Prolog-like syntax for GP (note that the parameter modes are not explicit in that syntax), and in our preferred GP syntax.

We assume that each functor name `f` has associated a nonnegative **arity** $|f|$.

We handle each integer as a special functor with arity `0`.

Note that functor and predicate names are strictly separate concepts in P.

register names `x` – for example, `a`, `t2` or `r1` in the example program. Registers are used to refer to intermediate values during the evaluation of a predicate. The registers correspond to the variables in the Prolog-like syntax, but when we add real logical variables in Chapter 3 we will keep that concept strictly separate from the local intermediate values. Thus henceforth the latter will be called "registers" and never "variables".

The precise lexical definition of predicate, functor, and register names is not important here; our GP syntax makes it explicit how each identifier is to be interpreted.

2.2.2 Grammar

GP programs are defined by this pseudo-grammar[2]:

$$
\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \langle \textit{predicate} \rangle_1 \ldots \langle \textit{predicate} \rangle_n \qquad (n \geq 1) \\
\langle \textit{predicate} \rangle & ::= & \mathrm{pr} \ \texttt{:-} \ \langle \textit{clauses} \rangle \\
\langle \textit{clauses} \rangle & ::= & \langle \textit{clause} \rangle \ . \\
 & | & \langle \textit{clause} \rangle \ ; \ \langle \textit{clauses} \rangle \\
\langle \textit{clause} \rangle & ::= & \mathsf{enter} \rightarrow (x_1,\ldots,x_{\lceil \mathrm{pr} \rceil}) \ \& \ \langle \textit{body} \rangle \\
\langle \textit{body} \rangle & ::= & \langle \textit{instruction} \rangle \ \& \ \langle \textit{body} \rangle \\
 & | & \mathsf{fail} \\
 & | & \mathsf{exit} \ (x_1,\ldots,x_{\lfloor \mathrm{pr} \rfloor}) \\
\langle \textit{instruction} \rangle & ::= & \mathsf{call} \ \mathrm{pr}'(x_1,\ldots,x_{\lceil \mathrm{pr}' \rceil}) \rightarrow (x_1,\ldots,x_{\lfloor \mathrm{pr}' \rfloor}) \\
 & | & \mathsf{builtin} \ \mathrm{pr}'(x_1,\ldots,x_{\lceil \mathrm{pr}' \rceil}) \rightarrow (x_1,\ldots,x_{\lfloor \mathrm{pr}' \rfloor}) \\
 & | & \mathsf{construct} \ f(x_1,\ldots,x_{|f|}) \rightarrow x \\
 & | & \mathsf{destruct} \ x \ \mathsf{as} \ f \rightarrow (x_1,\ldots,x_{|f|}) \\
 & | & \mathsf{alias} \ x \rightarrow x \\
 & | & \mathsf{unify} \ x{=}x \\
 & | & \mathsf{cut}
\end{array}
$$

Note that the length of register tuples depend on the arities of other names; the pr whose arities appear in the $\langle \textit{clause} \rangle$ and $\langle \textit{body} \rangle$ productions is of course the name of the predicate for which the $\langle \textit{clause} \rangle$ applies.

[2]The "pseudo" is because of the informal way to specify repetition. Of course the grammar could be replaced with a standard context-free grammar and the list-length constraints construed separate sanity checks, but we believe this exposition is clearer.

In this grammar, "goals" have been renamed to "instructions" and always start with a keyword that identifies a particular kind of instruction. Both of these changes emphasise a conception of GP as an *abstract machine* rather than a programming language[3]. This perspective will become unavoidable once we introduce memory management primitives, so we might as well use notation and terminology that supports it mentally.

We use a "&" sign rather than a comma to separate instructions. This makes symbolic manipulation of program fragments easier. If commas without enclosing parentheses were to appear in a ⟨*body*⟩, it would be awkward to write down, say, elements of ⟨*body*⟩ × ⟨*body*⟩ with standard pair notation.

2.2.3 Informal semantics

The instruction

$$\mathsf{call}\ pr(x_1,\ldots,x_{\lceil pr\rceil}) \rightarrow (x_1,\ldots,x_{\lfloor pr\rfloor})$$

corresponds to the goal $pr(X_1,\ldots,X_{\lceil pr\rceil},X'_1,\ldots,X'_{\lfloor pr\rfloor})$ in the Prolog-like syntax of Section 2.1.

The builtin instruction is similar to the call instruction but is used when the called predicate is a built-in that the GP implementation provides for doing, for example arithmetic or I/O. We discuss it further in Section 2.3

The instructions

$$\mathsf{construct}\ f(x'_1,\ldots,x'_{|f|}) \rightarrow x$$
$$\mathsf{destruct}\ x\ \mathsf{as}\ f \rightarrow (x'_1,\ldots,x'_{|f|})$$

are versions of the goal $X = f(X'_1,\ldots,X'_{|f|})$ in Section 2.1's notation. construct handles the case where X is unbound before the goal and the X'_is all are bound. It binds x and never fails. destruct handles the case where X but no X_i are bound in advance. It either fails or binds all of the x'_is.

The instructions

$$\mathsf{alias}\ x \rightarrow x'$$
$$\mathsf{unify}\ x_1 = x_2$$

are versions of the goal $X = X'$. alias handles the case where X but not X' is already bound. unify handles the case where X and X' are both bound in advance. In GP, unify simply checks that the terms bound to x_1 and x_2 are identical, and fails if they are not.

The cut instruction is the cut goal "!".

2.2.4 Scope rules for GP

We expect, without further comments, every GP program to adhere to the following scope conventions. They also apply to the rest of the xP languages.

[3]As we use the words here, the difference between a programming language and an abstract machine is mainly one of perspective. A programming language is one in which people write programs. The emphasis is on the human programmer who constructs code to solve a specific problem. On the other hand, an abstract machine is viewed as a model of a computation *process*. The program is given, and we want to study how the machine executes it.

Box 2.1—WHY IS THERE AN alias INSTRUCTION?

The reader may wonder why we chose to include the alias instruction in GP. After all, alias $x \to x'$ & b is operationally equivalent to $[x'/x]b$ (where $[/]$ is the obvious substitution operator).

The reason is that alias will turn out to be of technical importance in the typed TP language which employs a form of subtyping to allow a value of some type to be used as if it had a more general type (*i.e.*, a type that describes a larger set of values). We then use the alias command to allow the same value to be called different names when viewed as having different types. This allows us to isolate the technicalities of subtyping to a single construct and to attach type information to names rather than to appearances of names.

Thus we need to introduce the alias instruction sooner or later. As it is in fact possible to do so in the elementary setting of GP we might as well do it now. There's plenty of complications ahead anyway.

- When a register name appears to the right of the " $\to$ " symbol it is a *binding instance* of that register name. The scope of the binding is the ⟨*body*⟩ that follows the "&" sign immediately after the binding instance.

  Except for binding instances, register names may only appear inside the scope of a binding instance for the particular name.

  We generally assume that each register name in the program has exactly one binding instance in the entire program. We do not adhere strictly to this convention in example code; the reader is supposed to suitably alpha-rename the examples before applying theory to them.

- The scope of a predicate name pr is the entire program.

  Any predicate name that appears in a call instruction must have exactly one definition in the program.

  The name space for builtin instructions is separate from the name space for call instructions.

- Functor names f are not declared. Every mention of each functor name specifies the same functor.

2.3 Built-in predicates

The builtin instruction allow the addition of extra primitive operations to GP. We use it in our prototype implementation (and in examples) for features that are not essential to memory management.

The idea is that we can leave the semantics of these built-ins more weakly defined than the rest of the primitives and avoid wasting space on an absolutely rigorous treatment of them.

We now describe the built-ins supported by our prototype implementation. They all expect integers as input and produce integers as output; this simplifies our semantic treatment of builtin. It would not be conceptu-

ally difficult to allow built-ins to have as complex interfaces as user-written predicates, but it would need a more verbose semantics.

The first group of built-ins support integer arithmetic: `plus`, `minus`, `times`, `divide` and `modulo`. Each of these have input arity 2 and output arity 1. If the input parameters are not numbers the result is undefined. (The reason for this is that our prototype works that way. It would make perfect sense to have built-in predicates that tested their input for numberness and backtracked if the test failed; but the question is not important as our type system GP checks statically that the situation does not arise in practise).

We also suppose that there are two imperative built-ins for communicating with the outside world through a single input-output channel. We do not invent a special kind of functors for representing characters but represent characters by numbers as in Edinburgh Prolog (though our names for the I/O primitives follow the ISO standard).

`get_code` reads a character from a standard input channel. It has input arity 0 and output arity 1.

`put_code` outputs a character to a standard output channel. It has input arity 1 and output arity 0.

An important point about these imperative operations is that their effect on the outside world persist even if the execution path that called them eventually fails and backtracks past the operations.

This may not be in strict accordance with the declarative logical intuition behind the backtracking mechanism. However, practical implementations of Prolog all have these constructs. When Prolog is viewed as a programming language rather than as a theorem prover, this is no problem. Backtracking is not a time machine, it is simply a control-flow device.

2.4 The main goal

There is no explicit main goal in a GP program. Instead there is a designated predicate pr_0 with zero input and output arities, and an implicit main goal which would read

$$\texttt{?-}\ pr_0, \mathsf{fail}.$$

in Prolog notation.

This calls pr_0 and keeps backtracking into it every time it succeeds. If pr_0 eventually fails the program terminates. A GP program thus does not directly return any useful information other than the fact that it did not get caught in an infinite loop. (The primary rationale for this decision is that is relieves the operational semantics of having to define how to extract a structured result from the final state of the computation. The semantics of built-in predicates can be left comparatively vaguely defined).

However, during the computation the program may use the imperative I/O built-ins `get_code` and `put_code` to communicate with the outside world. For example, Figure 2.2 shows how a conventional Prolog-like answer-sequence dialogue could be programmed in GP using these features.

```
pr0 :- enter → ( )
         &  call real-goal ( ) → (answer)
         &  call writestring (answer) → ( )
            %  Ask whether the user wants more answers:
         &  builtin get_code( ) → (command)
            %  Backtrack into real-goal unless the user said ";"
            %  (whose character code is 59)
         &  destruct command as 59 → ( )
         &  cut
         &  exit ( );
       enter → ( )
            %  real-goal failed; answer "no.". First the n...
         &  construct 110( ) → n
         &  call write (n) → ( )
            %  ...then the o...
         &  construct 111( ) → o
         &  call write (o) → ( )
            %  ...then the full stop
         &  construct 64( ) → dot
         &  call write (dot) → ( )
         &  exit ( ).
writestring :- enter → (s)
                  &  destruct s as nil → ( )
                  &  cut
                  &  exit ( );
                enter → (s)
                  &  destruct s as cons → (char, rest)
                  &  builtin put_code(char) → ( )
                  &  call writestring (rest) → ( )
                  &  exit ( ).
```

Figure 2.2: An example that shows how a normal Prolog-like answer-sequence dialogue could be programmed in GP. `real-goal` *is supposed to compute some number (≥ 0) of answers in the form of strings (i.e., lists of character codes).*

2.5 Operational semantics for GP

We give a small-step operational semantics for GP.

The primary role of this semantics is to introduce our ways of handling control-flow before we need to include store manipulations in the semantics. The portions of this semantics that deal with predicate calls and backtracking (in ways that are inspired by Jones and Mycroft [1984]; Debray and Mishra [1988]) will recur without substantial changes in the semantics for P and RP.

Though this semantics is formally the ultimate definition of what GP programs mean, we need to address the question of how it relates to our original definition of GP as a subset of Prolog. The answer is that a GP program that does not "go wrong" in one of various ways discussed below behaves the same as the Prolog program it would be in the Prolog-like syntax for GP we introduced in Section 2.1.

It would be straight-forward, though tedious, to substantiate this claim by proving this equivalence when the Prolog program is to be interpreted according to interpreter V in Jones and Mycroft [1984] or the operational semantics with cuts in Debray and Mishra [1988]. The structures of the semantics are related enough that a correspondence relation between the states of either could be formulated. One could then show that each step of the present semantics is simulated by one or more steps of the Prolog semantics[4].

2.5.1 Objects used in the semantics

$$
\begin{array}{lcll}
\mathfrak{t} & \in & \langle \mathit{Term} \rangle & = \mathfrak{f}\langle \mathit{Term} \rangle^{|\mathfrak{f}|} \\
\mathfrak{x} & \in & \langle \mathit{Register} \rangle & \\
\mathcal{E} & \in & \langle \mathit{Env} \rangle & = \langle \mathit{Register} \rangle \xrightarrow{\text{fin}} \langle \mathit{Term} \rangle \\
\mathfrak{b} & \in & \langle \mathit{body} \rangle & \\
\mathfrak{c} & \in & \langle \mathit{clause} \rangle & \\
\mathfrak{C} & \in & \langle \mathit{clauses} \rangle & \\
\kappa & \in & \langle \mathit{Cont} \rangle & = (\langle \mathit{ValEnv} \rangle \times \langle \mathit{Register} \rangle^* \times \langle \mathit{body} \rangle \times \langle \mathit{Dump} \rangle)^* \\
\mathfrak{fr} & \in & \langle \mathit{Frame} \rangle & = \langle \mathit{ValEnv} \rangle \times \langle \mathit{body} \rangle \times \langle \mathit{Dump} \rangle \times \langle \mathit{Cont} \rangle \\
\mathfrak{fs} & \in & \langle \mathit{Frames} \rangle & = \langle \mathit{Frame} \rangle^* \\
\delta & \in & \langle \mathit{Dump} \rangle & = \langle \mathit{Frames} \rangle \\
\mathfrak{s} & \in & \langle \mathit{State} \rangle & = \langle \mathit{Frames} \rangle \uplus \{\mathsf{ok}, \mathsf{wrong}_1, \mathsf{wrong}_2\}
\end{array}
$$

$\langle \mathit{Frame} \rangle$ is a "nondeterministic state" which contains all the state information that does not concern backtracking. Its main components are $\mathcal{E}$ with bindings of visible variables and a program point $\mathfrak{b}$.

$\langle \mathit{Cont} \rangle$ is a stack of frames that have been suspended by call instructions. The $\langle \mathit{Register} \rangle$ list is the variables the should be bound to the called predicate's return values.

[4] That is, as long as the GP program did not use built-ins, as none of the cited Prolog semantics consider those at all.

A ⟨*Frame*⟩ together with its enclosed ⟨*Cont*⟩ corresponds to the ⟨*Stack*⟩ domain of Jones and Mycroft [1984] or Debray and Mishra [1988]. We need a separate representation of the front frame because of the output parameters which means that the call instruction is not quite finished before after the predicate has returned. In contrast, a Prolog semantics can safely forget about a goal once the named predicate has been started.

A ⟨*Frames*⟩ is a stack of alternative ⟨*Frame*⟩s the first of which is currently being executed. If it fails it gets popped off the stack, whereby execution backtracks to the next ⟨*Frame*⟩ on the stack. A ⟨*Dump*⟩ is the set of alternative states that are still relevant after a cut (in addition to the frame that executes the cut).

A ⟨*State*⟩ either is a ⟨*Frames*⟩, or one of the final states ok, wrong_1, and wrong_2.

Generally wrong_i is the semantic equivalent of a "program crash": error conditions that should be avoided by static checks because it is expensive or impossible to check for them at run-time.

wrong_1 is the kind of error that occurs when an undefined name is mentioned or predicate arities do not match. If the scope rules of Section 2.2.4 had been stated in a more formal way one would have been able to prove that "well-formed GP programs do not go wrong_1".

wrong_2 is caused by using a built-in predicate with input arguments that are not numbers. In Chapter 6 we present a type system TGP that can check that "well-typed TGP programs do not go wrong_2".

ok means that the program has terminated without going wrong_i. According to the decisions in Section 2.4 this happens when the program eventually fails.

2.5.2 The initial state

Following the decisions in Section 2.4, the initial ⟨*State*⟩ consists of a single frame that calls $\mathfrak{pr}_0$ and fails each time $\mathfrak{pr}_0$ succeeds:

$$s_0 = [(\varnothing, \mathsf{call}\ \mathfrak{pr}_0() \to () \ \&\ \mathsf{fail}, [])]$$

2.5.3 The transition relation

A ⟨*State*⟩ that consists of ⟨*Frames*⟩ is a non-final state. We now define a successor state for each non-final state:

The state [] (*i.e.*, the empty list of ⟨*Frame*⟩s) means that the main goal has eventually failed. This is the normal way of termination for GP programs, so the successor of [] is ok.

Any other non-final state has the shape

$$(\mathcal{E}, \mathfrak{b}, \delta, \kappa) :: \mathfrak{fs}$$

and we now proceed by case analysis on $\mathfrak{b}$.

$b =$ call $pr(x_1,\ldots,x_{\lceil pr\rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr\rfloor})$ & b'

- There must be a unique C such that pr :- C is in the program. It must have the form,

$$C = \mathsf{enter} \rightarrow (x_{11},\ldots,x_{1\lceil pr\rceil})\ \&\ b_1;\ \ldots\ ;\ \mathsf{enter} \rightarrow (x_{k1},\ldots,x_{k\lceil pr\rceil})\ \&\ b_k.$$

  Otherwise wrong_1
- If any $x_j \notin \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- $\mathcal{E}_i \leftarrow \{x_{ij} \mapsto \mathcal{E}(x_j) \mid 1 \le j \le \lceil pr\rceil\}$ for $1 \le i \le k$
- $\delta' \leftarrow fs$
- $\kappa' \leftarrow (\mathcal{E}, (x'_1, \ldots, x'_{\lfloor pr\rfloor}), b', \delta) :: \kappa$
- $fr_i \leftarrow (\mathcal{E}_i, b_i, \delta', \kappa')$ for $1 \le i \le k$
- The next state is $fr_1 :: fr_2 :: \ldots :: fr_k :: fs$

$b =$ builtin $pr(x_1,\ldots,x_{\lceil pr\rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr\rfloor})$ & b'

The implementation is supposed to provide a function

$$\phi_{pr} : \mathbb{Z}^{\lceil pr\rceil} \rightarrow \left(\mathbb{Z}^{\lfloor pr\rfloor} \uplus \{\mathsf{fail}, \bot\}\right)$$

- If any $x_i \notin \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- If any $\mathcal{E}(x_i) \notin \mathbb{Z}$ then wrong_2; else
- Compute $\phi_{pr}(\mathcal{E}(x_1), \ldots, \mathcal{E}(x_{\lceil pr\rceil}))$.
  If the result is fail, then the next state is fs.
  If the result is $\bot$, then the current state is its own successor (modelling nontermination when the builtin predicate does not return or when trying to divide by zero).
  Otherwise, let the result be $(n_1, \ldots, n_{\lfloor pr\rfloor})$.
- If any $x'_i \in \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- $\mathcal{E}' \leftarrow \mathcal{E}\{x'_i \mapsto n_i \mid 1 \le i \le \lfloor pr\rfloor\}$
- The next state is $(\mathcal{E}', b', \delta, \kappa) :: fs$

$b =$ construct $f(x_1,\ldots,x_{|f|}) \rightarrow x'$ & b'

- If any $x_i \notin \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- If $x' \in \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto f(\mathcal{E}(x_1), \ldots, \mathcal{E}(x_{|f|}))\}$
- The next state is $(\mathcal{E}', b', \delta, \kappa) :: fs$

$b =$ destruct x as $f \rightarrow (x'_1,\ldots,x'_{|f|})$ & b'

- If $x \notin \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- If not $\mathcal{E}(x) = f(t_1, \ldots, t_{|f|})$ for some t_is, then the next state is fs; else
- If any $x'_i \in \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- $\mathcal{E}' \leftarrow \mathcal{E}\{x'_i \mapsto t_i \mid 1 \le i \le |f|\}$
- The next state is $(\mathcal{E}', b', \delta, \kappa) :: fs$

$b =$ alias $x \rightarrow x'$ & b'

- If $x \notin \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- If $x' \in \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \mathcal{E}(x)\}$
- The next state is $(\mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ unify x_1=x_2 & b'

- If any $x_i \notin \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- If $\mathcal{E}(x_1) \neq \mathcal{E}(x_2)$ then the next state is $\mathfrak{fs}$; else
- The next state is $(\mathcal{E}, b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ cut & b'

- The next state is $(\mathcal{E}, b', \delta, \kappa) :: \delta$

$b =$ fail

- The next state is $\mathfrak{fs}$

$b =$ exit $(x_1,\ldots,x_n)$

- If any $x_i \notin \mathrm{Dom}\,\mathcal{E}$ then wrong_1; else
- If κ is an empty list then wrong_1; else
- $(\mathcal{E}', (x'_1, \ldots, x'_m), b', \delta') :: \kappa' \leftarrow \kappa$
- If $n \neq m$ then wrong_1; else
- If any $x'_i \in \mathrm{Dom}\,\mathcal{E}'$ then wrong_1; else
- $\mathcal{E}'' \leftarrow \mathcal{E}'\{x'_i \mapsto \mathcal{E}(x_i) \mid 1 \leq i \leq n\}$
- The next state is $(\mathcal{E}'', b', \delta', \kappa') :: \mathfrak{fs}$

Chapter 3

P: A Prolog-like intermediate language

In this chapter we extend GP with two new instructions, makevar and deref, that allow uninstantiated variables inside terms; we also describe the interaction between variables and the GP instructions. The result is the language P.

The intention is that P should be a sufficiently strong language that every Prolog program has a natural translation into P. Here, by a "natural" translation we mean that the terms manipulated by the P version should be the same terms as the original Prolog program manipulates; and that there is a P predicate that corresponds to each Prolog predicate. (The point of these conditions is that we do not consider a Prolog interpreter written in GP that interprets a ground encoding of the Prolog program to be a natural translation).

We describe how such a translation can be done in Chapter 8.

At the same time, the instructions of P should be orthogonal and primitive enough to facilitate adding explicit memory-management annotations to it (which we do in Chapter 4). The design of GP has already been heavily influenced by this; the influence will continue as we define P.

Furthermore, P should allow memory-efficient translation of those parts of a Prolog program that are written in a functional style. The reason for this requirement is that region-based memory management (in particular region inference) proves to work much better when there are no variables around. It is common that large parts of actual Prolog programs do not use (logical) variables for anything else than returning ground terms from predicates. These parts can be translated into the GP subset of P, which is why that subset exists at all. However, it is also common that certain parts of program do use variables in a non-trivial way; so P should allow GP-like code to interact with (and blend into) code that needs to have variables.

We generally avoid optimizations that are unrelated to memory management, unless they are very simple to describe and implement.

Unlike GP, P cannot be understood simply as a fancy syntax for certain restricted Prolog programs. It is a separate, rather low-level, language with

its own run-time restrictions that cannot easily be expressed in Prolog.

However, there *is* a translation back to Prolog that is valid and meaning-preserving for P programs that do not "go wrong" in ways we discuss below. In Chapter 7 we present a type system for P; P programs that can be typed in this type system are known not to go wrong. So we can consider the set of typeable programs to be a common subset of P and Prolog, and these are the programs we are really interested in.

3.1 The store model

P is marked out from Prolog by the fact that an explicit store model of term representations is needed to understand it. The store model we use is related to, but somewhat more abstract than, the heap in the WAM [Aït-Kaci 1991].

Let a countably infinite set $\langle Addr \rangle$ of **addresses** be given. A **store** is then a finite map from $\langle Addr \rangle$ to **data** which is defined thus:

$$\begin{array}{rcl} \alpha \in \langle Addr \rangle & & \\ \langle Datum \rangle & = & \biguplus_f \langle Addr \rangle^{|f|} \uplus \{\mathsf{uninst}\} \uplus \langle Addr \rangle \\ \sigma \in \langle Store \rangle & = & \langle Addr \rangle \xrightarrow{\text{fin}} \langle Datum \rangle \end{array}$$

The first of the three possibilities for $\langle Datum \rangle$ can be used to build ground terms in a hopefully obvious manner[1]. An element of $\biguplus_f \langle Addr \rangle^{|f|}$ is called a **structure** and written $f(\alpha_1, \ldots, \alpha_{|f|})$.

The same $\langle Addr \rangle$ can be referenced from several different places leading to a kind of "shared" representation common in implementations of functional languages. This must not be confused with the entirely different "structure sharing" concept used in some implementations of Prolog.

uninst stands for an uninstantiated logical variable. The variable is identified by the address at which the uninst is found; two uninsts stored at different addresses denote distinct variables.

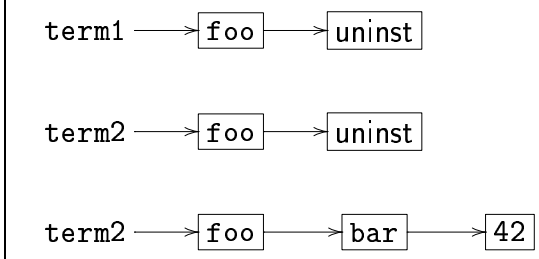
Finally a store cell with a value from $\langle Addr \rangle$ is an indirection that is left when a variable is instantiated. Because of the "sharing" we just mentioned it would be impractical to seek out and redirect all of the references to a variable when it is instantiated; rather the uninst marker is replaced with an reference to the term the variable is unified with. We call these indirections **instantiated variables**[2].

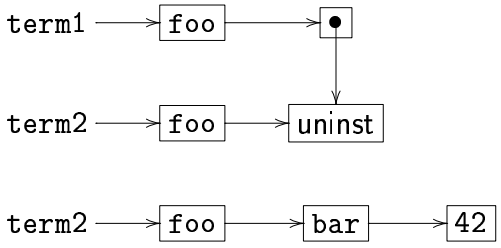
[1] You may note that we use a boxed representation of integers: the operands to a functor are all addresses, and integers live in store cells of their own. Thus, in the representation of the term `foo(42)` the store cell with the `foo` contains an *pointer* to the cell where the number 42 is stored. Similarly, when a register is bound to a number, the register really contains a pointer to a store cell where the actual number can be found.

There is no intrinsic reason why integers could not be unboxed (which would correspond roughly to letting integers be a special kind of $\langle Addr \rangle$ rather than a special kind of functor). We chose to box integers because that allows test programs to put extra stress on the memory management principles with a given amount of programming put into them.

[2] This might be contrary to some Prolog programmers' intuition, according to which variables cease to be variables when they are instantiated. We have no other good intuitive word for this concept, however.

```
example :- enter → ()
        & construct 42() → num
        & makevar → arg1
        & makevar → arg2
        & construct bar(num) → arg3
        & construct foo(arg1) → term1
        & construct foo(arg2) → term2
        & construct foo(arg3) → term3
        & unify term1 = term2
        & unify term2 = term3
        & exit (term1).
```





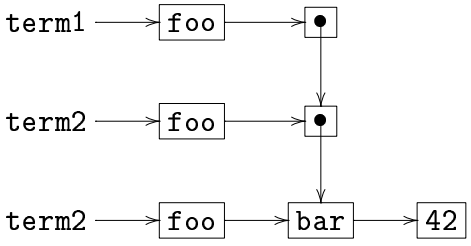


Figure 3.1: An example of how variables in terms behave. The three graphs show part of the store and the `term1`, `term2`, *and* `term3` *registers before, between, and after the* unify *instructions.*

Figure 3.1 shows examples of all four kinds of $\langle Datum \rangle$. Notice that after the second unification there is a chain of instantiated variables that need to be traversed when one wants to find the real operand to `term1`'s `foo` structure.

3.1.1 Local cycles in stores

Definition 3.1 *A **local cycle** in store σ is a cycle consisting entirely of instantiated variables. That is, $\alpha_0, \alpha_1, \ldots, \alpha_n = \alpha_0$ with $n > 0$ and $\sigma(\alpha_i) = \alpha_{i+1}$.*

In general we will only consider stores that have no local cycles. It can be checked in our operational semantics that no computation step can introduce a local cycle. In general, when we quantify over stores, it is implicit that the quantification is over stores that have no local cycles.

As long as we only consider stores without local cycles, the action of follow a chain of (possibly zero) instantiated variables to its end is well-defined:

Definition 3.2 $\mathsf{F}_\sigma : \langle Addr \rangle \to \langle Addr \rangle_\perp$ *is the function defined by*

$$\mathsf{F}_\sigma(\alpha) = \textbf{case } \sigma(\alpha) \textbf{ of } \begin{cases} \text{(not defined in } \sigma) & \mapsto & \perp \\ \alpha' \in \langle Addr \rangle & \mapsto & \mathsf{F}_\sigma(\alpha') \\ \text{(anything else)} & \mapsto & \alpha \end{cases}$$

Despite the recursive case, this is clearly well-defined as long as there are no local cycles in σ.

We use this function in the formal semantics of deref, and in Appendix A's construction of the meaning of stores.

3.1.2 The formal meaning of stores

In formulas we use the notation $\sigma[\![\alpha]\!]$ to mean the term represented by the pointer α in the store σ.

Theorem 3.3 *There is a countably infinite set of variable names $\mathbb{V}$ and a notion of "terms" over $\mathbb{V}$ (the variables t, t', t_i, etc. range over terms) such that for each store σ with no local cycles there is a function $\sigma[\![\,]\!]$ which maps addresses to terms such that*

- $\sigma[\![\alpha]\!] = \textbf{case } \sigma(\alpha) \textbf{ of } \begin{cases} f(\alpha_1, \ldots, \alpha_n) & \mapsto & f(\sigma[\![\alpha_1]\!], \ldots, \sigma[\![\alpha_n]\!]) \\ \alpha' \in \langle Addr \rangle & \mapsto & \sigma[\![\alpha']\!] \\ \mathsf{uninst} & \mapsto & \textit{some } X \in \mathbb{V} \end{cases}$
- $\sigma(\alpha_1) = \mathsf{uninst} \wedge \sigma(\alpha_2) = \mathsf{uninst} \Longrightarrow \alpha_1 \neq \alpha_2 \Rightarrow \sigma[\![\alpha_1]\!] \neq \sigma[\![\alpha_2]\!]$

If σ contains no cycles at all (local or otherwise), the notion of "terms" can be taken to be ordinary finite terms built from the chosen set of functors.

This theorem is obvious in the special case that σ is cycle-free: One can take $\mathbb{V} = \langle Addr \rangle$ and let $\sigma[\![\alpha]\!] = \alpha$ when $\sigma(\alpha) = \mathsf{uninst}$.

In the general case where σ may contain (non-local) cycles a more advanced notion of "term" than the usual one is necessary. We give one in Appendix A where a rigorous proof of the theorem also appears. (The reason why the definition and the proof has been postponed to an appendix is that they are somewhat involved mathematically and not directly relevant to regions).

3.2 The makevar instruction

The syntax for the makevar instruction is:

$$\langle instruction \rangle \quad ::= \quad \mathsf{makevar} \rightarrow \mathrm{x}$$

This instruction creates a fresh, uninstantiated variable and binds x to it.

In Prolog the instruction is invisible, or could perhaps be represented as the goal `makevar(X)`, where `makevar/1` is defined by the fact "`makevar(X).`".

3.3 The deref instruction

The syntax of the the deref instruction is

$$\langle instruction \rangle \quad ::= \quad \mathsf{deref}\ \mathrm{x} \rightarrow \mathrm{x}'$$

deref can be viewed as an elimination construct corresponding to the variable introduction of makevar.

If x is bound (perhaps indirectly) to an *un*instantiated variable, the deref instruction fails, causing backtracking to occur.

If x is bound, though a chain of zero or more instantiated variables, to a structure, the deref instruction succeeds. It also binds x' to that structure, without any intervening indirections.

The Prolog equivalent of the destruct instruction is

$$\texttt{nonvar}(\mathrm{X}), \mathrm{X}' = \mathrm{X}$$

(note that this does not capture the variable-chain traversal, because instantiated variables are invisible in Prolog).

3.4 The destruct instruction and variables

In P the destruct instruction has two restrictions that were not present in GP.

The first restriction arises as our response to the problem of what the effects of the instruction

$$\mathsf{destruct}\ \mathrm{x}\ \mathsf{as}\ \mathrm{f} \rightarrow (\mathrm{x}'_1,\ldots,\mathrm{x}'_{|\mathrm{f}|})$$

should be if x is bound to an uninstantiated variable. If we were to be true to the interpretation in Chapter 2 of destruct as a fancy way to write the Prolog goal $X = f(X'_1, ..., X'_{|f|})$, the only reasonable answer would be to bind the x'_is to fresh uninstantiated variables and instantiate the variable bound to x to a newly-created f structure consisting of the new variables.

This would, however, directly contradict our reasons for discriminating between construct and destruct in the first place, which was that construct must allocate memory and destruct must not.

The theory that follows would become messy and awkward if we allowed destruct to allocate memory. Instead we simply decide that

- *the operand to* destruct *must not be an uninstantiated variable.*

Violating this rule should not be just a failure that leads to backtracking; then some P programs would have an apparently valid behaviour that is different from the behaviour of the corresponding Prolog program. Instead we declare violations of the rule to be wrong_3, allocating a new suffix 3 for this kind of error. This decision means that it is still sound to let destruct correspond to $X = f(X'_1, ..., X'_n)$.

The type system TP we present in Chapter 7 can be used to prove that a P program never goes wrong_3; hence that it is equivalent to its corresponding Prolog program.

One way to avoid going wrong_3 would be to replace

```
  (...)
    &  destruct a as foo → (b,c)
  (...)
```

with

```
  (...)
    &  makevar → b
    &  makevar → c
    &  construct foo(b,c) → temp
    &  unify a = temp
  (...)
```

whenever there is a risk that a might be bound to an uninstantiated variable. This *always* allocates memory, though, no matter what a is bound to. The deref instruction offers a more economic replacement, as we shall see shortly.

The other restriction on destruct is primarily motivated by aesthetics. Consider the situation after the unifications on Figure 3.1 (page 34), and imagine that

```
  (...)
    &  destruct term1 as foo → (temp)
    &  destruct temp as frob → ( )
  (...)
```

were to follow. The first destruct instruction goes well, binding temp to the first of the instantiated variables. Now, the second destruct instruction has to traverse the chain of instantiated variables before it can find that the bar structure is not a frob and backtrack.

We think it is inelegant that the destruct instruction should do two separate tasks, so we set the rule that

- *It is* wrong_3 *for the argument register in a* destruct *instruction to be bound to an instantiated variable.*

This second rule would probably not be justifiable if any extra work were needed to ensure it was satisfied. It turns out, however, that the TP type system that protects against destruct on uninstantiated variables also protects against destruct on instantiated variables.

Under these circumstances we think that the elegance of letting destruct be a direct inverse to construct, and use a separate deref instruction for traversing instantiated-variable chains, should be allowed to govern the design choice.

3.5 Examples of P code

We can summarise P's syntax as follows. The scope rules are as outlined in Section 2.2.4.

$$
\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \langle \textit{predicate} \rangle_1 \ldots \langle \textit{predicate} \rangle_n \qquad (n \geq 1) \\
\langle \textit{predicate} \rangle & ::= & \mathrm{pr} \ :- \ \langle \textit{clauses} \rangle \\
\langle \textit{clauses} \rangle & ::= & \langle \textit{clause} \rangle \ . \\
 & | & \langle \textit{clause} \rangle \ ; \ \langle \textit{clauses} \rangle \\
\langle \textit{clause} \rangle & ::= & \mathsf{enter} \rightarrow (x_1,\ldots,x_{\lceil \mathrm{pr} \rceil}) \ \& \ \langle \textit{body} \rangle \\
\langle \textit{body} \rangle & ::= & \langle \textit{instruction} \rangle \ \& \ \langle \textit{body} \rangle \\
 & | & \mathsf{fail} \\
 & | & \mathsf{exit} \ (x_1,\ldots,x_{\lfloor \mathrm{pr} \rfloor}) \\
\langle \textit{instruction} \rangle & ::= & \mathsf{call} \ \mathrm{pr}'(x_1,\ldots,x_{\lceil \mathrm{pr}' \rceil}) \rightarrow (x_1,\ldots,x_{\lfloor \mathrm{pr}' \rfloor}) \\
 & | & \mathsf{builtin} \ \mathrm{pr}'(x_1,\ldots,x_{\lceil \mathrm{pr}' \rceil}) \rightarrow (x_1,\ldots,x_{\lfloor \mathrm{pr}' \rfloor}) \\
 & | & \mathsf{construct} \ f(x_1,\ldots,x_{|f|}) \rightarrow x \\
 & | & \mathsf{destruct} \ x \ \mathsf{as} \ f \rightarrow (x_1,\ldots,x_{|f|}) \\
 & | & \mathsf{makevar} \rightarrow x \\
 & | & \mathsf{deref} \ x \rightarrow x \\
 & | & \mathsf{alias} \ x \rightarrow x \\
 & | & \mathsf{unify} \ x{=}x \\
 & | & \mathsf{cut}
\end{array}
$$

We expand the "reverse" example program from Figure 2.1 (page 22) to a version that can also reverse a partially unknown list (giving multiple answers) yet only creates new variables if necessary. This new version is shown on Figure 3.2.

The difference from the GP version is that some of the destruct instructions (in the main `rev` predicate) has been replaced by calls to auxiliary

```
append([],B,B).
append([X|A],B,[X|C]) :- append(A,B,C).
rev([],[]).
rev([X|A],C) :- rev(A,Ar), append(Ar,[X],C)
```

```
match-nil :- enter → (var)
               & deref var → data
               & cut
               & destruct data as nil → ( )
               & exit ( );
             enter → (var)          % var is now known to be uninstantiated
               & construct nil( ) → temp
               & unify var = temp
               & exit ( ).
match-cons :- enter → (var)
                & deref var → data
                & cut
                & destruct data as cons → (d1, d2)
                & exit (d1, d2);
              enter → (var)         % var is now known to be uninstantiated
                & makevar → d1
                & makevar → d2
                & construct cons(d1, d2) → temp
                & unify var = temp
                & exit (d1, d2).
append :- enter → (p1, b)
            & destruct p1 as nil → ( )
            & cut
            & alias b → r3
            & exit (r3);
          enter → (p1, b)
            & destruct p1 as cons → (x, a)
            & call append (a, b) → (c)
            & construct cons(x, c) → r3
            & exit (r3).
rev :- enter → (p1)
         & call match-nil (p1) → ( )
         & construct nil( ) → r2
         & exit (r2);
       enter → (p1)
         & call match-cons (p1) → (x, a)
         & call rev (a) → (ar)
         & construct nil( ) → t1
         & construct cons(x, t1) → t2
         & call append (ar, t2) → (c)
         & exit (c).
```

Figure 3.2: The naïve reverse function in Prolog, and a P translation that can reverse a known or (partially) unknown list.

`match-nil` and `match-cons` predicates that use deref to check whether the argument is an uninstantiated variable and choose among according implementations of the original $P_1 = []$ and $P_1 = [X|A]$ goals. (The get-structure instructions in the WAM [Aït-Kaci 1991] can be viewed as an optimized implementation of this idea.)

The cut in `rev` is gone because an uninstantiated variable should be allowed to match both clauses.

We assume that the compiler, like our prototype implementation, is smart enough to realise that the "spine" of the lists that are arguments to `append` can never contain variables (even though the list's *elements* may be variables), and thus use the efficient GP implementation of `append`.

3.6 Unification in the store model

We have skipped over an important point in the previous development: how does the unify instruction react to variables? We can see examples of its behaviour on Figure 3.1 and from the implied working of `match-cons` on Figure 3.2. Both of these examples support the obvious notion that unify does the instantiations that are necessary to make $\sigma[\![x]\!]$ and $\sigma[\![x']\!]$ identical.

However, if we need to do any kind of formal reasoning about P, we need a more precise statement than that. The task of this section is to give one, phrased in our store model rather than the usual formalism of first-class terms and substitutions.

3.6.1 Basic definitions

In the definitions below, it is tacitly assumed that none of the stores have local cycles or contain references to undefined addresses.

Definition 3.4 *A store* σ_2 ***extends*** σ_1*, notated* $\sigma_2 \blacktriangleright \sigma_1$ *iff* $\mathrm{Dom}\,\sigma_1 \subseteq \mathrm{Dom}\,\sigma_2$ *and*

$$\forall \alpha \in \mathrm{Dom}\,\sigma_1 : \ \sigma_1(\alpha) \neq \sigma_2(\alpha) \Longrightarrow \sigma_1(\alpha) = \mathsf{uninst} \wedge \sigma_2(\alpha) \in \mathrm{Dom}\,\sigma_2.$$

It is easy to see that $\blacktriangleright$ *is a partial order on stores.*

An extension $\sigma_2 \blacktriangleright \sigma_1$ defines a substitution θ by $\theta(\sigma_1[\![\alpha]\!]) = \sigma_2[\![\alpha]\!]$. This definition of θ agrees with the usual extension of substitutions from variables to terms.

It is not difficult to see that if σ_1 is given together with and a substitution θ with $\mathrm{Dom}\,\theta \subseteq \mathrm{Codom}(\sigma_1[\![\,]\!])$, we can always find an extension $\sigma_2 \blacktriangleright \sigma_1$ such that the substitution defined by $\sigma_2 \blacktriangleright \sigma_1$ is identical to θ up to variable renaming.

Definition 3.5 *We call two stores* σ_1 *and* σ_2 ***similar****, notated* $\sigma_1 \sim \sigma_2$*, iff* $\mathrm{Dom}\,\sigma_1 = \mathrm{Dom}\,\sigma_2$ *and* $\forall \alpha \in \mathrm{Dom}\,\sigma_1 : \ \sigma_1[\![\alpha]\!] = \sigma_2[\![\alpha]\!]$.

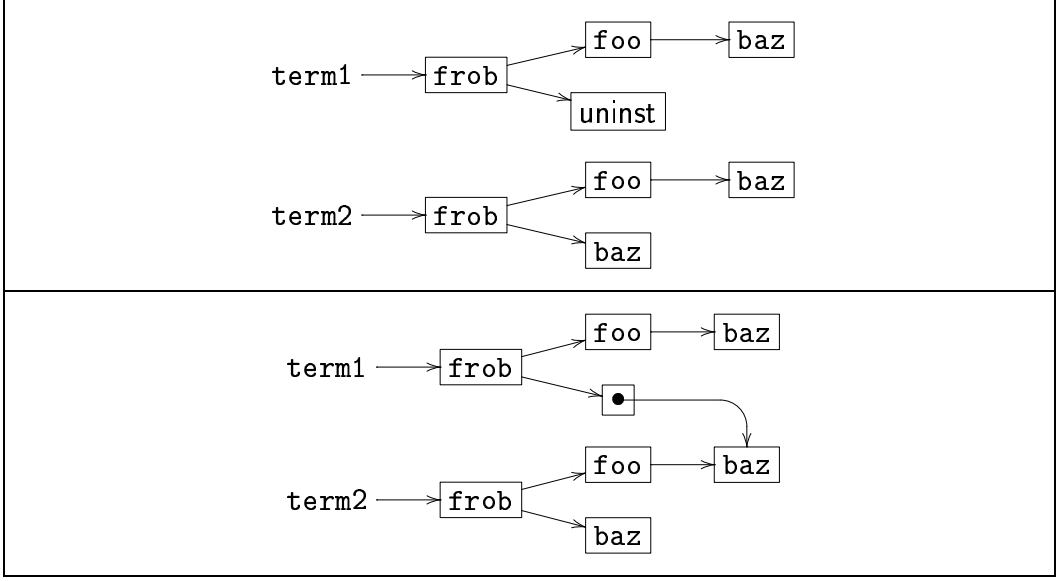


Figure 3.3: A most general unifier that ought not to be valid in P. The lower of the depicted stores is a most general unifier of `term1` *and* `term2` *in the topmost store, but not one that the* unify *instruction should be allowed to create.*

$\sim$ is obviously an equivalence relation

Definition 3.6 *A store σ_2 is **less general** than σ_1 if there is a store σ' that extends σ_1 and is similar to σ_2.*

Definition 3.7 *A **unifier** of α_0 and α_{00} in σ_0 is a store $\sigma_\omega \blacktriangleright \sigma_0$ such that $\sigma_\omega[\![\alpha_0]\!] = \sigma_\omega[\![\alpha_{00}]\!]$.*

Definition 3.8 *A **most general unifier** of α_0 and α_{00} in σ_0 is a unifier σ_ω with the property that any unifier is less general than σ_ω.*

3.6.2 Store-based unification, first attempt

Intuitively, it is reasonable to expect the instruction unify x_1=x_2 to either fail or compute a most general unifier of α_1 and α_2 (which are the addresses in the registers x_1 and x_2) in the current store.

It can be proved that whenever α_1 and α_2 have a unifier they also have a most general unifier (in fact, we prove this in Appendix A). Therefore it is tempting to simply use that description as the definition of P unification.

It turns out, however, that this definition is not precise enough when memory management is an issue. Consider the example in Figure 3.3. The store shown at the lower half of the figure *is*, according to the definitions in the previous section, a most general unifier of `term1` and `term2`, but it would wreak total havoc with the memory management if the unify instruction was allowed to produce it. The variable has been instantiated to *a* `baz`, but not *the* `baz` we would have expected. In fact, if all we knew about unify was that it produced a most general unifier, it could pick *any* `baz` anywhere in

the store, and we might need to be very conservative about the lifetimes of bazs.

The morale of this example is that when it comes to memory management, similar stores may have different properties. Because we have only defined most general unifiers up to similarity (and this is essential for obtaining a meaningful "most general" property), they are not enough to define what can be reasonably expected from the unify instruction.

We need to take a more operational approach to defining unification. One extreme strategy would be to fix one particular unification algorithm, but that would appear to be too specific. A proof, based on such a definition, that region inference is safe would tell nothing about region inference in a system where another algorithm is used. And at the detail level we are working at, even trivial changes to the algorithm, such as considering the operands to a functor in reverse order, may lead to a significantly different result.

3.6.3 Store-based unification, second attempt

We choose to specify P unification by the following nondeterministic algorithm. (Note that the nondeterminism involved here is "don't-care" nondeterminism, which is different from the kind of nondeterminism that is realised by backtracking in Prolog. Our definition simply allows an oracle to choose arbitrarily among a range of results; the possible results not chosen are simply forgotten, and the unify instruction never leaves a choice point).

Definition 3.9 *Define a binary relation* $\rhd$ *in*

$$\{(\sigma, \mathcal{C}) \mid \sigma \in \langle \mathit{Store} \rangle, \mathcal{C} \subseteq \langle \mathit{Addr} \rangle \times \langle \mathit{Addr} \rangle\}$$

by

a. If $(\alpha_1, \alpha_2) \in \mathcal{C}$
then $(\sigma, \mathcal{C}) \rhd (\sigma, \mathcal{C} \cup \{(\alpha_2, \alpha_1)\})$

b. If $(\alpha_1, \alpha_2) \in \mathcal{C}$ *and* $\sigma(\alpha_j) = f(\alpha_{j1}, \ldots, \alpha_{j|f|})$ *for* $j = 1, 2$
then $(\sigma, \mathcal{C}) \rhd (\sigma, \mathcal{C} \cup \{(\alpha_{11}, \alpha_{21})\})$

c. If $(\alpha_1, \alpha_2) \in \mathcal{C}$ *and* $\sigma(\alpha_1) \in \langle \mathit{Addr} \rangle$
then $(\sigma, \mathcal{C}) \rhd (\sigma, \mathcal{C} \cup \{(\sigma(\alpha_1), \alpha_2)\})$

d. If $(\alpha_1, \alpha_2) \in \mathcal{C}$, $\sigma(\alpha_1) = \mathsf{uninst}$, $\sigma(\alpha_2) \notin \langle \mathit{Addr} \rangle$, *and* $\alpha_1 \neq \alpha_2$
then $(\sigma, \mathcal{C}) \rhd (\sigma\{\alpha_1 \mapsto \alpha_2\}, \mathcal{C})$

Definition 3.10 *Let* $\rhd^*$ *be the reflexive, transitive closure of* $\rhd$.

Now, the result of unifying α_1 and α_2 in σ_0 can be computed by

1. If there exists $(\sigma, \mathcal{C})$ with $(\sigma_0, \{(\alpha_1, \alpha_2)\}) \rhd^* (\sigma, \mathcal{C})$ and $(\alpha', \alpha'') \in \mathcal{C}$ and $\alpha' \notin \mathrm{Dom}\,\sigma$, then the unification may go wrong, attempting to follow the dangling pointer. As described later (page 44) we use the suffix 4 for this kind of wrong: wrong_4.

2. The oracle may chose a σ such that $(\sigma_0, \{(\alpha_1, \alpha_2)\}) \rhd\!\!-^* (\sigma, \mathcal{C})$ for some $\mathcal{C}$, and decide that the unification succeeds with σ as the result.
3. The oracle may decide that α_1 and α_2 are not unifiable. In that case the result of the unification is fail.

This definition allows the oracle to perform normal unification as well as non-standard interpretations of unification. The non-standard interpretations include concepts such as the "half-match unification" proposed by Gabriel et al. [1984, Section 12.2] as well as totally degenerate strategies that, say, always fail or always succeed with an unchanged store.

The safety properties for region inference that we aim to eventually establish should hold for non-standard unification interpretations, but the observed behaviour of programs could be "incorrect" if a non-standard interpretation is used.

So we define a **cyclic standard interpretation** to be an oracle that always chooses a store that unifies α_1 and α_2 if it lets the unification succeed, and that only lets the unification fail if it is impossible to choose such a store. Our prototype implementation uses a cyclic standard interpretation to realise the unify instruction.

An **occurs-checking standard interpretation** is like a cyclic standard interpretation, except that it fails if the store that unifies α_1 and α_2 contains cycles.

In Appendix A we rigorously prove

Theorem 3.11 (Correctness of the unification algorithm) *Any unifier produced by our unification algorithm is most general. If there is any unifier of α_1 and α_2, then our algorithm can find a unifier.*

Thus, an occurs-checking standard interpretation corresponds directly to the most-general-unifier functions commonly used in Prolog semantics.

3.6.4 Comparison with other unification algorithms

Our definition of unification is essentially equivalent[3]. to the nondeterministic algorithm of Martelli and Montanari [1982, Section 2], when the pairs in $\mathcal{C}$ are interpreted as representing equations. This means that any of the unification algorithms they cite as special cases of theirs are also a special case of our algorithm and could be used for implementing unification in an implementation of P.

One can also view our algorithm directly as a generalisation of simple iterative imperative unification algorithms such as the one given by Aït-Kaci [1991, Figure 2.7]. Given a trace of a single run of Aït-Kaci's algorithm, it is easy to construct a matching $\rhd\!\!-$ sequence, maintaining the invariant that the

[3]One difference is that we do not remove equations from the $\mathcal{C}$ after they have been rewritten, but that is an insignificant technicality that does not affect the set of possible end stores.

set of pairs on the unification stack is always a subset of $\mathcal{C}$, and that (d_1, d_2) is also always in $\mathcal{C}$.

A similar technique can be used to simulate other unification algorithms that are based on synchronous depth-first (or breadth-first, for that matter) traversals of the terms to be unified [Manna and Waldinger 1981; Robinson 1971].

Multiequation-based unification algorithms such as those discussed by Martelli and Montanari [1982] can not be modelled by our definition, because a "transitivity" rule such as

$$(\sigma, \mathcal{C} \supseteq \{(\alpha_1, \alpha_2), (\alpha_2, \alpha_3)\}) \rightarrowtriangle (\sigma, \mathcal{C} \cup \{(\alpha_1, \alpha_3)\})$$

seems to be necessary to move between "corresponding" sets of equations. We cannot admit such a rule, because it would cause Theorem 7.5 to fail. (This is interesting: apparently Martelli and Montanari's multiequation-based algorithm is not a special case of their own nondeterministic algorithm, because the latter does not contain a rule of transitivity).

3.7 Operational semantics for P

We extend the semantics of Section 2.5 to cover all of P.

The primary purpose of this semantics (and in particular the RP semantics in Chapter 4 that builds on this one) is not to define the observable effects of running P programs, but to describe which memory allocations, references, and deallocations takes place as the programs runs. This is needed both to understand regions properly, and to prove that region inference is safe.

The connection between this semantics for P and the Prolog translations of the P instructions we have given is

Theorem 3.12 *When the* unify *instruction is implemented by a occurs-checking standard interpretation of our unification specification, it holds that if the P program does not go* wrong*, its behaviour under this semantics is the same as the Prolog version's behaviour under one of the existing Prolog semantics.*

As for GP, we omit the proof, as it would be somewhat tedious and not give any insights deeply relevant to region-based memory management.

See Figure 3.4 for the kinds of objects that are used in the P semantics. Each ⟨*Frame*⟩ in a state now contains its own ⟨*Store*⟩. Operationally, we imagine that the store in the first frame of the state is the *current* store; when that frame is discarded by backtracking, the current store gets replaced with one that was saved on frame stack earlier.

This gives a clear formal description of what backtracking does to variables, but it becomes the implementation's problem to maintain enough data at run-time to be able to get back to the stored state.

We add a new kind of wrong, wrong_4, to signal an attempt to reference an address that does not exist in the current store, a "dangling pointer". It

$$
\begin{array}{lcll}
\alpha & \in & \langle Addr \rangle & = \text{ an abstract countably infinite set} \\
 & \in & \langle Datum \rangle & = \biguplus_{f} \langle Addr \rangle^{|f|} \uplus \langle Addr \rangle \uplus \{\mathsf{uninst}\} \\
\sigma & \in & \langle Store \rangle & = \langle Addr \rangle \xrightarrow[\text{fin}]{} \langle Datum \rangle \\
\mathfrak{x} & \in & \langle Register \rangle & \\
\mathcal{E} & \in & \langle Env \rangle & = \langle Register \rangle \xrightarrow[\text{fin}]{} \langle Addr \rangle \\
\mathfrak{b} & \in & \langle body \rangle & \\
\mathfrak{c} & \in & \langle clause \rangle & \\
\mathfrak{C} & \in & \langle clauses \rangle & \\
\kappa & \in & \langle Cont \rangle & = (\langle ValEnv \rangle \times \langle Register \rangle^{*} \times \langle body \rangle \times \langle Dump \rangle)^{*} \\
\mathfrak{fr} & \in & \langle Frame \rangle & = \langle Store \rangle \times \langle ValEnv \rangle \times \langle body \rangle \times \langle Dump \rangle \times \langle Cont \rangle \\
\mathfrak{fs} & \in & \langle Frames \rangle & = \langle Frame \rangle^{*} \\
\delta & \in & \langle Dump \rangle & = \langle Frames \rangle \\
\mathfrak{s} & \in & \langle State \rangle & = \langle Frames \rangle \uplus \{\mathsf{ok}, \mathsf{wrong}_{1,2,3,4}\}
\end{array}
$$

Figure 3.4: Objects used in the P semantics

is straightforward to prove that P programs never go wrong_4: the transition rules all preserve the property that every address that is known in a given state will be defined in the store that appears in the containing $\langle Frame \rangle$.

3.7.1 The initial state

$$s_0 = [(\varnothing, \varnothing, \mathsf{call}\ \mathfrak{pr}_0() \rightarrow () \ \&\ \mathsf{fail}, [])]$$

3.7.2 The transition relation

The successor of $[]$ is ok. Any other non-final state has the shape

$$(\sigma, \mathcal{E}, \mathfrak{b}, \delta, \kappa) :: \mathfrak{fs}$$

and we now proceed by case analysis on $\mathfrak{b}$.

The successor state is implicitly wrong_1 if the computation of it would involve looking up a register name in a environment that does not define it, or extending a environment with a binding for a register name that it already defines. (These checks were explicit parts of the GP semantics, but we don't think there is any good reason to repeat them over and over).

$\mathfrak{b} =$ $\mathsf{call}\ \mathfrak{pr}(\mathfrak{x}_1,\ldots,\mathfrak{x}_{\lceil \mathfrak{pr} \rceil}) \rightarrow (\mathfrak{x}'_1,\ldots,\mathfrak{x}'_{\lfloor \mathfrak{pr} \rfloor}) \ \&\ \mathfrak{b}'$

- Let

  $$\mathfrak{pr} :- \mathsf{enter} \rightarrow (\mathfrak{x}_{11},\ldots,\mathfrak{x}_{1\lceil \mathfrak{pr} \rceil}) \ \&\ \mathfrak{b}_1 ; \ \ldots \ ; \ \mathsf{enter} \rightarrow (\mathfrak{x}_{k1},\ldots,\mathfrak{x}_{k\lceil \mathfrak{pr} \rceil}) \ \&\ \mathfrak{b}_k .$$

  be the definition of $\mathfrak{pr}$.
- $\mathcal{E}_i \leftarrow \{\mathfrak{x}_{ij} \mapsto \mathcal{E}(\mathfrak{x}_j) \mid 1 \leq j \leq \lceil \mathfrak{pr} \rceil\}$ for $1 \leq i \leq k$.

- $\delta' \leftarrow \mathfrak{fs}$
- $\kappa' \leftarrow (\mathcal{E}, (x'_1, \ldots, x'_{\lfloor pr \rfloor}), b', \delta) :: \kappa$
- $\mathfrak{fr}_i \leftarrow (\sigma, \mathcal{E}_i, b_i, \delta', \kappa')$ for $1 \le i \le k$
- The next state is $\mathfrak{fr}_1 :: \mathfrak{fr}_2 :: \ldots :: \mathfrak{fr}_k :: \mathfrak{fs}$

$b =$ builtin $pr(x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr \rfloor})$ & b'

The implementation is supposed to provide a function

$$\phi_{pr} : \mathbb{Z}^{\lceil pr \rceil} \rightarrow \left(\mathbb{Z}^{\lfloor pr \rfloor} \uplus \{\mathsf{fail}, \bot\}\right)$$

- If any $\mathcal{E}(x_i) \notin \mathrm{Dom}\,\sigma$ then wrong_4; else
- If any $\sigma(\mathcal{E}(x_i)) \notin \mathbb{Z}$ then wrong_2; else
- Compute $\phi_{pr}(\sigma(\mathcal{E}(x_1)), \ldots, \sigma(\mathcal{E}(x_{\lceil pr \rceil})))$.
  If the result is fail, then the next state is $\mathfrak{fs}$.
  If the result is $\bot$, then the current state is its own successor.
  Otherwise, let the result be $(n_1, \ldots, n_{\lfloor pr \rfloor})$.
- $\alpha_i \leftarrow$ fresh $\notin \mathrm{Dom}\,\sigma$ (for $1 \le i \le \lfloor pr \rfloor$; $\alpha_i \neq \alpha_j$)
- $\sigma' \leftarrow \sigma\{\alpha_i \mapsto n_i \mid 1 \le i \le \lfloor pr \rfloor\}$
- $\mathcal{E}' \leftarrow \mathcal{E}\{x'_i \mapsto \alpha_i \mid 1 \le i \le \lfloor pr \rfloor\}$
- The next state is $(\sigma', \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ construct $f(x_1,\ldots,x_{|f|}) \rightarrow x'$ & b'

- $\alpha \leftarrow$ fresh $\notin \mathrm{Dom}\,\sigma$
- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \alpha\}$
- $\sigma' \leftarrow \sigma\{\alpha \mapsto f(\mathcal{E}(x_1), \ldots, \mathcal{E}(x_{|f|}))\}$
- The next state is $(\sigma, \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ destruct x as $f \rightarrow (x'_1,\ldots,x'_{|f|})$ & b'

- If $\mathcal{E}(x) \notin \mathrm{Dom}\,\sigma$ then wrong_4; else
- If $\sigma(\mathcal{E}(x)) \in \langle \mathit{Addr} \rangle \cup \{\mathsf{uninst}\}$ then wrong_3; else
- If not $\mathcal{E}(x) = f(\alpha_1, \ldots, \alpha_{|f|})$ for some α_is, then the next state is $\mathfrak{fs}$; else
- $\mathcal{E}' \leftarrow \mathcal{E}\{x'_i \mapsto \alpha_i \mid 1 \le i \le |f|\}$
- The next state is $(\sigma, \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ makevar $\rightarrow x'$ & b'

- $\alpha \leftarrow$ fresh $\notin \mathrm{Dom}\,\sigma$
- $\sigma' \leftarrow \sigma\{\alpha \mapsto \mathsf{uninst}\}$
- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \mathsf{ref}\ \alpha\}$
- The next state is $(\sigma', \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ deref $x \rightarrow x'$ & b'

- If $\mathsf{F}_\sigma(\mathcal{E}(x)) = \bot$ then wrong_4; else
- $\alpha \leftarrow \mathsf{F}_\sigma(\mathcal{E}(x))$
- If $\sigma(\alpha) = \mathsf{uninst}$ then the next state is $\mathfrak{fs}$; else

- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \alpha\}$
- The next state is $(\sigma, \mathcal{E}', b', \delta, \kappa) :: fs$

$b =$ alias $x \rightarrow x'$ & b'

- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \mathcal{E}(x)\}$
- The next state is $(\sigma, \mathcal{E}', b', \delta, \kappa) :: fs$

$b =$ unify x_1=x_2 & b'

- Try to unify $\mathcal{E}(x_1)$ and $\mathcal{E}(x_2)$ in store σ according to the description in Section 3.6.3.
  If the result is wrong_4 then that is the next state; if the result is fail then the next state is fs; else the result is a new store σ'.
- The next state is $(\sigma', \mathcal{E}, b', \delta, \kappa) :: fs$

$b =$ cut & b'

- The next state is $(\sigma, \mathcal{E}, b', \delta, \kappa) :: \delta$

$b =$ fail

- The next state is fs

$b =$ exit $(x_1,\ldots,x_n)$

- $(\mathcal{E}', (x'_1, \ldots, x'_n), b', \delta') :: \kappa' \leftarrow \kappa$
- $\mathcal{E}'' \leftarrow \mathcal{E}'\{x'_i \mapsto \mathcal{E}(x_i) \mid 1 \leq i \leq n\}$
- The next state is $(\sigma, \mathcal{E}'', b', \delta', \kappa') :: fs$

Chapter 4

RP: Adding regions to P

In this chapter we add explicit region-based memory-management instructions to P, forming the language RP. The memory we'll use regions to manage is the memory that was modelled by the store in the P semantics.

4.1 Key decisions

This section presents the key decisions that define our approach to adapting regions to logic programming. If other decisions were made here, the resulting theory could look very different. We have not investigated exactly what consequences other decisions would have; but we think that our decisions are reasonable for a first attempt at using region-based memory management for Prolog.

If you are not familiar with regions, please review the compact introduction in Section 1.1.5ff before reading further on.

4.1.1 Transparent backtracking

We decide to define the region-based language RP without regard for backtracking. Backtracking is supposed to be handled by the RP implementation in a transparent way. That is, the RP program can kill some "old" regions and create some "new" ones. If it then backtracks to a point before the "old" regions were killed, the RP implementation is responsible for making them reappear with their contents unharmed. The RP implementation should also kill the "new" regions when the program backtracks past their creation. Additionally, the backtracking process should shrink any surviving regions to the size they had when the choice point was created. (And of course, the backtracking should also undo any variable instantiations that were done after the choice point was created).

In short, for the creator of a RP program, as far as memory management is concerned backtracking *is* a time machine.

The advantages of this decision should be obvious. The region inference can simply assume that backtracking never occurs and that a clause is chosen at random when a predicate is called. There is no need for using intricate

determinacy analyses to find out about the program's real-time control-flow patterns so as not to kill regions prematurely.

From the viewpoint of the region inference, the input program is simply a first-order functional language which sometimes makes a nondeterministic choice, and which has logical variables that are sometimes instantiated by unify instructions.

This view of the program corresponds to a certain subset of ML: The logical variables can be considered a weak form of ML references that are only updated once. This property suggests that the region inference algorithms that have been developed for ML [Tofte and Birkedal 1998] should be transferable to RP without major effort. Indeed, the main complication of Tofte and Birkedal is the handling of higher-order functions which P don't have, so the prospects for an RP region inference are excellent.

On the other hand, this decision can also be seen as simply pushing the hard problems into to the implementation. It is worthless unless there is an efficient implementation of the backtracking-as-time-machine concept for regions. In Chapter 5 we develop such an implementation and hope it will be efficient in practise.

It is an important fact that the problems to be solved for region inference and the problems to be solved for the RP implementation are completely disjoint once it has been decided to handle regions transparently. As long as both subproblems are solvable, this is our basic evidence that the decision is sound. A solution that did not make this division might be more efficient than ours but it would most probably also be more complicated, thus not a good candidate for a first investigation of how to adapt regions to Prolog.

4.1.2 The lifetime of regions

We now turn to the question of how the allocation and deallocation points for each region can be related.

Obviously, a region must be allocated before it can be deallocated, but it is common to impose additional restrictions. Tofte and Talpin [1993] and most of subsequent work on regions dictate that the lifetime of a region must coincide with the evaluation of a specific expression in the program. This does not seem unreasonable to the programmer, because Standard ML already has a hierarchic expression structure.

However, Aiken et al. [1995] showed that the memory usage of certain programs can be asymptotically better if this restriction is lifted. Also, Prolog does not have the strictly hierarchic syntax that makes the restriction natural for Standard ML (and in P, even the hierarchic syntax of terms has been sequentialised). These considerations support the idea that we should be more liberal about where regions can be allocated and deallocated.

We decide on the principle that

- *The create and kill points for each region should be lexically given points in the same clause, but there needs not be any relationship between the create and kill points for different regions.*

This is stronger (*i.e.*, it allows more region-usage patterns) than Tofte and Talpin's region inference rules. It is weaker than the system of Aiken et al., because the latter allows a procedure to create a region and leave it to its caller to later kill it, or to kill a region created by its caller.

The primary reason why we chose to use a weaker system than Aiken et al. is that we haven't yet had time to investigate how best to make use the added liberty in the region inference phase. Certainly Aiken et al.'s constraint-based analysis could be easily adapted to work in our framework, but we're not sure it's the most elegant solution.[1]

4.2 Extensions to the P syntax

We can now extend P's syntax to support explicit region-based memory management.

We start by adding a new class of identifiers, **region registers**. Following standard notation, we use the Greek letter ρ to stand for region registers.

4.2.1 The makeregion instruction

The first construct we have to be able to express is the creation and killing of a region. The makeregion instruction creates a new region and binds it to a region names.

$$\langle \textit{instruction} \rangle \quad ::= \quad \mathsf{makeregion} \rightarrow \rho$$

As for register names, we assume that there is at most one makeregion instruction for each region name in the program. The scope of the region register is the part of the clause that follows the makeregion instruction, until the corresponding killregion is met.

4.2.2 The killregion instruction

$$\langle \textit{instruction} \rangle \quad ::= \quad \mathsf{killregion}\ \rho$$

This instruction marks the point in a clause where the named region should be killed. There must be exactly one killregion instruction to match each makeregion instruction in the program, and it must occur in the same clause, and after, the makeregion instruction.

(As an exception, in a clause that ends in a fail instruction rather than an exit instruction, makeregions need not be matched by killregion instructions. What is important is that when a predicate *returns* it has killed all of the regions it created. If a clause ends in fail it does not return, so that is not an issue).

[1] See also Section 12.1.1.

4.2.3 Additions to the builtin, construct, and makevar instructions

The instructions that allocate memory are the builtin, construct, and makevar instructions. We need to extend the syntax of these with an indication of which region to allocate the structure or variable in:

$$\begin{array}{rcl} \langle \textit{instruction} \rangle & ::= & \mathsf{builtin}\ pr'(x_1,\ldots,x_{\lceil pr' \rceil}) \rightarrow (x_1\ \mathsf{at}\ \rho_1,\ldots,x_{\lfloor pr' \rfloor}\ \mathsf{at}\ \rho_{\lfloor pr' \rfloor}) \\ & | & \mathsf{construct\ at}\ \rho\ f(x_1,\ldots,x_{|f|}) \rightarrow x \\ & | & \mathsf{makevar\ at}\ \rho \rightarrow x \end{array}$$

4.2.4 Additions to the predicate-call mechanism

We need to let a predicate allocate memory in one of the calling predicate's regions. This is essential for predicates that allocate data that need to survive after the predicate itself returns. Intuitively, the caller must pass the predicate references to the regions where it wants the data structures to be allocated. This means that we must extend the syntax for predicate calls with **region parameters**. Region parameters are like normal input parameters, except that they carry region bindings between region registers instead of carrying term bindings between normal registers.

Again following traditional notation, region parameters are written in a separate parameter tuple notated with square brackets.

We assume that each RP predicate name pr has a **region arity** $[pr]$. The new syntax is:

$$\begin{array}{rcl} \langle \textit{predicate} \rangle & ::= & pr[\rho_1,\ldots,\rho_{[pr]}]\ \texttt{:-}\ \langle \textit{clauses} \rangle \\ \langle \textit{instruction} \rangle & ::= & \mathsf{call}\ pr'[\rho_1,\ldots,\rho_{[pr']}]\ (x_1,\ldots,x_{\lceil pr' \rceil}) \rightarrow (x_1,\ldots,x_{\lfloor pr' \rfloor}) \end{array}$$

There is not any deep reason why we let the formal region parameters appear in the ⟨*predicate*⟩ definition and not in the enter pseudoinstruction along with the formal value parameters. It simply turned out to be technically convenient in our prototype implementation to do it that way.

The scope of a formal region parameter is the entire definition of the predicate. A formal region parameter may not appear in a makeregion or killregion instruction.

4.3 Examples of RP code

We summarise RP's syntax as in Figure 4.1. The scope rules are as outlined in Sections 2.2.4, 4.2.1, and 4.2.4.

Figure 4.2 shows a region-annotated version of the P reverse function from Figure 3.2 (page 39). Here `rev` has three region parameters:

`PS`: This is the region where the cons cells of the input list are allocated. `rev` needs to know this if it finds an uninstantiated variable in the input list and hence needs to allocate a list structure itself.

$$
\begin{array}{rcl}
\langle \mathit{program} \rangle & ::= & \langle \mathit{predicate} \rangle_1 \ldots \langle \mathit{predicate} \rangle_n \qquad (n \geq 1) \\
\langle \mathit{predicate} \rangle & ::= & pr[\rho_1,\ldots,\rho_{\lfloor pr \rfloor}] \ :- \ \langle \mathit{clauses} \rangle \\
\langle \mathit{clauses} \rangle & ::= & \langle \mathit{clause} \rangle \ . \\
& | & \langle \mathit{clause} \rangle \ ; \ \langle \mathit{clauses} \rangle \\
\langle \mathit{clause} \rangle & ::= & \mathsf{enter} \rightarrow (x_1,\ldots,x_{\lceil pr \rceil}) \ \& \ \langle \mathit{body} \rangle \\
\langle \mathit{body} \rangle & ::= & \langle \mathit{instruction} \rangle \ \& \ \langle \mathit{body} \rangle \\
& | & \mathsf{fail} \\
& | & \mathsf{exit}\ (x_1,\ldots,x_{\lfloor pr \rfloor}) \\
\langle \mathit{instruction} \rangle & ::= & \mathsf{call}\ pr'[\rho_1,\ldots,\rho_{\lceil pr' \rceil}]\ (x_1,\ldots,x_{\lceil pr' \rceil}) \rightarrow (x_1,\ldots,x_{\lfloor pr' \rfloor}) \\
& | & \mathsf{builtin}\ pr'(x_1,\ldots,x_{\lceil pr' \rceil}) \rightarrow (x_1\ \mathsf{at}\ \rho_1,\ldots,x_{\lfloor pr' \rfloor}\ \mathsf{at}\ \rho_{\lfloor pr' \rfloor}) \\
& | & \mathsf{construct\ at}\ \rho\ f(x_1,\ldots,x_{|f|}) \rightarrow x \\
& | & \mathsf{destruct}\ x\ \mathsf{as}\ f \rightarrow (x_1,\ldots,x_{|f|}) \\
& | & \mathsf{makevar\ at}\ \rho \rightarrow x \\
& | & \mathsf{deref}\ x \rightarrow x \\
& | & \mathsf{alias}\ x \rightarrow x \\
& | & \mathsf{unify}\ x{=}x \\
& | & \mathsf{cut} \\
& | & \mathsf{makeregion} \rightarrow \rho \\
& | & \mathsf{killregion}\ \rho
\end{array}
$$

Figure 4.1: The grammar for RP

`E`: This is the region where the list data are allocated. `rev` only needs this if it finds an uninstantiated variable in the input list; then it allocates variables to represent unknown elements in region `E`.

`RS`: This is where the cons cells that make up the resulting list will be allocated. The actual list elements are shared with the input list.

`rev` itself creates a region `TS` where the intermediate list `ar` is allocated. As soon as this list has been traversed by `append` the region is killed, freeing the storage used by the intermediate list.

4.4 Operational semantics for RP

We extend the semantics of Section 3.7 to model the region-based allocation and deallocation of structures and variable cells.

The primary purpose of this semantics is to allow reasoning about when RP programs commit the fatal error of trying to inspect a piece of memory that has been deallocated. If that happen, the semantics will let the program go wrong_4.

```
match-nil[R0]  :- enter → (var)
                    &  deref var → data
                    &  cut
                    &  destruct data as nil → ( )
                    &  exit ( );
                  enter → (var)
                      %   var is now known to be uninstantiated
                    &  construct at R0 nil( ) → temp
                    &  unify var = temp
                    &  exit ( ).
match-cons[R0, R1, R2]  :- enter → (var)
                             &  deref var → data
                             &  cut
                             &  destruct data as cons → (d1, d2)
                             &  exit (d1, d2);
                           enter → (var)
                               %   var is now known to be uninstantiated
                             &  makevar at R1  → d1
                             &  makevar at R2  → d2
                             &  construct at R0 cons(d1, d2) → temp
                             &  unify var = temp
                             &  exit (d1, d2).
append[RS]  :- enter → (p1, b)
                 &  destruct p1 as nil → ( )
                 &  cut
                 &  alias b → r3
                 &  exit (r3);
               enter → (p1, b)
                 &  destruct p1 as cons → (x, a)
                 &  call append [RS] (a, b) → (c)
                 &  construct at RS cons(x, c) → r3
                 &  exit (r3).
rev[PS, E, RS]  :- enter → (p1)
                    &  call match-nil [PS] (p1) → ( )
                    &  construct at RS nil( ) → r2
                    &  exit (r2);
                  enter → (p1)
                    &  call match-cons [PS, E, PS] (p1) → (x, a)
                    &  makeregion  → TS
                    &  call rev [PS, E, TS] (a) → (ar)
                    &  construct at RS nil( ) → t1
                    &  construct at RS cons(x, t1) → t2
                    &  call append [RS] (ar, t2) → (c)
                    &  killregion TS
                    &  exit (c).
```

Figure 4.2: A RP version of the naïve reverse function. Except for the region annotations, this is the same code as in Figure 3.2.

A program that does not go wrong_1 or wrong_4 should behave the same under the RP semantics as it would under the plain P semantics if the region annotations were removed. (That could be proved formally by defining a appropriate consistency relation between states of the two semantics and showing that consistency is preserved by each computation step).

The only differences between this semantics and the P semantics are

- An address is now pair of a **region number** (not to be confused with a region register which is a syntactic entity) and an **offset** within the region. This representation originates from Tofte and Talpin [1993] except that they call region numbers "region names". We feel that the word "name" for a dynamic concept could be misleading; region numbers exist at run time.

  In the intended implementation of RP, an address is a single, conventional pointer. The separation of it into region number and offset is simply a formal tool to make it easy to express in the semantics what happens when a region is deallocated.
- Computation states include a **region environment** which maps region registers to region numbers, and a $\mathsf{killregion}$ **set** consisting of those regions that have been made locally and should be killed before the current predicate returns.
- The definitions of $\mathsf{builtin}$, $\mathsf{construct}$, and $\mathsf{makevar}$ use the region environment to select the address of the newly allocated store cell.
- The definition of the call instruction and the exit pseudoinstruction have been extended with machinery to handle region parameters in a way completely similar to normal input parameters.

Figure 4.3 defines the domains used in the semantics.

4.4.1 The initial state

The initial state is

$$s_0 = [(\varnothing, \varnothing, \varnothing, \varnothing, \mathsf{call}\ \mathfrak{pr}_0() \to () \ \&\ \mathsf{fail}, [])]$$

4.4.2 The transition relation

The successor of $[]$ is ok. Any other non-final state has the shape

$$(\sigma, \mathcal{R}, \mathcal{K}, \mathcal{E}, \mathfrak{b}, \delta, \kappa) :: \mathfrak{fs}$$

and we now proceed by case analysis on $\mathfrak{b}$.

The successor state is implicitly wrong_1 if the computation of it would involve looking up a register name in a environment that does not define it, or extending a environment with a binding for a register name that it already defines. The same convention applies to region environments.

$$
\begin{array}{lll}
r & \in \langle \mathit{RegNum} \rangle & = \mathbb{N} \\
o & \in \langle \mathit{Offset} \rangle & = \mathbb{N} \\
\alpha & \in \langle \mathit{Addr} \rangle & = \langle \mathit{RegNum} \rangle \times \langle \mathit{Offset} \rangle \\
 & \in \langle \mathit{Datum} \rangle & = \biguplus_f \langle \mathit{Addr} \rangle^{|f|} \uplus \langle \mathit{Addr} \rangle \uplus \{\mathsf{uninst}\} \\
\sigma & \in \langle \mathit{Store} \rangle & = \langle \mathit{RegNum} \rangle \xrightarrow{\text{fin}} \langle \mathit{Offset} \rangle \xrightarrow{\text{fin}} \langle \mathit{Datum} \rangle \\
 & & \simeq \langle \mathit{Addr} \rangle \xrightarrow{\text{fin}} \langle \mathit{Datum} \rangle \\
x & \in \langle \mathit{Register} \rangle & \\
\mathcal{E} & \in \langle \mathit{Env} \rangle & = \langle \mathit{Register} \rangle \xrightarrow{\text{fin}} \langle \mathit{Addr} \rangle \\
\rho & \in \langle \mathit{Region} \rangle & \\
\mathcal{R} & \in \langle \mathit{RegEnv} \rangle & = \langle \mathit{Region} \rangle \xrightarrow{\text{fin}} \langle \mathit{RegNum} \rangle \\
\mathcal{K} & \in \langle \mathit{KillSet} \rangle & = \mathcal{P}(\langle \mathit{Region} \rangle) \\
b & \in \langle \mathit{body} \rangle & \\
c & \in \langle \mathit{clause} \rangle & \\
C & \in \langle \mathit{clauses} \rangle & \\
\kappa & \in \langle \mathit{Cont} \rangle & = \left(\begin{array}{l} \langle \mathit{RegEnv} \rangle \times \langle \mathit{KillSet} \rangle \times \langle \mathit{ValEnv} \rangle \\ \quad \times \langle \mathit{Register} \rangle^* \times \langle \mathit{body} \rangle \times \langle \mathit{Dump} \rangle \end{array} \right)^* \\
fr & \in \langle \mathit{Frame} \rangle & = \langle \mathit{Store} \rangle \times \langle \mathit{RegEnv} \rangle \times \langle \mathit{KillSet} \rangle \times \langle \mathit{ValEnv} \rangle \\
 & & \quad \times \langle \mathit{body} \rangle \times \langle \mathit{Dump} \rangle \times \langle \mathit{Cont} \rangle \\
fs & \in \langle \mathit{Frames} \rangle & = \langle \mathit{Frame} \rangle^* \\
\delta & \in \langle \mathit{Dump} \rangle & = \langle \mathit{Frames} \rangle \\
s & \in \langle \mathit{State} \rangle & = \langle \mathit{Frames} \rangle \uplus \{\mathsf{ok}, \mathsf{wrong}_{1,2,3,4}\}
\end{array}
$$

Figure 4.3: The objects used in the RP semantics. By slight abuse of notation we sometimes manipulate a $\langle Store \rangle$ as if it were a curried function and sometimes as if it were a function from $\langle Addr \rangle$ to $\langle Datum \rangle$. The choice of symbols should make clear at each place what is meant.

$b =$ call $pr[\rho_1,\ldots,\rho_{\lfloor pr \rfloor}](x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr \rfloor})$ & b'

- Let
$$pr[\rho'_1,\ldots,\rho'_{\lfloor pr \rfloor}] \quad :- \quad \begin{array}{l} \mathsf{enter} \rightarrow (x_{11},\ldots,x_{1\lceil pr \rceil})\ \&\ b_1; \\ \ldots; \\ \mathsf{enter} \rightarrow (x_{k1},\ldots,x_{k\lceil pr \rceil})\ \&\ b_k. \end{array}$$
be the definition of pr.
- $\mathcal{R}' \leftarrow \{\rho'_i \mapsto \mathcal{R}(\rho_i) \mid 1 \le i \le \lfloor pr \rfloor\}$
- $\mathcal{E}_i \leftarrow \{x_{ij} \mapsto \mathcal{E}(x_j) \mid 1 \le j \le \lceil pr \rceil\}$ for $1 \le i \le k$
- $\delta' \leftarrow fs$
- $\kappa' \leftarrow (\mathcal{R}, \mathcal{K}, \mathcal{E}, (x'_1, \ldots, x'_{\lfloor pr \rfloor}), b', \delta) :: \kappa$
- $fr_i \leftarrow (\sigma, \mathcal{R}', \varnothing, \mathcal{E}_i, b_i, \delta', \kappa')$ for $1 \le i \le k$
- The next state is $fr_1 :: fr_2 :: \ldots :: fr_k :: fs$

$b =$ builtin $pr(x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1$ at $\rho_1,\ldots,x'_{\lfloor pr \rfloor}$ at $\rho_{\lfloor pr \rfloor})$ & b'

The implementation is supposed to provide a function

$$\phi_{pr} : \mathbb{Z}^{\lceil pr \rceil} \rightarrow \left(\mathbb{Z}^{\lfloor pr \rfloor} \uplus \{\mathsf{fail}, \bot\}\right)$$

- If any $\mathcal{E}(x_i) \notin \mathrm{Dom}\,\sigma$ then wrong_4; else
- If any $\sigma(\mathcal{E}(x_i)) \notin \mathbb{Z}$ then wrong_2; else
- Compute $\phi_{pr}(\sigma(\mathcal{E}(x_1)), \ldots, \sigma(\mathcal{E}(x_{\lceil pr \rceil})))$.
  If the result is fail, then the next state is $\mathfrak{fs}$.
  If the result is $\perp$, then the current state is its own successor.
  Otherwise, let the result be $(n_1, \ldots, n_{\lfloor pr \rfloor})$.
- $r_i \leftarrow \mathcal{R}(\rho_i)$
- If any $r_i \notin \mathrm{Dom}\,\sigma$ then wrong_4; else
- $o_i \leftarrow$ fresh $\notin \mathrm{Dom}\,\sigma(r_i)$ (for $1 \le i \le \lfloor pr \rfloor$; $o_i \neq o_j$)
- $\sigma' \leftarrow \sigma\{(r_i, o_i) \mapsto n_i \mid 1 \le i \le \lfloor pr \rfloor\}$
- $\mathcal{E}' \leftarrow \mathcal{E}\{x'_i \mapsto (r_i, o_i) \mid 1 \le i \le \lfloor pr \rfloor\}$
- The next state is $(\sigma', \mathcal{R}, \mathcal{K}, \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ construct at ρ $f(x_1,\ldots,x_{|f|}) \to x'$ & b'

- $r \leftarrow \mathcal{R}(\rho)$
- If $r \notin \mathrm{Dom}\,\sigma$ then wrong_4; else
- $o \leftarrow$ fresh $\notin \mathrm{Dom}\,\sigma(r)$
- $\alpha \leftarrow (r, o)$
- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \alpha\}$
- $\sigma' \leftarrow \sigma\{\alpha \mapsto f(\mathcal{E}(x_1), \ldots, \mathcal{E}(x_{|f|}))\}$
- The next state is $(\sigma', \mathcal{R}, \mathcal{K}, \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ destruct x as $f \to (x'_1,\ldots,x'_{|f|})$ & b'

- If $\mathcal{E}(x) \notin \mathrm{Dom}\,\sigma$ then wrong_4; else
- If $\sigma(\mathcal{E}(x)) \in \langle \mathit{Addr} \rangle \cup \{\mathsf{uninst}\}$ then wrong_3; else
- If not $\sigma(\mathcal{E}(x)) = f(\alpha_1, \ldots, \alpha_{|f|})$ for some α_is, then the next state is $\mathfrak{fs}$; else
- $\mathcal{E}' \leftarrow \mathcal{E}\{x'_i \mapsto \alpha_i \mid 1 \le i \le |f|\}$
- The next state is $(\sigma, \mathcal{R}, \mathcal{K}, \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ makevar at $\rho \to x'$ & b'

- $r \leftarrow \mathcal{R}(\rho)$
- If $r \notin \mathrm{Dom}\,\sigma$ then wrong_4; else
- $o \leftarrow$ fresh $\notin \mathrm{Dom}\,\sigma(r)$
- $\alpha \leftarrow (r, o)$
- $\sigma' \leftarrow \sigma\{\alpha \mapsto \mathsf{uninst}\}$
- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \alpha\}$
- The next state is $(\sigma', \mathcal{R}, \mathcal{K}, \mathcal{E}', b', \delta, \kappa) :: \mathfrak{fs}$

$b =$ deref $x \to x'$ & b'

- If $\mathsf{F}_\sigma(\mathcal{E}(x)) = \perp$ then wrong_4; else
- $\alpha \leftarrow \mathsf{F}_\sigma(\mathcal{E}(x))$
- If $\sigma(\alpha) = \mathsf{uninst}$ then the next state is $\mathfrak{fs}$; else
- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \alpha\}$

- The next state is $(\sigma, \mathcal{R}, \mathcal{K}, \mathcal{E}', b', \delta, \kappa) :: fs$

$b =$ alias $x \rightarrow x'$ & b'

- $\mathcal{E}' \leftarrow \mathcal{E}\{x' \mapsto \mathcal{E}(x)\}$
- The next state is $(\sigma, \mathcal{R}, \mathcal{K}, \mathcal{E}', b', \delta, \kappa) :: fs$

$b =$ unify x_1=x_2 & b'

- Try to unify $\mathcal{E}(x_1)$ and $\mathcal{E}(x_2)$ in store σ according to Definitions 3.9ff (page 42).
  If the result is wrong_4 then that is the next state; if the result is fail then the next state is fs; else the result is a new store σ'.
- The next state is $(\sigma', \mathcal{R}, \mathcal{K}, \mathcal{E}, b', \delta, \kappa) :: fs$

$b =$ cut & b'

- The next state is $(\sigma, \mathcal{R}, \mathcal{K}, \mathcal{E}, b', \delta, \kappa) :: \delta$

$b =$ makeregion $\rightarrow \rho$ & b'

- $r \leftarrow$ fresh $\notin \text{Dom}\,\sigma$
- $\mathcal{R}' \leftarrow \mathcal{R}\{\rho \mapsto r\}$
- $\sigma' \leftarrow \sigma\{r \mapsto \varnothing\}$
- $\mathcal{K}' \leftarrow \mathcal{K} \cup \{\rho\}$
- The next state is $(\sigma', \mathcal{R}', \mathcal{K}', \mathcal{E}, b', \delta, \kappa) :: fs$

$b =$ killregion ρ & b'

- If $\rho \notin \mathcal{K}$ then wrong_1; else
- $\sigma' \leftarrow \sigma\downarrow_{\text{Dom}\,\sigma \setminus \{\mathcal{R}(\rho)\}}$
- $\mathcal{R}' \leftarrow \mathcal{R}\downarrow_{\text{Dom}\,\mathcal{R} \setminus \{\rho\}}$
- $\mathcal{K}' \leftarrow \mathcal{K} \setminus \{\rho\}$
- The next state is $(\sigma', \mathcal{R}', \mathcal{K}', \mathcal{E}, b', \delta, \kappa) :: fs$

$b =$ fail

- The next state is fs

$b =$ exit $(x_1,\ldots,x_n)$

- If $\mathcal{K} \neq \varnothing$ then wrong_1; else
- $(\mathcal{R}', \mathcal{K}', \mathcal{E}', (x_1', \ldots, x_n'), b', \delta') :: \kappa' \leftarrow \kappa$
- $\mathcal{E}'' \leftarrow \mathcal{E}'\{x_i' \mapsto \mathcal{E}(x_i) \mid 1 \leq i \leq n\}$
- The next state is $(\sigma, \mathcal{R}', \mathcal{K}', \mathcal{E}'', b', \delta', \kappa') :: fs$

Chapter 5

A run-time design for RP

In this chapter we describe how to implement the region-based memory-management primitives of RP at run time.

We do not address those facets of an RP implementation that are not related to the region-based memory management. Examples of such facets are how to manage the program's control flow in the presence of backtracking, and how to find space to spill the values of registers while calling a predicate. These issues can be handled with well-known techniques[1], and knowledge of these techniques is *not* necessary for understanding this chapter.

The governing idea behind the exposition is that of the region model as an abstract data type: the RP code treats a "region" as an opaque concept, manipulated through a small set of primitives. It does not depend on any particular implementation of the region model. On the other hand, the set of primitives of course influences which implementations are possible.

We call the program module that implements the primitives the **memory manager**. The RP code (which may be interpreted or compiled into a lower-level language) will be called the **client program**. The memory that is managed by the memory manager is the **heap**.

We start by a simple set of primitives that allow a simple implementation, and add new primitives and change the implementation until we have described a complete implementation of the region-based memory management for RP.

5.1 A simple region model

The first model we describe is a simple one, similar to the one used in the ML Kit. It has only three primitives:

[1]The standard reference model here is the WAM [Aït-Kaci 1991]. Almost everything about executing RP programs that this chapter does not explain can be implemented by direct analogy with the WAM's "local stack". The exception is output parameters, but the straightforward convention of returning them in the argument registers should work well.

Quite a number of extensions and variants of the original WAM model for the local stack exist and are described in the literature. Most of these can be used in a region-based implementation as well. Indeed, our prototype RP implementation does not follow the WAM design closely.

makeregion: *(nothing)* $\rightarrow$ ρ: REGION
Create a new region ρ.

alloc: ρ: REGION, n: integer $\rightarrow$ α: pointer
Allocate n machine words of memory "in" the region ρ. The returned pointer points to the first of n consecutive words of memory which are guaranteed not to be used for any other purpose until a *killregion* operation on ρ is performed.

killregion: ρ: REGION $\rightarrow$ *(nothing)*
Destroy the region ρ. Any memory that has been *alloc*ated in ρ becomes available for allocation in other regions. It is an error to used ρ in any primitives after *killregion* has been called.

Each of the primitives will terminate the program with a "out of memory" message if it would need more memory than is avaliable to perform its specified function.[2] A good implementation should of course make sure that this only happens when the total number of regions and allocated words gets large.

5.1.1 Implementation

The implementation problem in the simple region model is that an unlimited number of regions can exist at the same time, and each of those may grow to an unlimited size.

This means that we cannot hope for being able to let the addresses of all the words allocated in a particular region be contiguous. To see that: Suppose we have a program that allocates a single word in each of ρ_1, through ρ_k and afterwards allocates a lot of words in one of those regions. When the first k words are allocated the memory manager does not yet know which of the k regions is going to grow big. It is not difficult to see that no matter where it places the initial k allocations, there is a way for the client program (the "adversary") to choose a region that cannot expand to more than a $(k-1)$th of the heap before it collides with one of the already-allocated words. That is clearly unacceptable.

A bad solution

One solution would be to build the region-based memory manager on top of a traditional memory manager with an explicit allocation and deallocation operations. When the client program requests n words we could allocate $n+1$ words from the underlying memory manager, using the first to keep a linked list of all blocks belonging to each region. This design would work, but is burdened by the (memory and running-time) overhead of the traditional memory manager. Also, the *killregion* primitive might take long time to execute, making it hard to reason about the running time of the program.

[2] This could even happen for the *killregion* operation, if there is not enough memory to store the administrative data that make sure that the kill can be undone by backtracking.

A better solution

The solution we do use is from the ML Kit [Tofte et al. 1997]. Each region consists of a linked list of fixed-size **pages**. In each page a single word is used to point to the next region in the list, and the entire rest of the page is available for the client program's allocations.

We implement the abstract type REGION as a pointer to the oldest of the region's pages. In this page the *makeregion* operation allocates a record of management data at the beginning of the payload area (thus it is not the *entire* remainder of that page that is available for the client program's data). This management record must contain

- a pointer to the newest page: the **newest-page pointer**.
- a count of how much space is left for allocation in the page: the **free-count**.

We see that the *makeregion* primitive has to allocate the first page for the region and initialise the management record properly.

The *alloc* primitive allocates memory only in the region's newest page. If there is not enough room for the allocation in the currently newest page, it adds a fresh page at the "new" end of the region and tries again[3]. The remaining words in the previously newest page are wasted as "slack", as least until the region is eventually killed.

Where do *makeregion* and *alloc* get fresh pages? We maintain a linked list of unused pages, the **free-list**; when a new page is needed it can usually be removed from the front of that. If the free-list is empty the primitive must request a batch of, say, 100 fresh pages from the operating system and turns those it does not use into a new free-list. (It would also work to get only one fresh page from the operating system. That would make the individual primitive that found the free-list empty faster, but it would mean the the program needed to make 100 times as many operating system calls, and those are usually a lot slower than simply popping the free-list).

The *killregion* simply concatenates the entire region onto the free-list. That is fast, because the pages in the region have already been linked together, and the management record allows us to locate both ends of the region in constant time.

5.1.2 Time efficiency

Our solution has the desirable property that all of the three primitives can be implemented as constant-time operations. At worst, the *alloc* primitive may have to

[3]The client program has better not try to allocate more memory at once than what fits in a single page. In RP as we have described it, it is easy to determine statically how big the largest possible allocation is and adjust the page size accordingly before the program starts. With richer languages where that is not the case—such as if strings were to be implemented more efficiently than as linked lists of characters—special workarounds and primitives are needed for the potentially big data types. There is a solution in the ML Kit but we do not go into detail here.

- determine that there is not enough room in the currently newest page and that the free-list is empty,
- obtain a constant number of pages from the operating system and link them together to form a new free-list,
- unlink the first page in the free-list,
- link it into the region, and
- update the region's management record

all of which take constant time. *killregion* is obviously also a constant-time operation.

This property has been the source of great enthusiasm about the region model, because it makes it possible to establish upper bounds on the time it takes from the client program to get from point A to point B. This is in contrast to memory management by garbage collector, where it is virtually impossible to predict when the garbage collector suddenly needs to mark and compact several megabytes worth of heap space before it can complete an allocation. Such matters are important for real-time or interactive programs. (Of course, to get real running-time guarantees one needs to have a real-time operating system which can deliver fresh pages in a known bounded time).

We shall see that we are not able to keep to the "all operations take constant time" slogan as we expand the model to cover all of RP, but it will still, in general, be possible to determine statically *which* primitives in the client program risk being slow.

5.1.3 Space efficiency

If we assume that regions grow big and that the page size is large compared to the size of individual allocations, our scheme has an excellent ratio of words used for memory management per word used for the client program's payload data. On average, only a few words per page full of payload data will be used for the link to the next page or wasted as slack at the end of a page.

However, if a region does *not* grow big—*i.e.*, if only a couple of words is ever allocated in it—the figure is not as favorable because an entire page will be used to hold that couple of words. This suggests that pages should not be too big, and definitely should be smaller than the "pages" of several hundred words each that are used for implementing virtual memory. In the experiments we report on in Chapter 11 we use a total page size of 16 words. This might seem excessively small, but remember that the administrative overhead per page is only one word.

The ML Kit contains a **multiplicity analysis** which aims at statically identifying regions that provably do not grow bigger than a small number of words. Those regions are handled differently at run time. We have not considered how to implement this idea efficiently for Prolog, but it would be natural to do so as a continuation of the work in this thesis.

5.2 Backtracking

We now extend our region model to support backtracking.

The important point here is that backtracking should act as a time machine: When we backtrack to an earlier point in the execution history, all the changes that have been made to the program's memory since that time should be undone. Regions that have been created should be forgot. Allocations that have been made should be undone. Regions that have been killed should magically reappear in the same place and with the same contents as they had at the point we backtrack to.

Obviously this is impossible without some cooperation from the client program. In this section we shall require three different kinds of cooperation. The first is that the client program tells us when it reaches a point it might later want to backtrack to:

pushchoice: *(nothing)* $\rightarrow$ *(nothing)*

> This primitive conceptually pushes the entire state (*i.e.*, the size, location and contents of each existing region) of the memory manager onto an (implicit and global) **choice point stack**.

The description of *pushchoice* also reveals the second kind of cooperation, namely that the client program must obey a stack discipline for backtracking. It can only ever backtrack to the state that was saved by the most recent *pushchoice*:

popchoice: *(nothing)* $\rightarrow$ *(nothing)*

> Replaces the memory manager's state with the topmost state on the choice point stack, and removes that state from the stack.

This second form of cooperation is less reasonable than the first in that it does not allow the `cut` instruction to be implemented. We'll describe in Section 5.3 how to relax the restriction enough to allow `cut`.

The third form of cooperation is that the client program must not change any data in the heap. It can write some data into the newly-allocated memory block immediately after an *alloc* operation, but must not change them afterwards. This restriction prevents logical variables from being implemented yet, but we'll add special support for logical variables in Section 5.4.

5.2.1 Implementation

Of course it is impractical to make *pushchoice* actually make a copy of the entire heap. Our stategy will be to maintain enough information about what has happened since the *pushchoice* to be able to undo that quickly when *popchoice* is called. That information in a data structure we created by *pushchoice* and called a **choice point**.

Snapshots

To begin with, we assume that the only primitive that is used between *pushchoice* and *popchoice* is *alloc*. Then the situation at the *popchoice* is that some regions may have grown since the *pushchoice*; we must then **shrink** these regions accordingly.

This means that we need to remember which size the regions had at the time of *pushchoice*. A choice point must contain a (pointer to a) list of little structures that we call **snapshots**. Each snapshot contains enough information to restore one region to the state it had when it was created. For now[4], that means that a snapshot contains a pointer to the region and a copy (a "snapshot") of the entire management record at the time the snapshot was created.

For now, we deliberately ignore the question of how to manage the memory that is used to store the snapshots (and choice points) themselves. We'll get to that in due time, but not until we've examined exactly how and when they are used.

When are snapshots created? A naïve answer would be that *pushchoice* should create snapshots for every existing region. This idea can be dismissed out of hand: it is hopelessly inefficient when many regions exist and few or none of them actually grow before the *popchoice* is executed (the vast majority of choice points are very short-lived in practise).

The solution is to create shapshots on an as-needed basis. *pushchoice* creates none, but each time an *alloc* is executed it checks whether the newest choice point has a snapshot for the region and creates one if it hasn't. The check can be made fast if we extend the region's management record with a **owner-of-newest-snapshot pointer** which points to the newest choice point that has a snapshot for the region. That way *pushchoice* and *alloc* both stay constant-time operations, and unnecessary snapshots do not get created.

How to shrink a region

We're still assuming that the only thing that happens between *pushchoice* and *popchoice* is *alloc*. Then *popchoice*'s job consists of popping a choice point and traversing its snapshot list while shrinking each region accordingly. But how does one actually shrink a region?

We know that a region is a linked list of pages, but we have not yet decided which way the links go. We now decide that *old pages contain pointers to newer pages*. That means that it is easy to shrink a region given the snapshot's copy of the newest-page pointer:

1. Update the region's newest page's successor field to point to the first page in the free-list.
2. Update the management record with the saved data from the snapshot (including the saved newest-page pointer).

[4]Eventually we'll add fields to the management record that do not need to be saved in snapshots

3. Let the new "head of the free-list" be the successor of the page that is the region's newest page according to the new newest-page pointer.

   (If the old and new newest-page pointers point to the same page, its successor will be the current value of the free-list pointer that was stored in step 1, and the entire operation has been a no-op with respect to the free-list. Otherwise the net effect is that the pages that were added to the region after the snapshot was taken have now been pushed onto the front of the free list).

4. Let the successor of the newest page be NULL, terminating the linked-list structure of the region.

This is enough to shrink the region. The snapshot is not needed anymore and can be forgotten. But that also means that the choice point we're popping is not anymore the owner of the newest snapshot for the region, so we also need to update the owner-of-newest-snapshot field in the management record. That again means that the snapshot should contain a copy of the owner-of-newest-snapshot field just before it was created itself.

To summarise: A region management record now contains

- a newest-page pointer,
- a free-count, and
- an owner-of-newest-snapshot pointer,

and a snapshot contains copies of all three in addition to a pointer to the region itself.

The termination list

Now let's consider what should happen if a *makeregion* primitive is executed between *pushchoice* and *popchoice*.

In this case, the *popchoice* will be backtracking to a point in time where the region did not exist at all. This means that *popchoice* must deallocate the region exactly as if the *killregion* implementation from Section 5.1.1. So *popchoice* must be able to find the region. We extend the choice point with a list of regions that should be deallocated when it is popped off the choice-point stack. We call this list the **termination list**.

Intuitively, the fact that a region is mentioned in a termination list serves the same purpose as a snapshot: it tells something about the region's state at the time the choice point was created. Only in the case of a termination list entry the saved state is not "so-and-so big" but "not there at all".

The implementation of the *makeregion* primitive must be extended with preparations for this:

1. Obtain a fresh page and make it into a one-page region with its own management structure allocated at the beginning of the payload area of the page. This is what *makeregion* did before we began to consider backtracking.

2. Add the new region to the termination list of the topmost choice point.
3. Let the owner-of-newest-snapshot pointer point to the topmost choice point[5]. This will actually be a lie, because there *are* no snapshots for the regions yet. It works out fine, however: A snapshot will only be created *if* another choice point is pushed *and* something is subsequently allocated in the region—which is precisely when a snapshot will be needed.

killregion and backtracking

Now we come to the difficult question: What happens when *killregion* occurs between *pushchoice* and *popchoice*?

There are two different cases to consider. The first is

pushchoice ... *makeregion* ... *killregion* ... *popchoice*

which is *not* difficult. Backtracking does not interfere with the region's life at all, and *killregion* can do just what it did in Section 5.1.1. The only problem is to know that this easy case applies; we'll get to that in a moment.

The difficult case is

makeregion ... *pushchoice* ... (*alloc*?) ... *killregion* ... *popchoice*.

Here the intended semantics of the primitives say that *killregion* should make the region disappear, and *popchoice* should make it reappear in the same place.

The only practical way to implement this is of course to make sure that the region is never deallocated at all. A first approximation to an implementation would be to have *killregion* do nothing at all in this case. Then it would be right there for *popchoice* to restore to its saved condition.

A unsatisfactory side of this approach shows if we imagine that *alloc* operations added a lot of pages to the region between *pushchoice* and *killregion*. The only thing *popchoice* does to these pages is add them to the free list, and by calling *killregion* the client promises that *it* is not going to need the data again. The pages just sit there being unused. It would be better to add them to the free-list as soon as the *killregion* call happens.

So that we do: When *killregion* can't entirely deallocate the region, it instead shrinks it to the size it would assume anyway at the next *popchoice*. This means that *killregion* has to be able to find the snapshot telling how many pages to free. Searching through the snapshot list for the appropriate choice point might be slow, so we add a **newest-snapshot pointer** to the region management record. (We also add a copy of it to the layout of snapshots).

The newest-snapshot pointer also provides a solution to the problem of finding out whether *killregion* may deallocate the region or not. We specify

[5]We assume there is always a topmost choice point, at least a dummy choice point that the memory manager created as part of its start-up code. The client program will never pop that, because it does not know it exists.

that *makeregion* should initialise the newest-snapshot pointer to NULL, correctly recording that there is no newest snapshot. The algorithm for *killregion* then becomes

1. If the owner-of-newest-snapshot pointer does not point to the currently topmost snapshot, then create a new snapshot for the region. (This is the same operation as *alloc* does to make the snapshot up-to-date).

2A. If the newest-snapshot pointer is NULL (even after step 1 has possibly created a snapshot and pointed the pointer at it), then deallocate the region entirely, contributing its pages to the free-list.

    The region should also be removed from the termination list that references it lest it pages get added to the free-list twice with confusion to follow.

2B. On the other hand, if there *is* a newest snapshot, then shrink (in the same way *popchoice* does) the region according to the saved data in the snapshot. The snapshot can then be removed from the choice point list.

Summary of the data structures used so far

A region management record now contains

- a newest-page pointer,
- a free-count,
- an owner-of-newest-snapshot pointer (initialised by *makeregion* to point to the topmost choice point), and
- a newest-snapshot pointer (initialised by *makeregion* to be NULL),

and a snapshot contains copies of all four in addition to a pointer to the region itself.

5.2.2 Time efficiency

We have not yet decided how the snapshot and termination list belonging to a choice point are to be held together. However, it is evident that we can find a representation which makes all the list operations we yet need take constant time.

Under this assumption it is easy to see that *makeregion*, *killregion*, *alloc*, and *pushchoice* can still be implemented to run in constant time.

The running time of *popchoice* is not so nice, because it has to traverse the snapshot and termination lists of the choice point and to a (constant-time) operation on each of their elements. Each of these lists can in principle be aribtrarily long.

However, each of the snapshots and regions-to-be-terminated that are handled by *popchoice* has been *created* somewhen since the choice point was created. If we charge the time used to "finalise" each of them on their

creation, we can conclude that each primitive operation still takes constant time—*when amortised* over the lifetime of each choice point.

This is still better than garbage collection—because a garbage collector might need to trace each pice of memory arbitarily many times, the cost of tracing cannot be amortized—but is not completely satisfactory, as it breaks the property that regions make it easy to reason about how long time it might take for a program to get from any point A to any point B. It now holds only when the execution path that contains point A has not yet failed when point B is reached.

We argue, however, that these situations still account for most of the cases where one would *want* to reason about running times. The time to get from A to B is really interesting only when A and B represent interaction with the program's environment. And Prolog programs rarely interact with their environment in execution paths that might fail and backtrack; consideration of program structure and readability usually mandate that just as severely as does memory management costs,

For the exceptions to this general rule, such as a top-level `repeat–fail` loop, it is often a simple task to derive by hand a bound on the number of regions that are affected by a backtracking operation. It would be an interesting extension of the work in this thesis to try to automate such an analysis.

5.2.3 Space efficiency

In the worst case, the backtracking machinery can use a lot of memory. It is not difficult to produce a pathological RP program such that *each*[6] heap allocation triggers the creation of a new snapshot. With the optimized representation we use in our prototype system a snapshot takes up 4 words of memory. This means that when regions do "grow big" in the sense of Section 5.1.3, the region-based memory manager may result in up to five times as much memory being used as the client program needs.

Our only answer to that scenario is that is does not seem to happen in practise. Our prototype implementation counts the memory used to hold snapshots, and in none of our benchmark programs do very many snapshots need to exist simultaneously.

Our benchmarks do not attempt to span the entire range of Prolog program behaviour. However, we are confident that our observation covers most of the programs that are met in practise. Our argument is that the number of choice points that exist at any given time is usually bounded, either due to "indexing" and other tricks on behalf of the compiler, or due to explicit `cuts` (which we add to the system in a little while). Regardless of memory-management paradigm it is ill-advised to write Prolog code that leaves "dead" choice points (*i.e.*, choice points that correspond to alternative execution paths that will fail immediately), so if there are many active choice points even after these factors have been considered it usually means that a

[6] Or at least, an arbitrarily large fraction of all

brute-force search through an exponentially large search space is going on. Prolog programmers do know to shy away from that.

5.3 Cuts

The "cut" is one of the most difficult constructs to master for a novice Prolog programmer. We shall now see that it also introduces complications of its own in region-based memory management.

Fortunately, our abstraction of the interface between memory manager and client program at least makes it easy to specify what a cut *does*: It pops some number of choice points off the top of the choice point stack *without* restoring the saved memory-manager states they represent.

In our model we can phrase that by splitting the task formerly done by *pushchoice* and *popchoice* into two separate primitives each:

mark: *(nothing)* $\rightarrow$ δ: MARK

Pushes a **mark** onto the choice point stack and returns a magic token δ which identifies the mark uniquely.

pushchoice: *(nothing)* $\rightarrow$ *(nothing)*

Unchanged.

backtrack: *(nothing)* $\rightarrow$ *(nothing)*

Replaces the memory manager's state with the topmost state (ignoring marks from *mark*) on the choice-point stack. The choice point is left on the choice point stacks; that is, if two *backtrack* operations directly follow each other, the latter of them has no (observable) effect.

cut: δ: MARK $\rightarrow$ *(nothing)*

Pops saved states and/or marks off the top of the choice-point stack until the mark δ is at the top of the stack.

Any mark that get popped off the stack by a *cut* becomes invalid and may not be used as argument to another *cut*. The δ mark itself stays valid, though.

The "main" state of the heap is not changed.

What was previously

pushchoice ... *popchoice*

must now be replaced by

mark, *pushchoice* ... *backtrack*, *cut*.

This sequence leaves a mark on the choice-point stack. Fortunately, as we shall see, marks do not actually take up space.

The marks on the choice point stack represent the $\langle Dump \rangle$ component in the states of the RP semantics in Section 4.4.

5.3.1 Implementation

A MARK is simply a copy of the top-of-choice-point-stack pointer at the time of the *mark* operation. Nothing is actually pushed onto the stack. (The metaphor of marks as items on the stack merely served to define when δ values become obsolete).

The *backtrack* operation is easy. It simply does what we found out in Section 5.2 that *popchoice* should do—except that it does not remove the choice point from the choice point stack. It does, however, empty its termination and snapshot lists.

The *cut* operation is more intricate. We only describe how to pop a single choice point off the stack; it is less complicated to generalise this to simultaneous pops of more than one choice point than it would be to describe the generalisation in detail. Throughout the discussion we call the choice point that is to be cut away C_0 and the one immediately beneath it (the one that becomes the new topmost choice point) C_1.

Superficially it is not a big problem to just pop a choice point off the choice point stack. The challenge is, however, to keep the snapshot structure (and the associated summary information in the region management records) consistent. Our guiding principle will be that *cut* should leave the memory management structure in the state they would have had if the corresponding *pushchoice* had never been executed.

We now proceed by considering the events which may have taken place between *pushchoice* and *cut*.

cut and *makeregion*

$$\mathit{pushchoice}_{C_1} \ldots \mathit{pushchoice}_{C_0} \ldots \mathit{makeregion} \ldots (\mathit{alloc?}) \ldots \mathit{cut}_{C_0}$$

Here, *makeregion* has inserted the new region in C_0's termination list and set its owner-of-newest-snapshot pointer to point to C_0. We know that this is still the value of the owner-of-newest-snapshot pointer: The region has no proper snapshots because snapshots are only created if the owner-of-newest-snapsnot pointer does *not* point to the topmost choice point (which has been C_0 in the region's entire lifetime).

Also, this is only way a region can end up in C_0's termination list, so the termination list allows to easily find the regions where this case applies.

If C_0 had not existed, the regions would have been in C_1's termination list instead, and their owner-of-newest-snapshot pointers would point to C_1. Thus what *cut* needs to do is to adjust the the owner-of-newest-snapshot pointers for each region in the termination list and append the entire list to C_1's termination list.

cut and *alloc*

$$\mathit{pushchoice}_{C_1} \ldots \mathit{pushchoice}_{C_0} \ldots \mathit{alloc} \ldots \mathit{cut}_{C_0}$$

Now consider an allocation in a region that already existed when C_0 was

created. The allocation has left a snapshot in C_0's snapshot list; because C_0 is the topmost choice point any snapshots found in its snapshot list will be the newest snapshot of their respective regions.

What would have happened if C_0 did not exist? Perhaps the allocation would have created a snapshot for C_1, perhaps it would not. The decision would have been governed by the value of the region's owner-of-newest-snapshot pointer at the time when C_0 was created. At the time of the *cut*, we can know which decision would have been taken, because exactly that value is stored in the C_0 snapshot.

This analysis implies that the *cut* operation should inspect each of the snapshots in C_0's snapshot list. If the owner-of-newest-snapshot pointer in a snapshot points to C_1, the snapshot becomes **obsoleted**, meaning that we should adjust things to look like it had never been created. Specifically, this means that the newest-snapshot and owner-of-newest-snapshot pointers in the region management record should be restored from the saved values in the obsoleted snapshot (the other half of the snapshot, containing copies of the newest-page pointer and the free-count should *not* be restored as we're not undoing the allocations).

Those snapshots that are not obsoleted would have been created in any case, only owned by C_1 instead of C_0. The *cut* operation should *reassign* them to C_1. This involves inserting the snapshot in C_1's snapshot list and adjusting the owner-of-newest-snapshot pointer in the region management record to point to C_1 instead of C_0.

cut and *killregion*

$$\mathit{pushchoice}_{C_1}\ ..(A)..\ \mathit{pushchoice}_{C_0}\ ..(B)..\ \ ...\ \mathit{killregion}\ ...\ \mathit{cut}_{C_0}$$

Once again, *killregion* is the most complex case. According to the analysis in Section 5.2.1, the *killregion* operation has undone any allocations in the region that happened during "..(B)..". If C_0 had never been created, *killregion* would also have undone the allocations that happened during "..(A)..". If "..(A).." contained the creation of the region, it would have been entirely deallocated.

It now becomes the job of *cut* to complete the *killregion* operation. The first problem is to know which regions are affected at all. Remember that Section 5.2.1's implementation of *killregion* made sure that C_0 had a snapshot for the region, then immediately removed that snapshot after shrinking the region. Then there is no way for *cut* to know that there is a "kill in progress".

Our solution is that *killregion* should *not* remove the snapshot after shrinking the region. The snapshot should be left in C_0's snapshot list, and a flag in the region management record should be set to indicate that the region has been killed.

This way *cut* will come across the snapshot while it scans the snapshot list. If the owner of the previous snapshot was not C_1, the snapshot should simply be reassigned to C_1 whether or not the region has been killed. However, when a snapshot is obsoleted (because it has a separate snapshot for

C_1) the killed-flag for the region is inspected. If it is set, an "artificial" *killregion* operation is executed, which restores the region to the state it had when C_1 was created.

We also see that it does not hurt for *killregion* to leave an extra shapshot behind if it is followed by a *backtrack* operation. The *backtrack* will shrink the region to the size recorded in the snapshot—this is harmless even though it has just been shrunk to that size once. *backtrack* should also clear the killed-flag for each of the regions it shrinks.

The killed-flag does not need to be copied in snapshots, because it is never set when snapshots are created.

cut and other choice point operations

$$\mathit{pushchoice}_{C_0}\ ..(A)..\ \mathit{pushchoice}_{C'}\ ..(B)..\ \mathit{cut}_{C'}\ ..(C)..\ \mathit{cut}_{C_0}$$

We do not need to provide special support for this situation (where a matched *pushchoice–cut* pair occurs before C_0 is cut away). If the first *cut* works correctly, the state before the second cut will be exactly as after

$$\mathit{pushchoice}_{C_0}\ ..(A)..\ ..(B)..\ ..(C)..\ \mathit{cut}_{C_0}$$

Likewise,

$$\mathit{pushchoice}_{C_0}\ ..(A)..\ \mathit{backtrack}\ ..(B)..\ \mathit{cut}$$

causes the *cut* to see the same state as after

$$\mathit{pushchoice}_{C_0}\ ..(B)..\ \mathit{cut}$$

5.3.2 Memory management for choice points and snapshots

It is now time to consider how to organise the memory where choice points and snapshots are stored.

The lifetimes of choice points obey a stack discipline, so it is natural to use stack allocation for them. An area of memory separate from the heap is set aside for storing choice points.

Each choice point consists of a termination list and a snapshot list. We now consider representations for each of those.

Termination lists

The key facts about termination lists are:

1. Every region belongs to one and only one termination list.
2. *makeregion* may add a region to the topmost choice point's termination list.
3. *killregion* (and *cut*) may remove a region from the topmost choice point's termination list.

4. *backtrack* empties the topmost choice point's termination list.
5. *cut* concatenates the topmost and next-to-topmost choice points' termination lists.

Given these usage patterns the natural choice is to organise the termination list as a doubly-linked list with forward and back pointers embedded in the region management records (These pointers do not need to be stored in snapshots). Then all of the necessary operations can be implemented in constant time.

Snapshots and snapshot lists

The key facts about snapshots and snapshot lists are:

1. Each snapshot list contains at most one snapshot for each region. A region may have multiple snapshots if they are in different snapshot lists.
2. *alloc* (and *killregion*) may add a snapshot to the topmost choice point's snapshot list.
3. *backtrack* empties the topmost choice point's snapshot list. The snapshots themselves become unused.
4. *cut* empties the topmost choice point's snapshot list. Some of the snapshots become unused, others must be inserted in the next-to-topmost choice point's snapshot list.

These requirements admit several possible memory management strategies.

One strategy we have considered is to allocate the snapshots in the regions themselves. This is an attractive idea, because the address of the snapshot itself would identify the values of the newest-page pointer and the fill count, so there would not be any need for storing them explicitly. The problem with the strategy is that it would be impossible to reclaim the memory used by snapshots that become obsoleted by a *cut*.

Another strategy would be to store the snapshots inside the choice points themselves. The only snapshot list that ever grows is that of the topmost choice point, so if the last element of a choice point was an extensible array of snapshots it would be free to grow into the unused part of the choice-point–stack area as long as the choice point was topmost.

This scheme is also attractive because it eliminates the need to reserve link fields in the snapshot to hold the snapshot list together. It would be slightly messy to reassign snapshots to another snapshot list, however.

In our prototype implementation we use a variant of the array strategy. We use a separate stack for storing snapshots and let each choice point contain a pointer to the first snapshot in its snapshot list. Then "the topmost choice point's snapshot list" becomes the snapshots between that pointer and the stack top.

The advantage of this variant is that in some cases snapshots can be reassigned to an earlier snapshot list without physically moving. The *cut*

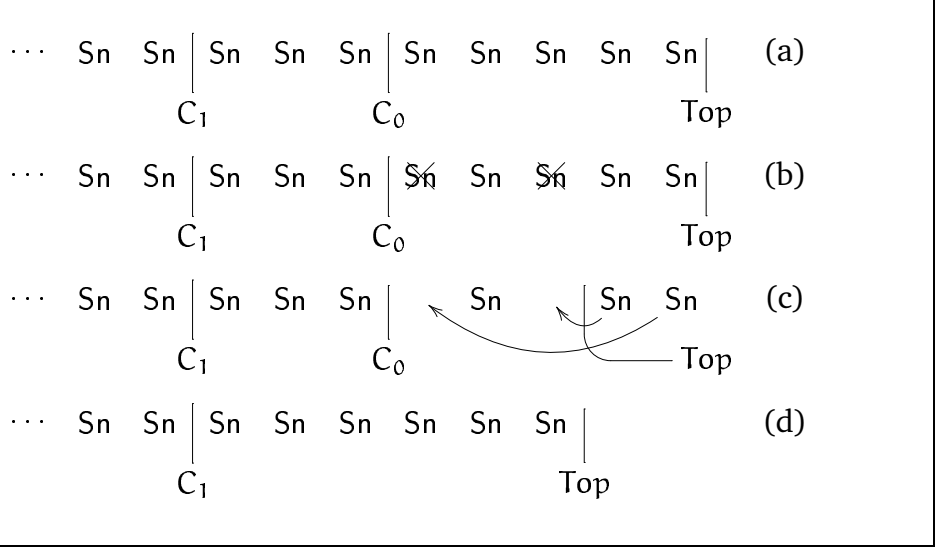


*Figure 5.1: Operations on a snapshot stack during a cut operation. Each "*Sn*" is a snapshot. (a) Before the cut. (b) Some snapshots become obsoleted. (c) The snapshot stack is compacted. (d) When* C_0 *disappears the two snapshot lists implicitly become concatenated.*

operation first traverses the snapshot list(s) of the choice point(s) to be cut away, potentially marking some of the snapshots as obsoleted. Afterwards it compacts the resulting list, that is, it overwrites the "lowest" obsoleted snapshots with the "highest" reassigned snapshot until the all of the reassigned snapshots are contiguous at the bottom of the stack segment. Then the list of reassigned snapshots is implicitly concatenated with the previous choice point's snapshot list. This algorithm means that if most of the snapshots are reassigned only a few of them actually need to be moved. See Figure 5.1.

5.3.3 Time efficiency

The bad news is that *cut* can, in principle, take a long time to execute and it isn't even possible to amortise it over other primitives. Each *cut* may need to reassign an aribitrary number of snapshots, and each choice point may need to be reassigned arbitrarily many times.

As an example of worst-case behaviour, consider a client program that

1. creates n regions
2. executes n *pushchoice* operations
3. allocates something in each of the n regions, triggering the creation of n snapshots.
4. *cut*s the n choice points away one by one. Each of the cuts need to reassign each of the n snapshots to the previous choice point.

The program executes $4n$ primitive operations, but the memory manager needs to decide n^2 times whether a choice point should be reassigned or obsoleted.

This might not be so bad as it looks, however. The example depends on a client program that uses the primitives in a specific sequence, but it is not at all clear that a client program that implements RP can ever behave in this way. We have not been able to construct an example Prolog program that actually exhibits this behaviour when translated to RP by our prototype translator.

Conjecture 5.1 *The region type system TRP we present in Chapter 9 implies that the first n memory-management primitives a TRP-correct program executes can be completed in $O(n)$ time, with a program-dependent constant factor.*

This not trivial claim is primarily supported by our inability to find counterexamples. We're not sure of how to construct a proof of it.

It is possible to construct an RP program that exhibits quadratic running time, see Figure 5.2. This program "cheats" by using logical variables to provoke snapshot creation in regions not passed as region parameters; it is not TRP-correct.

A second comforting factor is that cut can also take long time to execute in non-region-based Prolog implementations. Specifically, for the WAM [Aït-Kaci 1991, Section 5.11] a cut implies that a potentially slow "tidy_trail" operation must be performed because a cut may free memory on the WAM's local stack and trail entries referring to that memory must be deleted. The quadratic-time example above can be restated to use trail entries instead of snapshots to waste time; and then the example can be expressed as a Prolog example program.

5.3.4 Space efficiency

There are no additional space costs in supporting *cut* (except that we added a killed-flag to the region management records, but there are plenty of unused bits we can use for that), so the analysis and arguments in Section 5.2.3 still apply.

5.4 Logical variables

Now the only feature we need to add to our model before it can be used to implement RP is logical variables. Recall that we forbid the client program to change heap memory except immediately after the *alloc*; therefore the primitives we have defined so far cannot be used to implement instantiable variables. We now define primitives to allow a controlled amount of destructive update:

allocvar: ρ: REGION $\rightarrow$ α: pointer

Allocate a machine word of memory in the region ρ, and initialise it to have the value uninst (the client program and the implementation

```
main[] :- enter → ()
          & makeregion → R0
          & construct at R0 nil() → list
          & call manyvars [] (list) → ()
          & fail.
manyvars[] :- enter → (tail)
              & makeregion → TEMP
              & builtin get_code() → (bit at TEMP)
              & destruct bit as 1 → ()
              & killregion TEMP
              & cut
              & makeregion → MYREG
              & makevar at MYREG → var
              & construct at MYREG cons(var, tail) → list
              & call manyvars [] (list) → ()
              & killregion MYREG
              & exit ();
            enter → (list)
              & call manycuts [] (list) → ()
              & exit ().
manycuts[] :- enter → (count, list)
              & destruct count as cons → (_, newcount)
              & call manycuts [] (newcount, list) → ()
                % The following cut takes O(n) time because
                % the n variable instantiations in manyinsts
                % took place in different regions and each left
                % a snapshot that must now be reassigned
                % to the next choice point.
              & cut
              & exit ();
            enter → (zero, list)
              & call manyinsts [] (list) → ()
              & exit ().
manyinsts[] :- enter → (list)
               & destruct list as cons → (head, tail)
               & cut
               & unify head = tail
               & call manyinsts [] (tail) → ()
               & exit ();
             enter → (list)
               & destruct list as nil → ()
               & exit ().
```

Figure 5.2: A pathological RP program with quadratic memory-management overhead. The program reads an unary number n from the input stream and performs $\Theta(n)$ memory-management operations that take $\Theta(n^2)$ time.

are supposed to agree on which bit pattern that means). The client program promises never to write anything to the allocated word except by using the *instantiate* primitive.

instantiate: α: pointer, x: word $\rightarrow$ *(nothing)*

The pointer α must be the result of an earlier *allocvar* operation. The result of *instantiate* is to write the value x into the word pointed to by α. A later *backtrack* operation may undo the effect of *instantiate*, that is, reset the the word pointed to by α to uninst.

The client program may not use *instantiate* twice with the same α except when the first operation has been undone by a *backtrack* operation.

The *allocvar* primitive implements RP's makevar instruction; *instantiate* is used in the implementation of unify. (It should be noted that a Prolog-to-P translator need not use these instructions at all when compiling functional-style Prolog code. They are only when Prolog-level variables are used in a nontrivial way).

5.4.1 Implementation

The traditional way of implementing logical variables in an environment with backtracking is to have a global data structure called the **trail** which is simply an array of addresses to memory words that have been instantiated. Backtracking works by traversing an appropriate segment of the trail, resetting all of the referenced words to uninst.

This scheme does not work well in the region-based context: When a *killregion* operation frees memory in a region, any references to the freed memory would need to be removed from the trail, lest the trail might grow arbitrarily large if the client program never backtracks—even though the *killregion*s may keep the size of the *heap* bounded.[7]

Instead we choose to let each region have a private trail. The trail is a list of references to those variables in the region that have become instantiated. Instead of representing the list as an array (which would impractical for the same reasons that a region cannot occupy a contiguous part of the heap) we decide to use a linked list with the links allocated in the region's payload area. Because each variable can not be in the trail more than once, we can save space by letting the variable be embedded in the trail list cells rather than pointed-to by them.

Thus the representation of a variable that has been instantiated becomes a two-word structure. One word is the **public** word the client program sees, containing the "real" contents of the variable. The other, **private** word is a

[7] Contrarily to what (at least the author's) intuition suggests it would actually not be *unsound* to leave a global trail unchanged when deleting a region. The trail might contain references to cells that are now freed and later become used for something else—but those trail references are only going to be used in a backtracking operation that undoes the "something else" allocation anyway.

pointer to the next variable in the trail. We decide that variables that have been instantiated late point to variables that have been instantiated early, because that makes the trail operations we now describe easiest.

We extend region management records with a pointer to the head of the trail, that is, to the most recently instantiated variable in the region. The head-of-trail pointer is copied in snapshots.

This structure makes the job of *backtrack* with respect to variables easy:

1. When shrinking a region according to a snapshot met druring *backtrack*, repeat the following steps as long as the head-of-trail pointer in the region management record is different from that saved in the snapshot:
2. Reset the variable pointed to by the region management record's head-of-trail pointer to uninst.
3. Let the new head-of-trail pointer be a copy of the link word of the just-reset variable.

There is no reason to touch the trails for regions found in the termination list. As these regions are totally deallocated it does not matter whether or not the variables found in them are reset to uninst.

The same untrailing procedure should be followed by *killregion* when it shrinks a region on behalf of a later *backtrack* operation (as described on page 65); and by *cut* when it restarts an unfinished *killregion* operation (page 70).

We can also specify what *instantiate* should do:

1. Make sure that the region has an up-to-date snapshot, the same way *alloc* does it.
2. Write the specified data into the public word of the variable.
3. Write the head-of-trail pointer from the region's management record into the private word of the variable.
4. Let the head-of-trail pointer point to the variable itself.

but we have glossed over an important detail: Which region is "the region"? It is the region where the variable has been allocated, but how should *instantiate* know where that is? By design, the client program does not specify it when it calls *instantiate*—the client program simply does not know which region it is unless we changed RP to require that each unify instruction somehow specify the regions of each of the variables that might be instantiated.

Our solution is to store a pointer to the region in the variable itself. Until the variable is instantiated, the first word of the variable contains the uninst marker which the client program can inspect, but the second word is only used after the instantiation. We can use this second word to store a pointer to the region as long as the variable is uninstantiated. The pointer is not needed once the variable has been instantiated; when *backtrack* must restore the pointer as it uninstantiates the variable, it knows well in which region it found the trail entry.

We can now implement *allocvar*:

1. Allocate two words in ρ with the plain *alloc* operation.
2. Write uninst into the public words, and ρ into the private one..

5.4.2 Time efficiency

It is easy to see that *allocvar* and *instantiate* are both constant-time operations.

backtrack may use non-constant time for resetting variables according to the trail information. This is inevitable; similar time is used for other memory-management models as well[8]. Also, this time can be amortised over the preceding *instantiate* operations in a straightforward way.

The work with resetting variables can also occur in *killregion* or *cut*. This is unfortunate, because *killregion* used to be a constant-time operation. The extra cost of *killregion* is amortisable and only applies to regions where variables might have been allocated. Nevertheless it still makes it potentially difficult to reason about "from A to B" running times, and there can well be interesting A–B intervals that contain *killregion* instructions.

5.4.3 Space efficiency

In a conventional implementation with a global trail, a variable occupies one word of heap for its entire lifetime and one word of trail once it has been instantiated.

In our model, a variable occupies two words of heap for its entire lifetime.

Thus the traditional implementation is more efficient for variables that are never instantiated, or only instantiated late in their life. Our hope is that such variables are rare.

5.5 Summary of the implementation

We now give a summary of our primitives and the implementation we have developed.

5.5.1 Primitives

makeregion: *(nothing)* → ρ: REGION

Create a new region ρ.

[8]Here we ignore systems such as Mercury [Somogyi et al. 1996] where, thanks to potent mode analyses, instantiated variables do not need to be reset at backtracking at all. A region-based manager for such a system would also not need to reset variables, and this entire section could be ignored.

killregion: ρ: REGION → *(nothing)*

Destroy the region ρ. Any memory that has been *alloc*ated in ρ becomes available for allocation in other regions. It is an error to used ρ in any primitives after *killregion* has been called.

alloc: ρ: REGION, n: integer → α: pointer

Allocate n machine words of memory "in" the region ρ. The returned pointer points to the first of n consecutive words of memory which are guaranteed not to be used for any other purpose until a *killregion* operation on ρ is performed.

The client program promises not to write to the allocated cells other than filling values into them immediately after the *alloc* operation.

allocvar: ρ: REGION → α: pointer

Allocate a machine word of memory in the region ρ, and initialise it to have the value uninst (the client program and the implementation are supposed to agree on which bit pattern that means). The client program promises never to write anything to the allocated word except by using the *instantiate* primitive.

instantiate: α: pointer, x: word → *(nothing)*

The pointer α must be the result of an earlier *allocvar* operation. The result of *instantiate* is to write the value x into the word pointed to by α. If a *backtrack* operation is later performed, the word pointed to by α will be reset to uninst.

The client program may not use *instantiate* twice with the same α except when the first operation has been undone by a *backtrack* operation.

mark: *(nothing)* → δ: MARK

Pushes a **mark** onto the choice point stack and returns a magic token δ which identifies the mark uniquely.

pushchoice: *(nothing)* → *(nothing)*

This primitive conceptually pushes the entire state (*i.e.*, the size, location and contents of each existing region) of the memory manager onto an (implicit and global) **choice point stack**.

backtrack: *(nothing)* → *(nothing)*

Replaces the memory manager's state with the topmost state on the choice point stack. The choice point is left on the choice point stacks; that is, if two *backtrack* operations directly follow each other, the latter of them has no (observable) effect.

cut: δ: MARK → *(nothing)*

Pops saved states and/or marks off the top of the choice-point stack until the mark δ is at the top of the stack. The "main" state of the heap is not changed.

5.5.2 Data structures

The address space of an RP program is divided into the following major areas:

- The **heap**, divided into pages.
- The **snapshot stack**.
- The **choice-point stack**.
- The **local stack** (which we have not mentioned yet. The client program uses it for its private purposes).
- Global variables.
- Program text.

Unlike for the WAM, we do not assume anything about the relative positions of these memory areas.

Pages

Pages make up the heap. A page consists of

- A number of **payload** words. Our prototype implementation uses 15 payload words per page.
- A **link** word. The link field is used to tie pages together in the free list, and in regions. In region, the links in older pages point to newer pages.

Region management records

The management record in each region occupies the first payload words in the oldest of the region's pages. It consists of

- A **newest-page pointer**.
- A **free-count**. The number of unallocated payload words in the newest page.
- An **owner-of-newest-snapshot pointer**. *makeregion* lets this point to the topmost choice point even though no snapshot exists there.
- A **newest-snapshot pointer**. *makeregion* initialises this to be NULL.
- A **killed-flag**. Initially false, set to true by *killregion*, reset to false by *backtrack*. Is not saved in snapshots.
- A **head-of-trail pointer**. *makeregion* initialises this to be NULL.
- Forward and backward pointers for the termination list. Not saved in snapshots.

Snapshots

Snapshots are allocated on the snapshot stack. Each consists of

- A pointer to the region for which the snapshot applies.
- A copy of the region's newest-page pointer.
- A copy of the region's free-count.
- A copy of the region's owner-of-newest-snapshot pointer.
- A copy of the region's newest-snapshot pointer.
- A copy of the region's head-of-trail pointer.

Choice points

A choice point is allocated on the choice-point stack. It consists of

- Forward and backward pointers for the **termination list**. The termination list is a doubly-linked circular list. One of its elements is the choice point, the rest are region management records.
- A copy of the (global) snapshot stack pointer when the choice point was created.

Variables

Variables are allocated in the payload area of region pages along with other of the client program's data. A variable can be uninstantiated or instantiated. It is a two-word structure consisting of

- A **public word** which can be inspected by the client program. When the variable is uninstantiated the public word is a special "uninst" bit pattern.
- A **private word** which is only used by the memory manager. When the variable is uninstantiated the private word is a pointer to (the management record for) the region that contains the variable. When the variable is instantiated the private word is a trail link and points at the variable that was the top of the trail before this variable was instantiated.

Global data for the memory manager

Stack pointers for the choice point stack and the snapshot stack. A pointer to the head of the free-list.

5.5.3 Algorithms

makeregion: *(nothing)* → ρ: REGION

1. Obtain a fresh page from the free-list or by allocating new pages from the operating system.
2. Set the page's link word to NULL.
3. Initialise the first words as a region management record, using the values described for each word in Section 5.5.2. Link the page into the topmost choice point's termination list.

killregion: ρ: REGION → *(nothing)*

1. Make sure the region has an up-to-date snapshot.
2. If the newest-snapshot pointer is NULL after step 1, then add the entire region to the free-list. Otherwise continue with the following steps.
3. Shrink the region by performing steps 3A–3F of the *backtrack* primitive (below) on the newest snapshot for the region.
4. Set the region's killed-flag.

alloc: ρ: REGION, π: integer → α: pointer

1. Make sure the region has an up-to-date snapshot.
2. Do the following steps 2A–2E if the free-count is less than π:
   - 2A. Obtain a fresh page from the free-list or by allocating new pages from the operating system.
   - 2B. Set the link word of the new page to NULL.
   - 2C. Set the link word of the previously newest page (the one pointed to by the newest-page pointer for the region) to point to the new page.
   - 2D. Let the newest-page pointer point to the new page.
   - 2E. Set the free-count to the number of payload words per page.
3. Decrease the free-count by π; return a pointer to the first of the words thus reserved.

allocvar: ρ: REGION → α: pointer

1. Allocate two words in ρ as if by *alloc*.
2. Write uninst into the public (first) word, and ρ into the private (second) word.
3. Return a pointer to the public word.

instantiate: α: pointer, χ: word → *(nothing)*

1. Locate the region pointed to by the private word of the two-word structure pointed to by α.
2. Make sure that the region has an up-to-date snapshot,

3. Write the specified data into the variable's public word.
4. Write the head-of-trail pointer from the region's management record into the variable's private word.
5. Let the head-of-trail pointer be α.

mark: *(nothing)* $\rightarrow \delta$: MARK

Return a pointer to the topmost choice point.

pushchoice: *(nothing)* $\rightarrow$ *(nothing)*

1. Create a new choice point on the choice point stack with an empty termination list. (The snapshot list implicitly becomes empty when the current top-of-snapshot-stack is copied into the new choice point).

backtrack: *(nothing)* $\rightarrow$ *(nothing)*

1. For each region found in the topmost choice point's termination list, add the entire region to the free-list.
2. Afterwards the termination list is empty.
3. Do the following **shrink** operation for each snapshot that is newer than[9] the saved snapshot-stack top in the choice point:
   3A. Locate the region ρ referred to by the snapshot.
   3B. Repeat the following steps as long as the head-of-trail pointer in the region's management record is different from that saved in the snapshot:
      3B(i). Reset the variable pointed to by the region management record's head-of-trail pointer, by writing uninst into the public word and ρ into the private word.
      3B(ii). Let the new head-of-trail pointer be the *previous* value of the variable's private word.
   3C. Update the region's newest page's link word to point to the first page in the free-list.
   3D. Update the management record with the saved data from the snapshot (including the saved newest-page pointer). Clear the killed-flag.
   3E. Let the new "head of the free-list" be the successor of the page that is the region's newest page according to the new newest-page pointer.
   3F. Write NULL into the link word of the newest page be NULL, terminating the linked-list structure of the region.
4. Set the snapshot-stack top to the one saved in the choice point.

[9] That is, stored higher on the snapshot stack than

cut: δ: MARK $\rightarrow$ *(nothing)*

1. Call the snaphot pointed to by δ (which is topmost of those snapshots that are *not* cut away) C_δ.
2. Move all regions found in any of the cut choice point's termination lists to C_δ's termination list.
3. Do the following for each snapshot in a snapshot list belonging to one of the cut choice points, beginning with the newer choice points:
   - 3A. If the saved owner-of-newest-choice-point pointer in the snapshot points to a choice point that is older than C_δ, then reassign the snapshot to C_δ's snapshot list. Proceed with the next snapshot.
   - 3B. Otherwise, the snapshot is obsoleted. Remove it without shrinking the region.
   - 3C. If the snapshot was obsoleted *and* the killed-flag of the region it referred to was set, then repeat steps 2 through 4 of the *killregion* algorithm for the region.
4. Let C_δ be the new topmost choice point.

5.5.4 Optimizations

We now suggest a number of small optimizations with respect to the design we have just summarised. In general, the optimizations aim at reducing the size of snapshots and region management records as much as possible without costing more than increased constant factors in the running-time of the primitives. They also make the algorithms harder to understand, which is why we have not incorporated them in the description from the start.

With the optimizations we describe here, a region management record in or prototype implementation takes up 6 words (which might be reduced to 5 by an optimization we have not implemented), and a snapshot takes up 4 words.

A unified end-of-region pointer

As we have described the region management record, the newest-page pointer and the free-count are stored in separate words. We can pack these values into a single word by deciding that

- The size of a page (including the link field) must be a power of 2.
- The address of each page should be a multiple of the page size.
- The link word is stored at the end of the page.
- The payload words in each page are allocated highest address first.
- The newest-page pointer and free-count fields of the region management structure are replaced with an **end-of-region pointer** which points to the most recently allocated word in the region.

With these conventions the newest-page pointer and the free-count can be recovered from the end-of-region pointer by simple bit masking operations.

This optimization is used in our RP prototype. Of course, the snapshot copies of the newest-page pointer and free-count is similarly optimized.

Encoding the killed-flag

The killed-flag in region management records needs not occupy a word by itself. It is easy to pack the killed-flag into the least-significant bit of one of the pointer fields (we're assuming a byte-addressed machine), but there is an even simpler solution than that.

The killed-flag is only set by *killregion* operations. Before the killed-flag is set, the region has been shrunk back to the size recorded in the newest snapshot, and that snapshot continues to exist. This means that the region-size data in the management record is redundant as long as the region is killed.

In our prototype the kill flag is set by pointing the region's head-of-trail pointer to a reserved location that can never be a genuine trail entry. The real head-of-trail is found in the snapshot.

The choice of the head-of-trail pointer for this purpose is arbitrary. Any of the words that are copied in the snapshots would do.

Saving one further word in each snapshot

Suppose at some time there are k snapshots for a region ρ. Each of the k snapshots contain a pointer to ρ, but only the newest of them risks actually being accessed. Thus $k-1$ words store redundant information. If we can find one more redundant word, there'll be k redundant words in total, and it's concievable we can make each snapshot one word smaller.

There are no redundant words in the newest snapshot, but there is one in the *oldest* one: the copy of the newest-snapshot pointer in the oldest snapshot is always NULL.

That makes two redundant words in the oldest snapshot, no redundant words in the newest snapshot, and one redundant words in every snapshot in between. If we move one word of non-redundant data one level back in the chain of snapshots, we can end up having freed one word in each snapshot.

In our implementation we choose to use the saved owner-of-newest-shapshot pointer for this. That means that the three snapshot words containing

- A pointer to the region for which the snapshot applies.
- A copy of the region's owner-of-newest-snapshot pointer.
- A copy of the region's newest-snapshot pointer.

get replaced by two words:

- The first word is a copy of the newest-snapshot-pointer, *except* in the oldest snapshot; there it contains the copy of the owner-of-newest-snapshot pointer.
- The second word points to the owner of the snapshot itself (*i.e.*, what was previously the owner-of-newest-snapshot copy of the above snapshot), *except* in the newest snapshot; there it contains a pointer to the region for which the snapshot applies.

We assume a byte-addressed machine, so we can use the least significant bit of the first of these words to signal whether the snapshot is an oldest snapshot or not.

We do not need any bits to tell which snapshots are newest: The only snapshots we ever handle are found in the snapshot list of the topmost choice point or pointed to by the newest-snapshot pointer in a region management record, and those snapshots are *always* newest.

This optimization makes is more involved to create and remove snapshots, but each of the necessary operations are still constant-time operations.

Termination of a region's list of pages

The algorithms have assumed that the link word of the newest page for a region should be set to NULL to signal the end of the list of pages. In reality this link field is never inspected; the last page in the list is identified by being the one the region's newest-page pointer points to, not by having a NULL link word.

We could save one word of the region management record by storing it in the unused link word.

Our prototype implementation does not incorporate this optimization except that it does not write NULL into the link word when the region's size changes.

5.6 A prototype implementation of RP

We [Makholm 2000] have implemented a prototype implementation of RP based on the principles described above. The prototype has been used in the experiments we report on in Chapter 11.

The prototype consists a translator written in Moscow ML [Romanenko and Sestoft 1999] which translates RP code to a client program written in C, and a region-based run-time module written in C.

5.6.1 The run-time module

The larger part of the run-time module consists of a region-based memory manager which closely follows the design described in this chapter. It statically allocates a fixed-size stack area where the choice-point stack grows from high addresses and the snapshot stack grows from low addresses. Heap

pages are allocated in batches of 100 from the C library's `malloc()` function. The default size of pages is 15 payload words, but that figure can be changed when the system is configured (before the run-time module is compiled).

The memory manager contains code to collect statistics information such as the number of obsoleted snapshots per *cut* operation, the number of region shrink operations per *backtrack* operation, the number of freed words and pages per region shrink operation, and so on. The statistics collection slows down the program considerably, so we use C's conditional compilation facilities to turn it off before doing timing runs.

A second part of the run-time module contains a `main()` function and a unification function which is called by the client program to implement a unify instruction. The unification function then makes a number of *instantiate* operations and decides whehter its operands are unifiable.

The unification function does not implement the occurs check, so it might create circular structures. It is robust enough to be able to unify circular structures with each other.

5.6.2 The RP-to-C translator

The RP-to-C translator translates an RP program into a C client program which is a modestly direct simulation of the RP semantics given in Section 4.4.

The semantics' ⟨*Store*⟩ corresponds to the heap and is managed by the region-based memory manager. The current contents of the heap models the ⟨*Store*⟩ in the first ⟨*Frame*⟩ of the state. The memory manager's choice-point stack models the rest of the frames. Each choice point models the set of ⟨*Frames*⟩ elements created by a single call instruction; in the semantics these frames always contain the same ⟨*Store*⟩. The prototype extends the choice point structure described on page 81 with extra words to store the address of next clause to be tried, and its (region and value) parameters.[10]

Predicate calls are implemented by building a choice point which contains all clauses of the predicate-to-be called and then backtracking to the first of them. The last clause for a predicate then cuts the choice point away. For simplicity, this is done even when calling predicates that have only one clause.

The simulation is less direct in the handling of the semantics' ⟨*RegEnv*⟩ and ⟨*Env*⟩ objects. These environments hold the "local variables" for the currently executing predicate. As in traditional implementations of procedural languages these values are stored in statically allocated locations within a **environment frame** on a **local stack**. Managing the local stack is not as simple as for procedural languages, because care must be taken not to overwrite stack locations as long as their contents may be needed after backtracking.

[10]This means that the abstract interface between the client program and the memory manager is not strictly enforced in the prototype. We feel that maintaining separate choice points for use by the client program in unison with the memory manager's choice point stack would be ridiculous.

We use standard methods from the WAM [Aït-Kaci 1991] to decide where new environment frames can be allocated.

The translator spends quite some effort on keeping the environment frames small, so that our experiments yield reliable data on how much extra local stack space it takes to keep track of regions. Data that does not need to survive a predicate call is not allocated on the local stack at all. Region or value parameters that can be found in the choice point that started the current predicate (because it has not yet been `cut` away) are not stored in the call stack either. Those frame slots that do need to be allocated are carefully arranged such that they can (perhaps) be reused after their last use; this is called **environment trimming** by Aït-Kaci [1991]. We use a primitive **determinacy analysis** to locate points within each clause where it is safe to expand or reorder the environment frame.

The ⟨*Dump*⟩ and ⟨*Cont*⟩ objects are also stored in the environment frames, in the form of pointers to earlier choice points and environment frames.

As for the control flow within the client program, the call–return discipline of C is of little use because it does not support backtracking. We implement a block of straight-line code (which models the part of an RP ⟨*body*⟩ that comes between two `call` instructions) as a C function which returns the address of the next function to execute. The `main()` function in the run-time module contains a loop that repeatedly calls the "current" function, letting its return value be the new current function.[11]

Output parameters from predicates are transferred in a global array. The translator supports a limited form of "last call optimization": If a clause ends with

(...)
& call pr $[\rho_1, ..., \rho_i]$ $(x'_1, ..., x'_j) \rightarrow (x_1, ..., x_k)$
& exit $(x_1, ..., x_l)$
(...)

with $k \geq l$ (*i.e.*, when no reordering of the global array of output parameters is necessary before returning), the call is treated as a tail call and the caller's entire environment frame may be deallocated before the jump to pr (unless it may be needed by backtracking—that is determined at run time). A similar optimization is used if the last two instructions of a clause are `call` and `fail`.

[11] This means that each predicate call "costs" two C function calls (one when jumping to the called predicate and one when returning from it) in addition to the cost of managing an explicit call stack that is separate from the C call stack. Henderson et al. [1995] describe methods by which some of this cost can be avoided, but we found them too complex to justify using them in a proof-of-concept prototype.

Chapter 6

TGP: a type system for GP

In this and the forthcoming chapters we develop type systems for the xP languages.

The question naturally arises: why bother with types at all? After all, Prolog is a typeless language, and as the previous chapters show, it is possible to succintly define and implement the region-based language RP without any notion of types. The answer is that types are a convenient tool for *creating* RP programs mechanically.

It is entirely possible to write a compiler from Prolog to P which works without any concept of types. The trouble arises when we need to add region annotations to the P program to get an RP program. Any region inference algorithm needs to reason about which regions the different parts of the terms manipulated by the programs are allocated in. To do this at all, the algorithm must have some way of *identifying* the different parts of the terms manipulated by the program. In other words, we need a structured, compile-time, description of the structure of the terms that can be bound to a register at run time. That is what types are.

It follows that our perspective when talking about types is quite different from an average language designer's one. In particular, the common rationale for types that "types alert the programmer, at compile time, to many silly programming errors" is not relevant here. For all we care, the human programmer writes an untyped Prolog program; types are only used internally in the compiler that translates it to RP. (This does not mean that the ideas herein cannot be applied to typed logic programming languages; we are simply noting that the types we speak of do not *need* to be visible to the programmer).

Another slogan one frequently sees is that "well-typed programs do not go wrong". For the special notions of `wrong` our semantics define, this is actually true for the type systems we present, but it should be kept in mind that it is not our primary reason for talking about types.

The slogan that suits the role of types in this thesis best is that "types help the compiler reason about the program it compiles". (This view is the basis for conferences such as TIC [1998, the introduction to the proceedings contain references to many other applications of the idea]).

The two primary type systems in our design are TP which is used in the

translation from Prolog to P and TRP which is used by the region inference. The TGP system we introduce in this chapter is roughly the "common subtheory" of those two type systems.

The basic principles of the type system is reminiscent of that of Mycroft and O'Keefe [1984], with the principal differences that ours does not include polymorphism and does not require any special syntax for recursive types. Both of these differences are motivated by our wish to make the type system invisible to the programmer (who, with Mycroft and O'Keefe's system, is supposed to provide explicit declarations of recursive type constructors and the types of predicates). It is easy to reconstruct recursive types without having explicit names for them (see Section 8.8.1), but no good algorithms are known for inferring polymorphic predicate types without explicit user annotations, given that predicates can be mutually recursive.

6.1 The syntax and meaning of TGP types

Our plan is to attach to every register in a GP program a type built from this grammar:

$$\begin{array}{rcl} \tau & ::= & \mathsf{int} \\ & | & f_1(\tau_{11}, ..., \tau_{1|f_1|}) + \cdots + f_n(\tau_{n1}, ..., \tau_{n|f_n|}) \end{array}$$

where, in the second line, the f_is are all distinct and none of them are numbers.

We use the general shorthand notation of writing f instead of $f(\,)$ whenever f is a nullary functor (*i.e.*, $|f| = 0$). We also use the abbreviation

$$\sum_i f_i(\tau_{ij})_j \quad \equiv \quad f_1(\tau_{11}, ..., \tau_{1|f_1|}) + \cdots + f_n(\tau_{n1}, ..., \tau_{n|f_n|})$$

to conserve space when writing formulas about types.

We use the variables τ, τ', τ_i, etc. to range over individual types. $\mathbb{T}$ is the set of all types.

6.1.1 The meaning of types

The type of register describes in a straightforward way the terms that may be bound to that register. For example, the type

$$f(a + b, g(\mathsf{int})) + g(a + c)$$

describes the terms "$f(a, g(42))$", "$f(b, g(42))$", "$g(a)$", and "$g(c)$" but neither of "$f(a, g(a))$", "$f(c, g(42))$", or "$g(42)$".

Formally, we define the meaning of a type by

Definition 6.1 *The function that takes a type τ to its meaning $[\![\tau]\!]$ is the (largest) function $\mathbb{T} \to \mathcal{P}(\langle Term \rangle)$ that satisfies the recursive specification,*

$$\begin{array}{rcl} [\![\mathsf{int}]\!] & = & \mathbb{Z} \\ [\![\sum_i f_i(\tau_{ij})_j]\!] & = & \bigcup_i \{\, f_i(t_1, ..., t_{|f_i|}) \mid t_1 \in [\![\tau_{i1}]\!], ..., t_{|f_i|} \in [\![\tau_{i|f_i|}]\!] \,\} \end{array}$$

Box 6.1—MIXING INTEGERS WITH OTHER STRUCTURES

Our type system has no type that describes both of "$\mathtt{g}(42)$" and "$\mathtt{g}(\mathtt{a})$", so it might appear that it is impossible to translate a Prolog fragment such as

$$\ldots,\ \mathtt{foo}(\mathtt{g}(42),\mathtt{X}),\ \mathtt{foo}(\mathtt{g}(\mathtt{a}),\mathtt{Y}),\ \ldots$$

into TP.

Our solution is to "wrap" all numbers in the Prolog source in a unary functor $\mathcal{N}$ which we reserve for this purpose. Thus, early in the translation, the Prolog fragment would be replaced by

$$\ldots,\ \mathtt{foo}(\mathtt{g}(\mathcal{N}(42)),\mathtt{X}),\ \mathtt{foo}(\mathtt{g}(\mathtt{a}),\mathtt{Y}),\ \ldots$$

where the first parameter to `foo` has the perfectly good type $\mathtt{g}(\mathcal{N}(\mathsf{int})+\mathtt{a})$. Calls to predefined predicates (remember that the input and output from predefined predicates are all integers) get augmented with interface code to destruct and construct the $\mathcal{N}$s.

Later in the compilation it may be found that the $\mathcal{N}$ wrapping is unnecessary. This is the case when the type of a term becomes "$\mathcal{N}(\mathsf{int})$" (rather than, *e.g.*, "$\mathcal{N}(\mathsf{int})+\mathtt{a}$"). In that case the $\mathcal{N}$s can be optimized away by replacing each of the construct and destruct instructions that refer to such an $\mathcal{N}$ type with alias instructions and adjusting the type information accordingly.

This scheme makes run-time tags unnecessary for most numbers in the program while still allowing the programmer to mix numbers and structures when she wants (or by mistake). This is a side advantage of using types internally in a compiler for a typeless language, but has nothing in particular to do with regions.

6.1.2 Recursive types

We need to allow types to be recursive so that we can express, for example, the type of all lists of integers which is a solution to the equation,

$$\tau_\ell = \mathtt{nil} + \mathtt{cons}(\mathsf{int},\tau_\ell)$$

In the compiler such types are easily implemented by letting the representation of a type be a graph rather than a tree. In Section A.1 of Appendix A we provide a mathematical underpinning of this simple implementation strategy. With that as our formal excuse we shall henceforth understand that types may be recursive, even though we have not introduced a formal recursion device such as a μ operator or global type names.

When we try to apply Definition 6.1 to recursive types, it is not trivial that it really defines anything. We give a general discussion of such definitions in Section A.1.4. In the case of Definition 6.1 the point is that the codomain of $[\![\cdot]\!]$ is a power set, hence a complete lattice. The fixed point theorem for complete lattices then guarantees that a unique largest solution exists.

6.1.3 Predicate types

We assign a special form of types to entire predicates: a **predicate type** has the form $\left\langle \frac{\tau_1,\ldots,\tau_{\lceil pr \rceil}}{\tau'_1,\ldots,\tau'_{\lfloor pr \rfloor}} \right\rangle$, where the τ_is and τ'_is are the types of the predicate's input and output parameters, respectively.

6.2 Well-typed GP programs

Recall that formally we assume that no register name is defined more than once in a GP program. That makes it easy to view a typing as something that is given in addition to an already existing GP program.

Definition 6.2 *A **typing** for a GP program is a mapping $\mathcal{T}$ which to each register name x in the program assigns a type $\mathcal{T}(x) \in \mathbb{T}$ and to each predicate name pr a predicate type $\mathcal{T}(pr) = \left\langle \frac{\tau_1,\ldots,\tau_{\lceil pr \rceil}}{\tau'_1,\ldots,\tau'_{\lfloor pr \rfloor}} \right\rangle \in \mathbb{T}^{\lceil pr \rceil + \lfloor pr \rfloor}$.*

Figure 6.1 defines a set of typing rules which determine when a typing "fits" a given GP program. The rules define a relation

$$\mathcal{T} \vdash \langle body \rangle \rightsquigarrow (\tau_1, \ldots, \tau_n)$$

which intuitively means, "the $\langle body \rangle$ is well-typed under $\mathcal{T}$, and if it ever executes an exit pseudoinstruction the return values will have the types τ_1 through τ_n", and a relation

$$\mathcal{T} \vdash \langle predicate \rangle$$

which intuitively means, "the $\langle predicate \rangle$ is well-typed under $\mathcal{T}$".

Definition 6.3 *A typing $\mathcal{T}$ for a GP program is a **well-typing** iff the judgement "$\mathcal{T} \vdash \langle predicate \rangle$" can be derived by the rules in Figure 6.1 for each predicate in the program.*

*A GP program for which a well-typing exists is a **well-typed** program. A well-typed program together with its well-typing is a **TGP program**.*

6.3 Well-typed GP programs do not go wrong_2

We can now prove that "TGP programs do not go wrong_2". Recall that wrong_2 is the kind of run-time error that occurs when an input argument to a built-in predicate is not an integers. Thus, Theorem 6.4 below states that TGP prevents a program from using non-numeric terms as integers:

Theorem 6.4 *When a TGP program is executed according to the semantics in Section 2.5, the error state wrong_2 never arises.*

Theorem 6.4 follows directly from the following main lemma:

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{call } pr(x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr \rfloor}) \;\&\; b \rightsquigarrow \vec{\tau}} \mathcal{T}(pr) = \left\langle \frac{\mathcal{T}(x_1),\ldots,\mathcal{T}(x_{\lceil pr \rceil})}{\mathcal{T}(x'_1),\ldots,\mathcal{T}(x'_{\lfloor pr \rfloor})} \right\rangle$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{builtin } pr(x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr \rfloor}) \;\&\; b \rightsquigarrow \vec{\tau}} \begin{cases} \forall i : \mathcal{T}(x_i) = \text{int} \\ \forall i : \mathcal{T}(x'_i) = \text{int} \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{construct } f(x_1,\ldots,x_{|f|}) \rightarrow x' \;\&\; b \rightsquigarrow \vec{\tau}} \begin{cases} f_i \notin \mathbb{Z} \\ \mathcal{T}(x') = f(\mathcal{T}(x_1),\ldots,\mathcal{T}(x_{|f|}))+\ldots \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{construct } f() \rightarrow x' \;\&\; b \rightsquigarrow \vec{\tau}} \begin{cases} f \in \mathbb{Z} \\ \mathcal{T}(x') = \text{int} \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{destruct } x \text{ as } f \rightarrow (x'_1,\ldots,x'_{|f|}) \;\&\; b \rightsquigarrow \vec{\tau}} \begin{cases} f \notin \mathbb{Z} \\ \mathcal{T}(x) = f(\mathcal{T}(x'_1),\ldots,\mathcal{T}(x'_{|f|}))+\ldots \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{destruct } x \text{ as } f \rightarrow () \;\&\; b \rightsquigarrow \vec{\tau}} \begin{cases} f \in \mathbb{Z} \\ \mathcal{T}(x) = \text{int} \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{alias } x \rightarrow x' \;\&\; b \rightsquigarrow \vec{\tau}} \mathcal{T}(x) = \mathcal{T}(x')$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{unify } x_1{=}x_2 \;\&\; b \rightsquigarrow \vec{\tau}} \mathcal{T}(x_1) = \mathcal{T}(x_2) \qquad \frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\tau}}{\mathcal{T} \vdash \text{cut } \&\; b \rightsquigarrow \vec{\tau}}$$

$$\frac{}{\mathcal{T} \vdash \text{fail} \rightsquigarrow \vec{\tau}} \qquad \frac{}{\mathcal{T} \vdash \text{exit}(x_1,\ldots,x_n) \rightsquigarrow (\mathcal{T}(x_1),\ldots,\mathcal{T}(x_n))}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow (\tau'_1,\ldots,\tau'_{\lfloor pr \rfloor})}{\mathcal{T} \vdash pr \text{ :- enter} \rightarrow (x_1,\ldots,x_{\lceil pr \rceil}) \;\&\; b.} \mathcal{T}(pr) = \left\langle \frac{\mathcal{T}(x_1),\ldots,\mathcal{T}(x_{\lceil pr \rceil})}{\tau'_1,\ldots,\tau'_{\lfloor pr \rfloor}} \right\rangle$$

$$\frac{\mathcal{T} \vdash pr \text{ :- } c_1. \quad \cdots \quad \mathcal{T} \vdash pr \text{ :- } c_n.}{\mathcal{T} \vdash pr \text{ :- } c_1;\ldots;c_n.}$$

Figure 6.1: Typing rules for GP programs. In the rules for construct *and* destruct*, the notation "*$f(\tau_1,\ldots,\tau_{|f|})+\ldots$*" stands for any type* $\sum_i f_i(\tau_{ij})_j$ *such that for some* i *(not necessarily the first one),* $f_i = f$ *and* $\forall j : \tau_j = \tau_{ij}$.

Lemma 6.5 *When a TGP program with well-typing $\mathcal{T}$ is executed according to the semantics in Section 2.5, every $\langle$ValEnv$\rangle$ $\mathcal{E}$ in every state of the computation satisfies*

$$\forall x \in \mathrm{Dom}\,\mathcal{E} : \quad \mathcal{E}(x) \in [\![\mathcal{T}(x)]\!]$$

Lemma 6.5 can be proved by induction on the length of the computation. We do not give the individual induction steps (each corresponding to one of the next-state rules in Section 2.5); they all follow almost immediately from the induction hypothesis (*i.e.*, that the lemma holds for the previous state) and the typing rule for the instruction in question.

The induction step for exit pseudoinstructions needs the following auxiliary lemma which intuitively states that all pending predicate exits are always well-typed. The lemma itself can be proved directly by induction on the length of the computation.

Lemma 6.6 *Let a TGP program with well-typing $\mathcal{T}$ be given. Define the relation $\looparrowright \subseteq \langle$body$\rangle \times \langle$Cont$\rangle$ by*

- $b \looparrowright []$ *iff* b *has the form "... & fail".*
- $b \looparrowright ((\mathcal{E}', (x'_1, ..., x'_n), b', \delta') :: \kappa')$ *iff* $\mathcal{T} \vdash b \rightsquigarrow (\mathcal{T}(x'_1), ..., \mathcal{T}(x'_n))$ *and* $b' \looparrowright \kappa'$.

When the program is executed according to the semantics in Section 2.5, for every $\langle$Frame$\rangle$ $fr = (\mathcal{E}, b, \delta, \kappa)$ that occurs in the computation, $b \looparrowright \kappa$.

Chapter 7

TP: a type system for P

In this chapter we define the type system TP for P programs. The major difference between TGP and TP is that types in the latter system contains a primitive notion of **modes**.

The term "modes" is used in Prolog contexts about any of a number of schemes that (statically) encode information about where and when (uninstantiated) variables may appear in a logic program. The parts of a program that, according to the mode annotation, do not need to handle variables, can be executed faster because they do not need to check whether terms are variables or not.

As we are interested in memory management, the importance of modes is even greater: When the Prolog goal "$X = f(Y)$" is executed and the modes tell us that X is not an uninstantiated variable, the execution consists simply of inspecting a term that has already been created. If, however X can be a variable, the goal may need to allocate memory for an f somewhere. This difference is crucial for a region-based implementation.

In our general design, these matters are supposed to be solved in the translation from Prolog to P. A P program can use the destruct instruction to inspect only nonvariable terms, and must use another instruction deref to tell variables from nonvariables. The Prolog-to-P translator needs to know about modes so that it can avoid unnecessary deref instructions, but the region-based phases we present in Chapters 9 and 10 do not need to be concerned about modes.

Hence, the reader who is interested primarily in the explicitly region-based aspects of this thesis may skip directly to Chapter 9 without missing much that will be used in that and the following chapters.

7.1 Design principles for TP

Our basic approach is to use the TGP we developed in Chapter 6, except that we annotate each type τ with a **mode** $m \in \{\mathsf{bare}, \mathsf{ref}\}$ to form a type-and-mode combination $\mu = m.\tau$. Then, for example, "$\mathsf{bare}.\mathsf{int}$" describes something which is an integer, and "$\mathsf{ref}.\mathsf{int}$" describes something which is either a variable or an integer. The component types of structured types are

similarly annotated. For example, "$\mathsf{bare}.f_0 + f_1(\mathsf{ref}.\mathsf{int})$" describes something which is either an f_0 or an f_1 whose argument can be a variable or an integer.

We'll make this more precise in a little while, but first we'll take some time to describe how and why our system differs from existing mode systems.

There is a long and still living tradition of automatic mode analyses for Prolog programs, starting with Mellish [1981]. In those systems the mode of a (syntactic) variable is a quasi-dynamic concept: X's mode is different at different points of the clause in which appears. As the clause is executed the term bound to the (syntactic) variable gets more and more instantiated.

Our system TP is considerably *less* sophisticated than that. For us, a mode is part of a P register's type, which does not change over time. The mode merely points out where variables may exist at some time in the register's life time.

The reason why TP can get away with not changing the mode of a register during the execution of a clause is that the static scope of a P register determines when it has any value *at all*. The register scoping is reflected in the explicit parameter moding of P. It is not a coincidence we also call our parameter classification "modes": In fact the chief job of early mode systems (such as the one used by Mellish [1981]) was to distinguish input parameters from output parameters. Thus, with TP and parameter moding we our language now has two unrelated mode systems.

This design may seem inelegant, especially because the recent mode systems in the literature[1] promise to combine the precision we get from TP with the time sensitivity we get from register scopes. Thus arises the legitimate question why we chose to use this makeshift combination of two weak systems instead of one strong one.

Our simple reason is that what we do happens to work well enough for our purposes, and is relatively simple to describe and implement. It would not be inherently difficult to use an existing stronger and uniform mode system for constructing a region-based Prolog implementation; we simply preferred to spend our energy on other less well-understood parts of the system rather than on implementing a more advanced mode system.

Another reason is that we originally planned to use the subtyping ideas in Section 7.4 as an ingredient in a powerful region inference algorithm based on "region subtyping". As it turned out, we have not yet had time to develop those ideas fully (only briefly mentioning them in Section 12.1.9). That is the reason why we use an entire chapter on describing TP rather than just a section of Chapter 8.

Finally, the combination of parameter modes and a mode system very much like TP is used in Visual Prolog [Prolog Development Center 2000]. We originally intended that this project would have somewhat more to do with Visual Prolog than it ended up doing.

[1] For example, Somogyi [1987] which is the basis for the Mercury language [Somogyi et al. 1996].

7.2 The syntax and meaning of TP types

A **type** in TP is something built from this grammar:

$$\begin{array}{rcl} \mu & ::= & m.\tau \\ \tau & ::= & \mathsf{int} \mid f_1(\mu_{11}, ..., \mu_{1|f_1|}) + \cdots + f_n(\mu_{n1}, ..., \mu_{n|f_n|}) \\ m & ::= & \mathsf{ref} \mid \mathsf{bare} \end{array}$$

where, in the second production for τ, the f_is are all distinct and none of them are numbers.

We call a μ a **type** and m a **mode**. We try to avoid speaking of τs except in formulas, lest we confuse them with types.[2] $\mathbb{T}$ is still the set of all types μ.

As for TGP we need to understand that the types can be recursive.

Now, what does a TP type mean?

TP types are similar enough to TGP types that it is easy to see how a TP type μ may denote a set of ground terms: Just ignore the modes, and an analogue of Definition 6.1 on page 90 can be applied.

When we want to explain the modes, however, we have to leave the world of ground terms and use the store model developed in Section 3.1. Recall that in the store model the intuitive idea of a term is replaced by a *pointer* α into a *store* σ. Instead of defining which terms each type describes, we define a ternary relation

$$\sigma \vdash \alpha : \mu$$

which intuitively means, "In store σ, the term represented by α has type μ".

Definition 7.1 *The relation* $(\vdash:)$ *is defined co-inductively by the following inference system:*

$$\frac{}{\sigma \vdash \alpha : \mathsf{ref}.\tau}\, \sigma(\alpha) = \mathsf{uninst} \qquad \frac{\sigma \vdash \alpha' : \mathsf{ref}.\tau}{\sigma \vdash \alpha : \mathsf{ref}.\tau}\, \sigma(\alpha) = \alpha'$$

$$\frac{}{\sigma \vdash \alpha : m.\mathsf{int}}\, \sigma(\alpha) \in \mathbb{Z} \qquad \frac{\forall j\, [\sigma \vdash \alpha_j : \mu_{ij}]}{\sigma \vdash \alpha : m.\sum_i f_i(\mu_{ij})_j}\, \sigma(\alpha) = f_i(\alpha_1, ..., \alpha_{|f_i|})$$

The word "co-inductive" in the definition intuitively means that we allow proof "trees" made with the inference system to be infinitely tall.[3] This technical trick is necessary for being able to assign types to circular terms. It also allows the following nice characterisation of the meaning of types:

[2] We borrowed the variable letters μ and τ from region-annotated types, where there already is a tradition of letting a $\mu = (\tau, \rho)$ stand for a "type and place"—where the "type" component τ can include other μs as component types.

We feel that our mode annotations are structurally similar enough to warrant a similar notation. We use an infix dot rather than a conventional pair notation, however, because we feel it would be visually confusing to add yet another set of brackets to the type syntax.

Finally, the m comes before the τ because we feel that expressions like "$m.\sum_i f_i(\mu_{ij})_j$" are easier to comprehend than "$\sum_i f_i(\mu_{ij})_j.m$" would be.

[3] More formally, the meaning of the definition is that $(\vdash:)$ is the largest (under normal set inclusion) set of triples from $\langle \mathit{Store} \rangle \times \langle \mathit{Addr} \rangle \times \mathbb{T}$ which has the property that every triples in the set can be derived in at least one way from triples in the set (it is even allowed to derive a triple from itself). A proof that such a largest set always exists may be found

Definition 7.2 *A **store typing** ξ is a finite mapping from addresses to sets of types.*

Theorem 7.3 $\sigma_0 \vdash \alpha_0 : \mu_0$ *if and only if there is a store typing* ξ *with* $\mathrm{Dom}\,\xi = \mathrm{Dom}\,\sigma_0$ *such that for all* $\alpha \in \mathrm{Dom}\,\sigma_0$ *and* $\mu \in \xi(\alpha)$*,*

ST1. If $\sigma_0(\alpha) = \mathsf{uninst}$ *then* $\mu = \mathsf{ref}.\tau$ *for some* τ*.*

ST2. If $\sigma_0(\alpha) = \alpha'$ *then* $\mu = \mathsf{ref}.\tau$ *for some* τ*, and* $\mu \in \xi(\alpha')$*.*

ST3. If $\sigma_0(\alpha) \in \mathbb{Z}$ *then* $\mu = m.\mathsf{int}$ *for some* m*.*

ST4. If $\sigma_0(\alpha) = f(\alpha_1, \ldots, \alpha_{|f|})$ *with* $f \notin \mathbb{Z}$*, then* $\mu = m.\sum_i f_i(\mu_{ij})_j$ *and there is an* i *such that* $f = f_i$ *and* $\mu_{ij} \in \xi(\alpha_j)$ *for* $1 \le j \le |f|$*.*

and $\mu_0 \in \xi(\alpha_0)$

Proof.[4] ξ can be seen as encoding a set of judgements of the form $\sigma_0 \vdash \alpha : \mu$. The conditions ST1–ST4 are really just a paraphrase of the inference system in Definition 7.1: Essentially they require that for every judgement "in" ξ the premises of the applicable inference rule must also be "in" ξ.

If ξ is given, a (possibly infinite) proof tree for $\sigma_0 \vdash \alpha_0 : \mu_0$ can be constructed bottom-up using only judgements "in" ξ, because there are always enough judgements "in" ξ to add an extra layer to the proof tree.

Conversely, if a (possibly infinite) proof of $\sigma_0 \vdash \alpha_0 : \mu_0$ is given, the set of judgements in the proof tree can be encoded as a ξ which will then automatically meet the conditions ST1–ST4. □

The importance of Theorem 7.3, apart from characterising the meaning of TP types in a way that many readers may find more intuitive than co-inductive definitions, is that a store typing encodes not only *facts* ("this type describes those terms") but *proofs* of facts. Thus the store typing allows us to reason about not only *that* certain types describe the values manipulated by a program but also *how* they do so.

It is important that $\xi(\alpha)$ is a *set* of types rather than a single type. Otherwise it would not be possible to find a store typing that certified

$$\sigma \vdash \alpha_1 : \mathsf{bare}.f(\mathsf{bare}.\mathsf{int}, \mathsf{ref}.\mathsf{int})$$

where $\sigma(\alpha) = \mathbf{case}\ \alpha\ \mathbf{of} \begin{cases} \alpha_1 & \mapsto & f(\alpha_2, \alpha_2) \\ \alpha_2 & \mapsto & 42 \end{cases}$

7.3 Well-typed P programs

Define a TP **typing** $\mathcal{T}$ by analogy with the TGP definition (Definition 6.2 on page 92).

---

in Pitts [1994], along with a general introduction to co-inductive definitions, or in Aczel [1977, Sections 1.1 and 1.6].

Another example of a co-inductive definition is the definition on $\cong$ in Theorem A.21 of Appendix A where we spell out the mathematical details more throughly than here. To see the connection to Definition 7.1, imagine that Definition A.19 had been phrased in terms of a (one-rule) inference system.

[4]The argument is more immediate if one uses the formal interpretation of the co-inductive Definition 7.1 given by the previous footnote.

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{call}\ pr(x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr \rfloor})\ \&\ b \rightsquigarrow \vec{\mu}}\ \mathcal{T}(pr) = \left\langle \frac{\mathcal{T}(x_1),\ldots,\mathcal{T}(x_{\lceil pr \rceil})}{\mathcal{T}(x'_1),\ldots,\mathcal{T}(x'_{\lfloor pr \rfloor})} \right\rangle$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{builtin}\ pr(x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr \rfloor})\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} \forall i : \mathcal{T}(x_i) = \mathsf{bare.int} \\ \forall i : \mathcal{T}(x'_i) = \mathsf{bare.int} \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{construct}\ f(x_1,\ldots,x_{|f|}) \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f \notin \mathbb{Z} \\ \tau_0 = f(\mathcal{T}(x_1),\ldots,\mathcal{T}(x_{|f|})) + \ldots \\ \mathcal{T}(x') = \mathsf{bare}.\tau_0 \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{construct}\ f() \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f \in \mathbb{Z} \\ \mathcal{T}(x') = \mathsf{bare.int} \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{destruct}\ x\ \mathsf{as}\ f \rightarrow (x'_1,\ldots,x'_{|f|})\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f \notin \mathbb{Z} \\ \tau_0 = f(\mathcal{T}(x'_1),\ldots,\mathcal{T}(x'_{|f|})) + \ldots \\ \mathcal{T}(x) = \mathsf{bare}.\tau_0 \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{destruct}\ x\ \mathsf{as}\ f \rightarrow ()\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f \in \mathbb{Z} \\ \mathcal{T}(x) = \mathsf{bare.int} \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{makevar} \rightarrow x'\ \ \&\ b \rightsquigarrow \vec{\mu}}\ \mathcal{T}(x') = \mathsf{ref}.\tau$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{deref}\ x \rightarrow x'\ \ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} \mathcal{T}(x) = \mathsf{ref}.\tau \\ \mathcal{T}(x') = \mathsf{bare}.\tau \end{cases}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{alias}\ x \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}}\ \mathcal{T}(x) \sqsubseteq \mathcal{T}(x')$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{unify}\ x_1{=}x_2\ \&\ b \rightsquigarrow \vec{\mu}}\ \mathcal{T}(x_1) = \mathcal{T}(x_2) \qquad \frac{\mathcal{T} \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T} \vdash \mathsf{cut}\ \&\ b \rightsquigarrow \vec{\mu}}$$

$$\frac{}{\mathcal{T} \vdash \mathsf{fail} \rightsquigarrow \vec{\mu}} \qquad \frac{}{\mathcal{T} \vdash \mathsf{exit}(x_1,\ldots,x_n) \rightsquigarrow (\mathcal{T}(x_1),\ldots,\mathcal{T}(x_n))}$$

$$\frac{\mathcal{T} \vdash b \rightsquigarrow (\mu'_1,\ldots,\mu'_{\lfloor pr \rfloor})}{\mathcal{T} \vdash pr\ \mathsf{:-}\ \mathsf{enter} \rightarrow (x_1,\ldots,x_{\lceil pr \rceil})\ \&\ b.}\ \mathcal{T}(pr) = \left\langle \frac{\mathcal{T}(x_1),\ldots,\mathcal{T}(x_{\lceil pr \rceil})}{\mu'_1,\ldots,\mu'_{\lfloor pr \rfloor}} \right\rangle$$

$$\frac{\mathcal{T} \vdash pr\ \mathsf{:-}\ c_1. \quad \cdots \quad \mathcal{T} \vdash pr\ \mathsf{:-}\ c_n.}{\mathcal{T} \vdash pr\ \mathsf{:-}\ c_1;\ldots;c_n.}$$

Figure 7.1: Typing rules for P programs. Compare with Figure 6.1 on page 93. The "$\sqsubseteq$" relation in the side condition for the alias *rule will be defined in Section 7.4.*

A TP **well-typing** is one that complies with the rules in Figure 7.1. Most of the rules are completely equivalent to the TGP rules in Figure 6.1, except that builtin, construct and destruct require that certain types have mode bare rather than ref. (That destruct requires the destructed value to have mode bare will eventually imply that the destruct instruction cannot go wrong_3, which is the primary purpose of the TP system).

The rules for makevar and deref are not surprising either.

The surprise is in the rule for alias which allows the type of a value to change when the supposedly do-nothing instruction gives it a new name. In the following section we discuss why this **subtyping** feature is necessary and what it means precisely.

7.4 Subtyping in TP

The intuitive motivation behind the subtyping in TP is this. We want to be able to use the TP type system to show that code such as

```
(...)
  & construct 17( ) → a
  & destruct a as 17 → ( )
  & makevar → b
  & unify a = b
(...)
```

does not go wrong_3—that is, that the operand a to destruct is not a variable, which is clear in this case because is has just been constructed. However, the unify instruction requires the types of a and b to be identical, and b's type must be ref.τ for some τ. Thus the common type of a and b must be ref.int which does not offer any guarantee that the destruct does not go wrong_3.

The alias instruction is designed to help solve this problem. With an alias thrown in, the code

```
(...)
  & construct 17( ) → a
  & alias a → a2
  & destruct a as 17 → ( )
  & makevar → b
  & unify a2 = b
(...)
```

can be made to work with the type system. The type of a can be bare.int while the types of a2 and b are both ref.int.

After the TP type checking has succeeded it is safe to eliminate the alias instruction. The resulting P program will not be TP well typed, but we still know it will not go wrong_3, because alias elimination is a local optimization that clearly does not make a program go wrong_3 unless it already did before.

Thus the alias instruction is really just a technical trick that allows us to isolate the knowledge that any bare.int value can also be ref.int value to one place in the type system. We feel that this makes the presentation of the type system as well as proofs about it simpler.

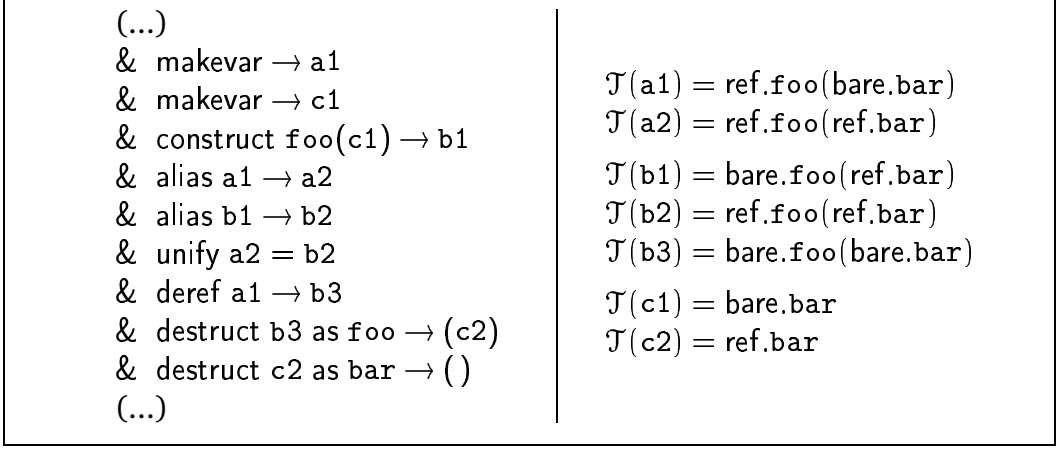


Figure 7.2: A piece of code which goes wrong_3 *even though it would be TP correct if the first definition of* $\sqsubseteq$ *was used. The registers* a1 *and* a2 *are both bound to the same actual cell in the store; likewise all three* bn*'s (and both* cn*'s) refers to the same cell.*

Now, of course, the challenge is to find a general and safe way of deriving facts such as "any bare.int value can also be a ref.int value".

We use the notation bare.int $\sqsubseteq$ ref.int for such facts. The intuition behind the notation is that $\mu_1 \sqsubseteq \mu_2$ requires that μ_2 describes a larger set of (σ, α) pairs than μ_1 does. (However, we shall see that this condition is not sufficient).

7.4.1 First attempt (doesn't work)

An intuitive first attempt at defining $\sqsubseteq$ would be to define that

$$\mu_1 \sqsubseteq \mu_2 \quad \text{iff} \quad \forall \sigma, \alpha : \ \sigma \vdash \alpha : \mu_1 \Longrightarrow \sigma \vdash \alpha : \mu_2$$

which corresponds to requiring the alias instruction to preserve the invariant that the value of each live register matches the type of the register.

However, this definition does *not* work, as witnessed by the code in Figure 7.2. The code is apparently well-typed according to the proposed definition of $\sqsubseteq$: Any term that is described by ref.foo(bare.bar) or bare.foo(ref.bar) is also described by ref.foo(ref.bar). Yet the code manages to point c2 (whose type is bare.τ) to an uninstantiated variable which makes the second destruct instruction go wrong_3.

A closer inspection of the example shows that the invariant still holds after the alias instruction, but the unify instruction causes it to break. Interestingly, the register for which the invariant breaks is a1—which is not used at all in the unify instruction! However, the side effect of the unify causes the value of a1 to change from X (which *is* described by a1's type) to foo(Y) (which is not).

The real problem is that the uninstantiated variable bound to a1 and a2 is known by two different types. A unify instruction that knows only one of

the types may then instantiate it to a value which is described by that one type but not the other.

Similar problems are not uncommon for updateable locations in connection with advanced type systems. The cure is invariably to make sure that no updateable location is ever known by two different types. It can be surprisingly complicated to do that without losing the benefits of the advanced type system (see, *e.g.*, Leroy [1992]). Fortunately this case does not turn out that bad.

7.4.2 Second attempt

We replace the previous invariant with a new one: At each point in the execution of the program there is a single store typing ξ such that for all the relevant environments $\mathcal{E}$ and variables x, $\mathcal{T}(x) \in \xi(\mathcal{E}(x))$ *and* for all addresses α with $\sigma(\alpha) = \mathsf{uninst}$, $\xi(\alpha)$ has only one element.

We now define a $\sqsubseteq$ which makes the alias instruction preserve this invariant:

Definition 7.4 *The relation $\sqsubseteq$ is defined co-inductively by the following inference system:*

$$\frac{}{\mathsf{int} \preceq \mathsf{int}} \qquad \frac{\forall i, j : \quad \mu_{ij} \sqsubseteq \mu'_{ij}}{\sum_i f_i(\mu_{ij})_j \preceq \sum_i f_i(\mu'_{ij})_j} \qquad \frac{}{\mathsf{ref}.\tau \sqsubseteq \mathsf{ref}.\tau} \qquad \frac{\tau \preceq \tau'}{\mathsf{bare}.\tau \sqsubseteq m'.\tau'}$$

where, for compactness, judgements of the form "$\tau \preceq \tau$" are used as well as "$\mu \sqsubseteq \mu$" are used.

In this definition co-induction is necessary because types can be recursive. Despite the co-induction it is decidable whether $\mu_1 \sqsubseteq \mu_2$ for given μ_1 and μ_2; this can be proven analogously to Corollary A.28.

Intuitively the definition means that a bare in μ_1 can be changed to a ref in μ_2, as long as there are no refs above the original bare in μ_1. Thus, for example,

$$\begin{aligned} \mathsf{bare}.f(\mathsf{bare}.f(\mathsf{bare}.f(\mathsf{bare}.\mathsf{int}))) &\sqsubseteq \mathsf{ref}.f(\mathsf{ref}.f(\mathsf{bare}.f(\mathsf{ref}.\mathsf{int}))) \\ \mathsf{ref}.f(\mathsf{ref}.f(\mathsf{bare}.f(\mathsf{bare}.\mathsf{int}))) &\not\sqsubseteq \mathsf{ref}.f(\mathsf{ref}.f(\mathsf{bare}.f(\mathsf{ref}.\mathsf{int}))) \\ \mathsf{bare}.f(\mathsf{ref}.f(\mathsf{bare}.f(\mathsf{ref}.\mathsf{int}))) &\sqsubseteq \mathsf{ref}.f(\mathsf{ref}.f(\mathsf{bare}.f(\mathsf{ref}.\mathsf{int}))) \end{aligned}$$

We will prove that this definition does cause the invariant to be preserved. We also claim that it is the most liberal definition that does this, but we have no proof of that.

It is easy to see that the $\sqsubseteq$ thus defined is reflexive and transitive, hence a partial order.

7.5 Well-typed P programs do not go wrong_3

We can now prove that the TP type system fulfils its purpose:

Theorem 7.5 *When a TP program is executed according to the semantics in Section 3.7, the error states* wrong_2 *and* wrong_3 *never arise.*

The theorem follows from this induction lemma:

Lemma 7.6 *When a TP program with well-typing* $\mathcal{T}$ *is executed according to the semantics in Section 3.7, for every* ⟨*Frame*⟩ $\mathfrak{fr} = (\sigma, \mathcal{E}, \mathfrak{b}, \delta, \kappa)$ *that appears during the computation, there is a store typing* ξ *such that*

F1. ξ *"matches"* σ *in the way described by conditions ST1–ST4 of Theorem 7.3 (page 98).*

F2. $\forall \alpha : \quad \sigma(\alpha) = \mathsf{uninst} \implies \#(\xi(\alpha)) \leq 1$

F3. $\forall x \in \mathrm{Dom}\,\mathcal{E} : \quad \mathcal{T}(x) \in \xi(\mathcal{E}(x))$

F4. Condition F3 also holds for the $\mathcal{E}$ *in each of the elements of* κ.

Proof. By induction on the length of the computation. The structure of the induction steps parallels the definition of the transition relation in Section 3.7.2, working by case analysis on the next instruction to be executed.

In most of the cases the required properties of the new state are easily established. The previous store typings can be used unchanged or with the addition of the obvious typings for newly allocated cells. In the exit case an auxiliary lemma analogous to Lemma 6.6 is necessary.

We omit the detailed arguments except for the unify and alias instructions which are where the interesting things happen.

For the unify instruction we assume that the unification succeeds (if it fails, the frames in the new state all appeared in the old state, so the induction hypothesis directly contains what we need to prove). Our aim is to show that the ξ we get from the induction hypothesis can be reused unchanged even though the store changes. With the terminology of Definitions 3.9ff, we can prove by induction on the number of $\triangleright\!\!-$ steps that $(\sigma_0, \{(\alpha_1, \alpha_2)\}) \triangleright\!\!-^* (\sigma, \mathcal{C})$ implies that

- Conditions F1 and F2 hold for ξ in σ as well as in σ_0.
- For each $(\alpha_1', \alpha_2') \in \mathcal{C}$, $\xi(\alpha_1')$ and $\xi(\alpha_2')$ has at least one common element.

For zero $\triangleright\!\!-$ steps this is immediate from the induction hypothesis in the "outer" induction: $\xi(\alpha_1)$ and $\xi(\alpha_2)$ certainly has a common element, namely the (common) type of the two registers being unified.

Then there is an induction step for each of the cases (a)–(d) in Definition 3.9. Cases (a) through (c) follow directly from the conditions on ξ; the interesting case is (d). Here it is crucial that we know that $\xi(\alpha_1)$ is a singleton set whose element μ is in $\xi(\alpha_2)$. This means that ξ still meets condition ST2 after we change $\sigma(\alpha_1)$ from uninst to α_2.

This completes the case for the unify instruction. Now we turn to the alias instruction. This is in some sense its dual: the store does not change, but we

have to show that the store typing can be extended such that the invariant is preserved.

More precisely, we already know a Ξ that meets conditions F1 through F4 and for which $\mu_1 = \mathcal{T}(x) \in \Xi(\mathcal{E}(x))$; we are going to construct a Ξ' such that $\Xi(\alpha) \subseteq \Xi'(\alpha)$ for all α and $\mu_2 = \mathcal{T}(x') \in \Xi(\mathcal{E}(x))$. Because the alias instruction is type-correct we know that $\mu_1 \sqsubseteq \mu_2$.

Now define $A_0 = \{(\mathcal{E}(x), \mu_1, \mu_2)\}$,

$$A_n = \left\{ (\alpha_j, \mu_{ij}, \mu'_{ij}) \;\middle|\; \begin{array}{l} (\alpha, m.\sum_i f_i(\mu_{ij})_j, m'.\sum_i f_i(\mu'_{ij})_j) \in A_{n-1}, \\ \sigma(\alpha) = f_i(\alpha_1, \ldots, \alpha_{|f_i|}), \\ 1 \le j \le |f_i| \end{array} \right\}$$
$$\cup \left\{ (\alpha', \mu, \mu') \;\middle|\; (\alpha, \mu, \mu') \in A_{n-1}, \;\; \sigma(\alpha) = \alpha' \right\}$$

and $A = \bigcup_{n=0}^{\infty} A_n$. Then, by induction on n, $(\alpha, \mu, \mu') \in A$ implies $\mu \in \Xi(\alpha)$ and $\mu \sqsubseteq \mu'$

Define

$$\Xi'(\alpha) = \Xi(\alpha) \cup \{ \mu' \mid (\alpha, \mu, \mu') \in A \}$$

We now have to prove that this Ξ' meets conditions F1–F4. F1, F3, and F4 follow easily from the just mentioned properties of Ξ and A. For F2, let an α with $\sigma(\alpha) = \mathsf{uninst}$ be given. We then know that $\Xi(\alpha) = \{\mathsf{ref}.\tau\}$ for some τ. Then, for any $(\alpha, \mu, \mu') \in A$, μ must be $\mathsf{ref}.\tau$. Because $\mu = \mathsf{ref}.\tau \sqsubseteq \mu'$ it must be that $\mu = \mu'$, thus $\Xi'(\alpha)$ is a singleton set as required. □

Chapter 8

Translating Prolog into TP

In this chapter we describe how to translate Prolog into (TP-typeable) P, thus substantiating our implicit claim that P has anything to do with Prolog at all.

The main problems to be solved in this translation are

- P has explicit parameter modes that classify the parameters to each predicate as input and output parameters; Prolog does not distinguish syntactically between these categories.
- P has specialized instructions for the creation, analysis, and unification of terms; Prolog has a single language construct that are used for all three activities.

Both of these can in principle be solved with "brute force and negligence of quality" by letting every parameter be an input parameter and translating every Prolog goal to a sequence of makevar and construct instructions followed by a unify or call. The resulting P code would, however, be very badly suited for reasoning about memory usage, so we want to describe how a translator can use P's features with considerably more finesse.

The translation method we describe is the one we use in our prototype region-based Prolog implementation [Makholm 2000]. Other methods may well be imagined; we selected on this one primarily because it was easy to invent and implement, yet does not produce prohibitively bad P code.

Because the exact method of translation is in principle irrelevant to the region-based techniques that are the main focus of this thesis, and because none of the techniques we employ are novel, we describe the workings of the individual steps in the translation only briefly. Our goal is to argue that one *can* translate (our "core" subset of) Prolog to P, not to show in detail *how* to do it.

The structure of the chapter corresponds to the organisation of the Prolog translator `pro2P` in our prototype implementation [Makholm 2000]. The translation works in several phases; when `pro2P` is invoked with the switches `-v` and `-d` it announces the individual phases and dumps the result of each to auxiliary files for inspection by curious users (see Figure 8.1).

```
embla:~/2del/speciale/impl$ ./pro2P -d -v 8queens.pro
Reading 8queens.pro
Dumping to 8queens.dump.a0
Doing disjunction unfolding
Dumping to 8queens.dump.a1
Doing parameter moding
Dumping to 8queens.dump.a2
Doing number wrapping
Dumping to 8queens.dump.a3
Doing normalization
Dumping to 8queens.dump.a4
Doing dead-code elimination
Checking variable scopes
Dumping to 8queens.dump.a5
Doing singleton elimination
Dumping to 8queens.dump.a6
Doing normalization
Dumping to 8queens.dump.a7
Doing match-or-build translation
8queens.pro(14) warning: plus/3 may fail due to unbound arguments
8queens.pro(15) warning: plus/3 may fail due to unbound arguments
8queens.pro(30) warning: plus/3 may fail due to unbound arguments
Dumping to 8queens.dump.a8
Doing cut construction
Checking output program
Writing to 8queens.P
embla:~/2del/speciale/impl$
```

Figure 8.1: A sample run of the `pro2P` *translator. The "Doing..." lines announce transformation phases starting. The "Checking..." lines are internal consistency checks which are not mentioned in the text.*

8.1 Parsing and unfolding of syntactic sugar

The front end of the translator expresses the input program as a series of pure Horn clauses with cut, *i.e.*, with the abstract syntax shown in Figure 8.2. The main goal of the program is, implicitly,

```
?- main(), fail.
```

The input programs can be written in this syntax (with the standard lexical conventions for distinguishing between variables and functors/predicates). We also support a number of convenient variant forms ("syntactic sugar") which the parser translates into canonical form:

- *Nullary functors* (also known as *atoms*) can be written without parentheses.
- *Lists* in Prolog's normal notations (such as "`[1,2,3]`", "`[1,2|[3]]`", or "`.(1,.(2,.(3,[])))`"). All notations are converted to terms built

$$
\begin{array}{rcl}
\langle program \rangle & ::= & \langle clause \rangle ... \langle clause \rangle \\
\langle clause \rangle & ::= & \langle atomic \rangle \texttt{ :- } \langle goal \rangle, ..., \langle goal \rangle. \\
\langle goal \rangle & ::= & \langle atomic \rangle \\
& | & \langle term \rangle = \langle term \rangle \\
& | & ! \\
& | & \mathsf{fail} \\
\langle atomic \rangle & ::= & \mathtt{pr}(\langle term \rangle_1, ..., \langle term \rangle_n) \\
\langle term \rangle & ::= & \mathtt{f}(\langle term \rangle_1, ..., \langle term \rangle_{|\mathtt{f}|}) \\
& | & ... \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid ... \\
& | & \mathtt{x}
\end{array}
$$

Figure 8.2: The "core" Prolog subset we work with. The output of the parsing phase is an abstract syntax tree built from this syntax. Throughout the implementation, a "$\mathtt{f}$*" is represented as an alphanumeric identifier with a "*$/n$*" suffix, and "*$\mathtt{pr}$*" and "*$\mathtt{x}$*" are simply alphanumeric identifiers. Extending our formal definition of P, the implementation always allows the same "*$\mathtt{x}$*" identifier to be reused in different clauses. In this step the same "*$\mathtt{pr}$*" identifier can even be used with different arities.*

from the functors `Cons/2` and `Nil/0` which cannot be input otherwise because they lexically ought to be variable names.

- *Anonymous variables*, "_", are given a name guaranteed not to be used anywhere else.
- *Edinburgh strings* in "double quotes" stand for lists of character codes.
- Certain *infix functors*, selected by the principle that we add them to the parser when they occur in legacy code we try to use the prototype on.
- Limited support for `is/2`. Under the assumption that all variables in the goal are numbers, an `is` goal is translated to a series of calls to our arithmetic primitives.
- *Negation-as-failure*, spelled "\+". This is implemented by generating an auxiliary predicate with a cut in it.
- *Disequalities*. "$\langle term \rangle_1$ `\=` $\langle term \rangle_2$" is implemented by applying negation-as-failure to the primitive "=" goal.
- *Local disjunctions, i.e.*, goals of the form

$$(\langle goal \rangle, ..., \langle goal \rangle; \langle goal \rangle, ... \langle goal \rangle)$$

Also implemented by generating an auxiliary predicate. Cuts inside the disjunction are not allowed[1].

[1] In Standard Prolog such cuts cut all the way back to the choice point of the *clause*, not

- *If-then-else,* spelled "(⟨*cond* − *goals*⟩ -> ⟨*then* − *goals*⟩; ⟨*else* − *goals*⟩)". Implemented by generating an auxiliary predicate of the form

$$\begin{array}{lll} A & :- & \langle cond - goals \rangle, \; !, \; \langle then - goals \rangle. \\ A & :- & \langle else - goals \rangle. \end{array}$$

The unfolding of syntactic sugar takes place in the parser. The constructions that need auxiliary predicates leave the code for the auxiliary predicate in a special construction at the place of the call; a secondary "disjunction unfolding" phase decides on a predicate name and moves the code to the top level of the program.

The dumps *name*`.dump.a0` and *name*`.dump.a1` show the internal representation of the program before and after the "disjunction unfolding" phase.

8.2 Parameter moding

The parameter moding phase divides the parameters of each predicate into input and output parameters. It also converts the program from being a series of Prolog-like clauses to being a series of P-like predicate definitions, except that unify instructions may unify arbitrary terms, not just registers.

The properties of the new program are:

- A predicate name in the new program is a pair consisting of a predicate name pr from the old program and a sequence of "`I`"s and "`O`"s that encode the parameter modes.
- The predicate $(\texttt{main}, \varepsilon)$ (where ε is the empty sequence) needs to be defined in the new program.
- For each predicate that needs to be defined in the new program, a definition is obtained in the following way, which may cause other predicates to need to be defined.

  Imagine the predicate to be defined is $(\texttt{foo}, \texttt{IOIO})$. Then, find each clause

$$\texttt{foo}(t_1, t_2, t_3, t_4) \; :- \; \langle goals \rangle.$$

  from the old program where the predicate and the number of parameters matches.[2] Each of these becomes a clause for $(\texttt{foo}, \texttt{IOIO})$ in the "new" program:

$$\begin{array}{lll} (\texttt{foo}, \texttt{IOIO}) \; :- & \multicolumn{2}{l}{\text{enter} \rightarrow (x_1,x_3) \;\&\; \text{unify } x_1{=}t_1 \;\&\; \text{unify } x_3{=}t_3} \\ & \&\; \langle goals \rangle & \\ & \&\; \text{unify } x_2{=}t_2 \;\&\; \text{unify } x_4{=}t_4 \;\&\; \text{exit}(x_2,x_4) & \end{array}$$

  where x_1 through x_4 are fresh register names and the goals in ⟨*goals*⟩ get translated as follows:

---

just the choice point of the local disjunction. Supporting that would require (small) changes to P.

[2] If there are no such clauses at all, we generate the definition "$(\texttt{foo}, \texttt{IOIO})$:- enter $\rightarrow$ (x_1,x_3) & fail." and emit a warning to the user.

- A "$\langle term \rangle = \langle term \rangle$" goal becomes a (pseudo) unify instruction.
- A "!" goal becomes a cut instruction.
- A "fail" goal becomes a fail in P which causes the rest of the definition to disappear.
- A "$pr'(t_1,...,t_n)$" goal is transformed to either a call or a builtin instruction in the following way.

  First, each t_i in the parameter list which is either a compound term[3] or a register that also appears somewhere else in the parameter list, replace t_i with a fresh register x and insert a unify $x=t_i$ instruction before the present goal.

  After this substitution the parameters are all distinct[4] registers. We classify those that have been mentioned before in the definition (which includes the newly-created ones that replaced compound terms) as input parameters and the rest as output parameters.

  If the predicate name pr' and the distribution of input and output parameters match one of the built-in predicates known by the implementation, the goal now becomes a builtin instruction.

  Otherwise it becomes a call instruction with the name of the called predicate derived from pr' and the input–output division. It needs to be defined somewhere in the new program.

Clearly, the process of generating the "new" program is well-defined (the generation of one new predicate is not affected by which other new predicates exist) and terminating (each $\langle atomic \rangle$ with n parameters in the program can cause at most 2^n different new predicates to be generated).

Our implementation uses a simple depth-first algorithm with a global cache of already-defined new predicates to implement this transformation.

After the transformation is complete, we generate new alphanumeric names for all of the new predicates. Our implementation tries to reuse as many of the original names as possible.

The dump *name*`.dump.a2` shows the program after the parameter modding phase.

8.3 Number wrapping

The number wrapping phase transforms the program such that the only numbers that appear in the data it manipulates are operands to a special functor `N/1` which we reserve for that purpose (we can be sure it does not appear in the program before, because the parser would classify it as a variable name).

Two simultaneous transformations are involved in this:

- Every $\langle term \rangle$ of the form $c \in \mathbb{Z}$ is replaced with the $\langle term \rangle$ `N/1`(c).

[3] We take "compound terms" to include number constants and unary functors.

[4] See Box 8.1 for why this is important.

Box 8.1—THE IMPORTANCE OF USING DISTINCT REGISTERS IN CALLS

It is important that the parameter moding makes sure that the parameters to a predicate call are all *distinct* variables before it selects parameter modes.

To see why, consider this Prolog program:

```
twelve(5, 7) :- !.
twelve(6, 6).
main :- twelve(X, X).
```

Here we see that X is unbound before the call to `twelve`, so it might be tempting to classify the parameters as output and try to unify them only after the call. However, in that case the first clause of `twelve` would *succeed* in binding one of its parameters to 5 and another to 7, and it would cut the other clause away before it can determined that that decision was wrong. Thus the semantics of the resulting P program would not be faithful to the semantics of the original Prolog program.

In our solution, the body of `main` gets rewritten to

```
X = Y, twelve(X, Y)
```

before the parameters are classified. Then both parameters become input parameters, and the P version eventually becomes

```
main :- enter → ( )
        & makevar → X
        & call twelve (X, X) → ( )
        & exit ( ).
```

which means that the first clause of `twelve` will be trying to unify the same variable with 5 and 7 at once and (correctly) fail before it cuts the other possibility away.

- Every "builtin $pr(x_1,\ldots,x_{\lceil pr \rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr \rfloor})$" instruction is replaced by

  ```
  (...)
    & unify x1 = N/1(x̃1)
    ⋮
    & unify x⌈pr⌉ = N/1(x̃⌈pr⌉)
    & builtin pr(x̃1,...,x̃⌈pr⌉) → (x̃'1,...,x̃'⌊pr⌋)
    & unify x'1 = N/1(x̃'1)
    ⋮
    & unify x'⌈pr⌉ = N/1(x̃'⌊pr⌋)
  (...)
  ```

  where the $\tilde{x}_i$s and $\tilde{x}'_i$s are fresh variables.

This makes sure that the program can be typed, even though our type systems do not allow numbers and other compound terms to be described by the same type. See Box 6.1 on page 91 for a further discussion. Most of the `N/1` functors disappear again in the singleton elimination phase (Section 8.6).

The dump *name*`.dump.a3` shows the program after the number wrapping phase.

8.4 Unification normalization

What now prevents the program from being a P program is that the unify instructions in it equate arbitrary terms, not just known registers.

The unification normalization transforms the unify instructions such that only two forms remain:

- Ordinary P unification between two known registers—that is, registers that have been mentioned earlier in the clause
- Unification of a known register with a compound term

Eventually the latter kind will be transformed into sequences that involve destruct instructions, but it is convenient to have done away with other forms of unification before tackling that.

The normalization phase transforms the unify instructions in each clause from left to right. The different cases that can be met are:

1. *known register* versus *known register*. If the two registers are the same, then the instruction can be discarded. Otherwise leave it unchanged.
2. *known register* versus *unknown registers*. At run time, the two registers will always have precisely the same value, so we can eliminate the instruction by substituting one of the registers for the other in the entire clause. Our prototype prefers to use a register whose name comes from the original Prolog program as the "surviving" one.
3. *known register* versus *compound term*. Leave the instruction unchanged for now (except normalizing the order of the operand such that the register is always the first operand).
4. *unknown register* versus *unknown register*. The main example of this case is the situation described in Box 8.1. Insert a "makevar $\rightarrow x$" instruction before the unify, where x is one of the registers. This causes the unify instruction to fall under case 2 or (if the two unknown registers were identical) case 1. Proceed from there.
5. *unknown register* versus *compound term*. First, insert makevar instructions before the instruction for every unknown register found in the compound term.

   If that makes the other operand known (such as in the goal "$X = f(X)$") continue as for case 3.

   Otherwise, replace the instruction with a series of construct instructions that build the requested term.
6. *compound term* versus *compound term*. If the root functors of the two terms are different, the instruction can never succeed. Replace it with fail (which causes the rest of the clause to disappear).

   Otherwise, replace the instruction by a unify instruction for each matching pair of subterms, and recursively normalize those new instructions.

The dump *name*`.dump.a4` shows the program after the unification normalization phase.

Box 8.2—SOUNDNESS OF SINGLETON ELIMINATION

It is easy to see that the singleton elimination transformation preserves the property of typeability, but it may still appear that it is not semantically sound, because it destroys the difference between an uninstantiated variable and a needless functor whose operand is an uninstantiated variable.

It is correct that this difference is destroyed, but it does not matter that is is. The only way for the original program to know the difference between X and `f(X)` is to try to unify the value with something that is not `f/1`. And if one does indeed do that, `f` will not be needless.

This argument is spoiled if the implementation is extended to support the meta-logical predicates `var/1` and `nonvar/1` directly. One way of saving it would be to augment type nodes with a flag that told whether terms of that types are used as argument to `var/1` or `nonvar/1`. Types where the flag was set would not be considered singleton types.

8.5 Dead-code elimination

The dead-code elimination phase transforms every

$$\mathsf{call}\ pr(x_1,\ldots,x_{\lceil pr\rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr\rfloor})$$

to

$$\mathsf{call}\ pr(x_1,\ldots,x_{\lceil pr\rceil}) \rightarrow (\,)\ \&\ \mathsf{fail}$$

whenever pr is a predicate that can never succeed because all of its clauses ends in fail.

The purpose of the transformation is primarily technical. Our internal P representation does not store the arities of predicates explicitly. The type inference code therefore expects to be able to find $\lceil pr\rceil$ and $\lfloor pr\rfloor$ by looking at the enter and exit pseudoinstructions in pr's definition. The dead-code elimination makes sure that $\lfloor pr\rfloor$ has the predictable value 0 in the cases where there *are* no exit pseudoinstructions to look at.

The dump *name*`.dump.a5` shows the program after the dead-code elimination phase.

8.6 Singleton elimination

The singleton elimination phase removes needless unary functors from the program. The main example of a needless unary functor is the `N/1` functors inserted by the number wrapping phase (Section 8.3). These are not always needless, but for well-behaved programs they usually are.

The singleton elimination phase works by doing type inference on the program, using a type system that looks like TGP except that (nonsurprising) typing rules have been added for the makevar instruction and the extended form of the unify instruction that is still in use at this point.

After the type inference, a functor is considered **needless** if its type has degenerated from "$\sum_i f_i(\tau_{ij})_j$" to "$f_1(\tau_{11})$". The singleton elimination consists of removing needless functors from the ⟨*term*⟩s in (pseudo) unify instructions and changing construct or destruct instructions that build or inspect them to alias instructions. (See Box 8.2 for a discussion of the safety of doing that).

Type inference for the TGP-like system is easy and can be done by conventional means. There are only two significant difference from a completely standard unification-based monomorphic type inference.

The first difference is that when we need to unify, *e.g.*, "$f_1(\tau_{11}, ..., \tau_{1|f_1|}) + f_2(\tau_{21}, ..., \tau_{2|f_2|}) + ...$" with "$f_1(\tau_{11}, ..., \tau_{1|f_1|}) + f_3(\tau_{31}, ..., \tau_{3|f_3|}) + ...$", the result must be "$f_1(\tau_{11}, ..., \tau_{1|f_1|}) + f_2(\tau_{21}, ..., \tau_{2|f_2|}) + f_3(\tau_{31}, ..., \tau_{3|f_3|}) + ...$", instead of a unification error. The same thing happens when inferring record types for SML.

The second difference is that we need to be able to infer recursive types. The only thing that is necessary for that is to do type unification without an occurs check (and to use a unification algorithm which does not risk nontermination in the presence of circular terms, of course).

The dump *name*`.dump.a6` shows the program after the dead-code elimination phase.

Our prototype throws away the type information before the dump operation. This is because it was easier to to add another call to the type inference engine in the match-or-build translation phase than to adjust the type data for the effects of the second normalization phase that comes before.

8.7 Second normalization

The singleton elimination is followed by a second normalization pass which does the same as the normalization described in Section 8.4. Its job here is to eliminate any instructions that have become unnecessary because their only task was to apply or remove a needless functor.

The alias instructions that are created from construct instruction by the singleton elimination are treated like unifications between a known and an unknown register (and therefore all eliminated).

The dump *name*`.dump.a7` shows the program after the second normalization phase.

8.8 Match-or-build translation

The match-or-build translation phase transforms the remaining pseudo unify instructions (those that unify a known register with a compound term) to either sequences of destruct instructions or auxiliary match-or-build predicates akin to the ones on Figure 3.2 (page 39).

The transformation prefers plain destruct instructions, but it has to make sure that they are only used where the TP system of Chapter 7 allows it. Thus the first step is to compute a TP typing of the program.

8.8.1 TP type inference

The TP type inference proceeds in four phases.

First, alias instructions are inserted everywhere in the program, such that each register is used in at most one non-alias instruction.[5] This gives maximal freedom to use TP's subtyping capabilities.

Second, a TP typing with all the modes set to ref is computed. That is completely equivalent to the type inference step of the singleton elimination transformation (Section 8.6).

Third, fresh copies of the type of each register are created, such that the operands of an alias instruction do not share any type nodes. This is necessary for them to be able to have different mode annotations.

Fourth and last, a special analysis determines which of the refs in the types can safely be replaces by bare.

This analysis uses the logic-based technique developed in Makholm [1999, Section 4.1]. It consists of building—at analysis time—a little formal theory[6] about how the given program can be annotated. For each type node in the program, the theory contains a proposition meaning, "this type node is annotated with a ref", such that each truth assignment for all propositions describes a (not necessarily well-) typing for the program. The also contain other auxiliary propositions (to be described in a moment), but the total number of propositions is finite.

A finite number of inference rules and axioms are then derived from the program, such that a truth assignment describes a TP well-typing if and only if it is a **model**, that is, is satisfies the axioms and inference rules. Considering a proposition to be true iff it is provable in the theory (which is decidable because there are only finitely many propositions and rules in the theory) yields a model[7] which clearly is the "least true" model—which again means that it describes the TP well-typing with the fewest ref's.

The generation of inference rules is simple for most of the instructions in

[5] Our prototype immediately optimizes away some of the alias instructions, where it is clear that they are never necessary. That is not essential.

[6] People have different expectations about what concepts the phrase "formal theory" entails. We use the term here in the minimal sense of Mendelson [1997, Section 1.4], where all that is required is some set of "propositions" (which Mendelson calls "well-formed formulas") and a way to deduce propositions from other propositions.

[7] This is true because we do not include implicit connections between the truth values of proposition. Compare, *e.g.*, with propositional logic which requires that the truth assignment of propositions such as $\mathcal{A}$ and $\neg\mathcal{A}$ are related in particular way. In such theories, taking the provable propositions to be true does not necessarily yield a well-formed truth assignment, let alone a model.

the program. For example, the instruction

$$\text{construct } f(x) \rightarrow x' \quad \text{where} \quad \begin{array}{l} \mathcal{T}(x) = m_1.\text{int} \\ \mathcal{T}(x') = m_2.f(m_3.\text{int}) + \ldots \end{array}$$

is only TP well-typed if $m_1 = m_3$, so we generate the inference rules

$$\frac{m_1 \text{ is ref}}{m_3 \text{ is ref}} \qquad \frac{m_3 \text{ is ref}}{m_1 \text{ is ref}}$$

If $\mathcal{T}(x)$ had been a more complex type, we would generate similar pairs of inference rules for each matching pair of type nodes in $\mathcal{T}(x)$ and $\mathcal{T}(x')$. We do not generate any inference rule corresponding to the requirement that $m_2 = \text{bare}$. That is safe because the only other occurrences of x' will be as the right-hand arguments of alias instructions, so nothing can ever force m_2 to be ref.

call instructions result in similar groups of rules being generated to relate the types of the formal and actual parameters.

A makevar instruction result in axiom being generated: the mode annotation of the new variables' topmost type node must be ref.

The unify instruction first is more complicated. Consider, for example[8]

$$\text{unify } x{=}f_1(f_2(x')) \quad \text{where} \quad \begin{array}{l} \mathcal{T}(x_1) = m_1.f_1(m_2.f_2(m_3.f_3(m_4.\text{int}))) \\ \mathcal{T}(x') = m_3'.f_3(m_4'.\text{int}) \end{array}$$

The first task of the pseudo unify instruction is to make sure that the value bound to x has the specified shape. This is not purely an inspection operation; if m_2 or m_1 is ref, it may encounter an uninstantiated variable and have to allocate a new f_2 and perhaps also a f_1 itself. What should the argument of the new f_2 be? If x' is a known register, the value of that can be used, but of x' is unknown, the unify instruction has no other choice than to allocate a new uninstantiated variable to use in the place of the f_3, but in that case m_3 has to be ref, too. Thus, if x' is unknown (which can be determined at analysis time), we generate the rules

$$\frac{m_1 \text{ is ref}}{m_3 \text{ is ref}} \qquad \frac{m_2 \text{ is ref}}{m_3 \text{ is ref}}$$

We also need to make sure that the $\mathcal{T}(x')$ is identical to the appropriate part of $\mathcal{T}(x)$. This applies equally when x' is known (in which case its value must be unified with the appropriate part of x's value) and when x' is unknown (in which case its value *is* the appropriate part of x's value). To that end we generate the rules

$$\frac{m_3' \text{ is ref}}{m_3 \text{ is ref}} \qquad \frac{m_3 \text{ is ref}}{m_3' \text{ is ref}} \qquad \frac{m_4' \text{ is ref}}{m_4 \text{ is ref}} \qquad \frac{m_4 \text{ is ref}}{m_4' \text{ is ref}}$$

[8] Strictly speaking, the types in this example do not occur after the singleton elimination phase. It would only add to the confusion to use more realistic types, however.

We now turn to the alias instruction, for example

$$\text{alias } x \to x' \quad \text{where} \quad \begin{array}{l} \mathcal{T}(x) = \mu_1 = m_1.f_1(m_2.f_2(m_4.\text{int})) \\ \mathcal{T}(x') = \mu_1' = m_1'.f_1(m_2'.f_2(m_4'.\text{int})) \end{array}$$

Here we must generate rules to enforce the $\mu_1 \sqsubseteq \mu_1'$ relation defined in Section 7.4.

Because μ_1 and μ_2 were constructed as fresh copies of the same original type we know that their type nodes line up in pairs: (μ_1, μ_1'), (μ_2, μ_2') and (μ_3, μ_3'). Now, the definition of $\sqsubseteq$ implies that for each pair (μ_i, μ_i') either $\mu_i \sqsubseteq \mu_i'$ or $\mu_i = \mu_i'$ must hold. So we always need the rules.

$$\frac{m_1 \text{ is ref}}{m_1' \text{ is ref}} \qquad \frac{m_2 \text{ is ref}}{m_2' \text{ is ref}} \qquad \frac{m_3 \text{ is ref}}{m_3' \text{ is ref}}$$

However, the reverse implications are only necessary if we must have $\mu_i = \mu_i'$. We do not know in advance whether that will be true, but we can construct propositions and inference rules to have the theory compute it for us. We add auxiliary propositions to the theory with the form "we need $\mu_i = \mu_i'$" and add the rules

$$\frac{m_1 \text{ is ref}}{\text{we need } \mu_1 = \mu_1'} \qquad \frac{m_2 \text{ is ref}}{\text{we need } \mu_2 = \mu_2'} \qquad \frac{m_3 \text{ is ref}}{\text{we need } \mu_3 = \mu_3'}$$

$$\frac{\text{we need } \mu_1 = \mu_1'}{\text{we need } \mu_2 = \mu_2'} \qquad \frac{\text{we need } \mu_2 = \mu_2'}{\text{we need } \mu_3 = \mu_3'}$$

Then we can let m_i' conditionally influence m_i by adding the rules

$$\frac{m_1' \text{ is ref} \quad \text{we need } \mu_2 = \mu_2'}{m_1 \text{ is ref}} \qquad \frac{m_2' \text{ is ref} \quad \text{we need } \mu_2 = \mu_2'}{m_2 \text{ is ref}}$$

$$\frac{m_3' \text{ is ref} \quad \text{we need } \mu_2 = \mu_2'}{m_3 \text{ is ref}}$$

The particular beauty of this scheme is that it works perfectly fine when the types are recursive.

Once the propositions and rules have all been generated it is a simple matter[9] to compute which propositions are provable, starting with the axioms and iteratively adding propositions that are immediate consequences of the already-proved propositions. Then a TP well-typing can be derived from this set of provable propositions.

8.8.2 The actual match-or-build transformation

Once we have a TP well-typing of the program, we can transform instructions of the form

$$\text{unify } x{=}f(t_1,\ldots,t_{|f|})$$

[9] An imperative linear-time algorithm is easily constructed [Makholm 1999].

to appropriate sequences of proper P instructions.

If $\mathcal{T}(x) = \mathsf{bare}.\tau$, then it is safe to simply use a destruct instruction to take x's value apart. Replace the instruction with

```
(...)
  &  destruct x as f → (x̃1,...,x̃|f|)
  &  unify x̃1 = t1
     ⋮
  &  unify x̃|f| = t|f|
(...)
```

where the $\tilde{x}_i$s are fresh registers. Add appropriate types (extracted from $\mathcal{T}(x)$) for the $\tilde{x}_i$s to the TP typing and repeat the transformation for those of the new unify instructions for which t_i is a compound term.

If $\mathcal{T}(x) = \mathsf{ref}.\tau$, things get more complicated. Let $(x_1, ..., x_m)$ be those known registers that appear in the original instruction (including x) and $(x'_1, ..., x'_n)$ be the unknown registers that appear in the original instruction. Replace the instruction with

$$\mathsf{call}\ \tilde{pr}(x_1,...,x_m) \rightarrow (x'_1,...,x'_n)$$

where $\tilde{pr}$ is a new predicate with the definition

```
p̃r :- enter → (x1, ..., xm)
          &  deref x → x̃
          &  cut
             %  this is the "match" case:
          &  unify x̃ = f(t1, ..., t|f|)
          &  exit (x'1, ..., x'n);
       enter → (x1, ..., xm)
             %  this is the "build" case:
          &  unify x̃ = f(t1, ..., t|f|)
          &  alias x̃ → x̃'
          &  unify x = x̃'
          &  exit (x'1, ..., x'n).
```

where $\tilde{x}$ and $\tilde{x}'$ are fresh registers.

Repeat the transformation for the unify instruction in the "match" case. It handles the case where x's value was not an uninstantiated variable; the deref instruction removes the ref from its type so the new unify instruction eventually becomes a destruct sequence.

The "build" case handles the case x's value is an uninstantiated variable. It builds a new term in the fresh register $\tilde{x}$ and uses an ordinary unify instruction to instantiate the variable. The pseudo unify instruction that builds the term is left in the program for now.

8.8.3 Yet another normalization step

After the match-or-build transformation the type information is removed from the program, and the normalization transformation described in Section 8.4 is run for the third time.

This time its main job is to translate the pseudo unify instructions in the "build" cases of the newly generated predicates to appropriate sequences of construct (and possibly makevar) operations.

The normalization step also eliminates the alias instruction that was inserted before the type inference. They are not necessary anymore, because now we are sure that the program cannot go wrong_3.

The kind of pseudo unify instructions generated by the match-or-build transformation can all be reduced completely to standard P instructions by the normalization, so afterwards the program is a correct P program.

The dump *name*`.dump.a8` shows the program after this third normalization phase.

8.9 Cut construction

The cut construction phase attempts to insert "green" cuts[10] in as many of the program's clauses as possible. This is not strictly necessary, but it improves performance of the P program by making the life times of most choice point shorter. This is especially important with a region-based execution model, where an active choice point can cause deallocation of a region to be postponed.

The cut construction phase inserts a cut at the earliest place in each clause where

- A programmer-supplied cut has not yet been executed.
- No predicates that may leave their own choice points have yet been called. A conservative, fixpoint-based determinacy analysis is used to determine which predicates risk leaving choice points.
- Enough destruct instructions have been executed to know that the values of the input parameters will cause a destruct instruction to fail in each subsequent clause before it can do any "dangerous" actions. Here, "dangerous" actions include the builtin instruction (which may do I/O), calling a recursive predicate (which may loop indefinitely), and calling any predicate that contains "dangerous" actions itself.

The third condition is always true in the last clause, which means that the cut construction phase always inserts a cut as the first instruction in the last clause of any predicate. This does not harm performance because our P and RP implementations *expect* a cut to be there and insert one by their own accord if they do not find it.

The final P program is the output of the cut construction case.

[10] That is, cuts whose effect is not observable in the behaviour of the program.

8.10 Conclusion

We have shown how to translate Prolog code into P code that makes good use of P's facilities.

The translation consists of a series of individual transformations. This design makes it possible to convince oneself that the translation is correct, that is, it preserves the semantics of the original P program. If the P-like intermediate code between the phases is interpreted as a Prolog program (using the Prolog equivalents of each P instruction that we have given in Chapters 2 and 3), it is possible to understand the correctness (if not the eventual purpose) of each transformation separately. At the end of the translation we have a P program whose Prolog interpretation is equivalent to the original Prolog program. We can then invoke the similarity between Prolog semantics and our P semantics we suggested in the beginning of Section 3.7.

Although we have not actually proved any of the transformations correct, we feel reasonably certain that it can be done. We are therefore confident that the Prolog-to-P translator produces correct P code.

Our experience from practical experiments with our prototype translator is that the *efficiency* of the generated P code is in most cases satisfactory. It generally manages to distinguish between the possible roles of predicate parameters and unification in the same way the Prolog programmer thinks about her program.

Sometimes, however, the TP type inference we describe in Section 8.8.1 exhibited an strange lack of precision, where some modes became `ref` instead of `bare` for no apparent reason. We eventually discovered that the problem is due to unexperienced interference between monomorphic type inference, declaration-free type recursion, and the type-based mode analysis. See Section 12.1.8 for an explanation of what goes wrong. Similar problem can occur in our region inference algorithm, and the effect appears to be possible for many different type-based analyses. Further research may be necessary to identify a generally applicable way of avoiding it.

Chapter 9

TRP: a region-annotated type system for RP

In this chapter we define the region-annotated type system TRP for RP programs. The system builds on the ideas we have presented in the TGP system of Chapter 6 and to some extent on those of TP in Chapter 7.

TRP's reason for existing is to help prove that some RP programs are "region safe", that is, that it will never try to inspect values in regions that have been deallocated. Remember that the semantics in Section 4.4 uses the error state wrong_4 to signal that that has happened. So we can define

Definition 9.1 *An RP program is **region safe** if, when it is executed according to the definitions in Section 4.4, the error state* wrong_4 *cannot arise.*

The time when TRP is used is in the region inference phase of our region-based Prolog implementation. The region inference takes as input a TP-typeable P program and is expected to produce an equivalent region-safe RP program. In this process, TRP serves as

- a *design guideline*: The region inference works by adding to the P program exactly those region annotations that are necessary to make it a TRP–well-typed RP program.
- a *divide-and-conquer point* in the argument that the region inference works. This argument is naturally divided into two parts: First, argue that the output of (certain phases of) the region inference is always TRP well-typed. Second, prove that every TRP–well-typed program is region safe.

TRP is similar to—and derived from—the region inference rules of Tofte and Talpin [1993]. It is simpler than that, however, because we can look away from the complex machinery they use to handle higher-order functions. In other aspects, TRP is stronger than the Tofte–Talpin system, because we allow regions to be deallocated even when types mentioning them are still in scope.

9.1 The syntax and meaning of TRP types

Like the TP types, the TRP type system is an extension of the TGP types. Instead of adding a mode to each type node, we add the name of a region register:

$$\begin{array}{rcl} \mu & ::= & \rho.\tau \\ \tau & ::= & \mathsf{int} \\ & | & f_1(\mu_{11}, ..., \mu_{1|f_1|}) + \cdots + f_n(\mu_{n1}, ..., \mu_{n|f_n|}) \\ \rho & ::= & \text{(a region register from the program text)} \end{array}$$

where, in the second production for τ, the f_is are all distinct and none of them are numbers. Types can be recursive, just as they can in TGP and TP. As for TP, we call each μ a **type**.[1]

The intended meaning of a type like "$\rho.\texttt{foo}(\mu_1) + \texttt{bar}(\mu_2)$" is that it describes either a variable allocated in the region bound to ρ or a `foo` or `bar` structure allocated in the region bound to ρ. In either case any instantiated variables necessary to find the value must also be allocated in ρ.

When we define what this means precisely, we face the problem that the region annotations in types are region *registers* (*e.g.*, syntactic identifiers from the RP program) whereas the region store works in terms of region *numbers* which are a dynamic concept. Therefore the relation between types and pointers in a store must include the region environment $\mathcal{R}$ in which the type is to be interpreted. We write it

$$\mathcal{R}, \sigma \vdash \alpha : \mu$$

which means, "In store σ the term represented by α has the type μ when μ is interpreted in $\mathcal{R}$.

Definition 9.2 *The relation* $(,\vdash:)$ *is defined co-inductively*[2] *by the following inference system:*

$$\frac{}{\mathcal{R}, \sigma \vdash (\mathcal{R}(\rho), o) : \rho.\tau}\ \sigma(\mathcal{R}(\rho))(o) = \mathsf{uninst}$$

$$\frac{\mathcal{R}, \sigma \vdash \alpha' : \rho.\tau}{\mathcal{R}, \sigma \vdash (\mathcal{R}(\rho), o) : \rho.\tau}\ \sigma(\mathcal{R}(\rho))(o) = \alpha'$$

$$\frac{}{\mathcal{R}, \sigma \vdash (\mathcal{R}(\rho), o) : \rho.\mathsf{int}}\ \sigma(\mathcal{R}(\rho))(o) \in \mathbb{Z}$$

[1] This is different form the terminology of most of the previous work on region type systems, where μ is called a "type-and-place" and τ a "type". We feel that the role of μ's in the typing rules are closer to to the roles of types in ordinary type systems, so the μ's should have the privilege of being called types.

See footnote 2 on page 97 for the rationales for other deviations from the Tofte–Talpin syntax.

[2] See the remarks following Definition 7.1 on page 97 for a discussion of co-inductive definitions.

$$\frac{\forall j\,[\mathcal{R}, \sigma \vdash \alpha_j : \mu_{ij}]}{\mathcal{R}, \sigma \vdash (\mathcal{R}(\rho), o) : \rho.\,\sum_i f_i(\mu_{ij})_j}\ \sigma(\mathcal{R}(\rho))(o) = f_i(\alpha_1, ..., \alpha_{|f_i|})$$

$$\frac{}{\mathcal{R}, \sigma \vdash \alpha : \rho.\tau}\ \rho \notin \mathrm{Dom}\,\mathcal{R}$$

The last rule implies that for ρs not defined in $\mathcal{R}$, the type $\rho.\tau$ describes everything. That means that removing a region from $\mathcal{R}$ can never cause anything to cease being described by any type.

We also need a notation for the region variables that occur in a type:

$$\begin{aligned} \mathrm{rgns}(\rho.\mathsf{int}) &= \{\rho\} \\ \mathrm{rgns}(\rho.\sum_i f_i(\mu_{ij})_j) &= \{\rho\} \cup \bigcup_{i,j} \mathrm{rgns}(\mu_{ij}) \end{aligned}$$

(see Section A.1.4 for a discussion of how to give meaning to this when types can be recursive. We define "rgns" to be the least solution to the recursive definition).

We extend the T(G)P predicate types to include the names of the predicate's formal region parameters. A TRP predicate type has the form

$$\left\langle \left[\rho_1, ..., \rho_{\lceil pr \rceil}\right] \frac{\mu_1, ..., \mu_{\lceil pr \rceil}}{\mu'_1, ..., \mu'_{\lceil pr \rceil}} \right\rangle$$

where the ρ_is are distinct region registers. Such a region-annotated type is implicitly **region polymorphic**. That means that when the predicate is called, the region registers in the types of the actual parameters need not be identical to the ones in the predicate type, except that identical regions in the predicate type must correspond to identical regions in the call context.

9.2 Typing rules for RP programs

Define a TRP **typing** $\mathcal{T}$ by analogy with the TGP definition (Definition 6.2 on page 92).

A TRP **well-typing** is one that complies with the rules in Figures 9.1 and 9.2. The main judgement form in these rules is

$$\mathcal{T}, \Delta \vdash b \rightsquigarrow (\mu_1, ..., \mu_n)$$

where Δ is a mapping from $\langle \mathit{Region} \rangle$ to the set $\{\mathsf{unknown}, \mathsf{parm}, \mathsf{local}, \mathsf{dead}\}$ which describe properties of the region environment prior to the execution of b. Intuitively,

$\Delta(\rho) = \mathsf{unknown}$ means that the region register ρ is not in scope, but it is possible to create a region with that name without risking to create a name conflict.

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{call } pr[\rho_1,\ldots,\rho_{\lceil pr\rceil}]\ (x_1,\ldots,x_{\lceil pr\rceil}) \rightarrow (x'_1,\ldots,x'_{\lfloor pr\rfloor})\ \ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} \mathcal{T}(pr) = \left\langle \left[\rho'_1,\ldots,\rho'_{\lceil pr\rceil}\right] \frac{\mu_1,\ldots,\mu_{\lceil pr\rceil}}{\mu_1,\ldots,\mu_{\lfloor pr\rfloor}} \right\rangle \\ \forall i : \mathcal{T}(x_i) = \theta\mu_i \\ \forall i : \mathcal{T}(x'_i) = \theta\mu'_i \\ \forall i : \rho_i = \theta\rho'_i \\ \forall i : \Delta(\rho_i) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{builtin } pr(x_1,\ldots,x_{\lceil pr\rceil}) \rightarrow (x'_1 \text{ at } \rho'_i,\ldots,x'_{\lfloor pr\rfloor} \text{ at } \rho'_i)\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} \forall i : \mathcal{T}(x_i) = \rho_i.\text{int} \\ \forall i : \Delta(\rho_i) \in \{\text{parm}, \text{local}\} \\ \forall i : \mathcal{T}(x'_i) = \rho'_i.\text{int} \\ \forall i : \Delta(\rho'_i) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{construct at } \rho\ f(x_1,\ldots,x_{|f|}) \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f_i \notin \mathbb{Z} \\ \tau_0 = f(\mathcal{T}(x_1),\ldots,\mathcal{T}(x_{|f|})) + \ldots \\ \mathcal{T}(x') = \rho.\tau_0 \\ \Delta(\rho) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{construct at } \rho\ f() \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f \in \mathbb{Z} \\ \mathcal{T}(x') = \rho.\text{int} \\ \Delta(\rho) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{destruct } x \text{ as } f \rightarrow (x'_1,\ldots,x'_{|f|})\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f \notin \mathbb{Z} \\ \tau_0 = f(\mathcal{T}(x'_1),\ldots,\mathcal{T}(x'_{|f|})) + \ldots \\ \mathcal{T}(x) = \rho.\tau_0 \\ \Delta(\rho) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{destruct } x \text{ as } f \rightarrow ()\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} f \in \mathbb{Z} \\ \mathcal{T}(x) = \rho.\text{int} \\ \Delta(\rho) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{makevar at } \rho \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} \mathcal{T}(x') = \rho.\tau \\ \Delta(\rho) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{deref } x \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} \mathcal{T}(x) = \mathcal{T}(x') = \rho.\tau \\ \Delta(\rho) \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{alias } x \rightarrow x'\ \&\ b \rightsquigarrow \vec{\mu}}\ \mathcal{T}(x) = \mathcal{T}(x')$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{unify } x_1{=}x_2\ \&\ b \rightsquigarrow \vec{\mu}} \begin{cases} \mathcal{T}(x_1) = \mathcal{T}(x_2) \\ \forall \rho \in \text{rgns}(\mathcal{T}(x_1)) : \rho \in \{\text{parm}, \text{local}\} \end{cases}$$

$$\frac{\mathcal{T},\Delta \vdash b \rightsquigarrow \vec{\mu}}{\mathcal{T},\Delta \vdash \text{cut}\ \&\ b \rightsquigarrow \vec{\mu}} \qquad \frac{}{\mathcal{T},\Delta \vdash \text{fail} \rightsquigarrow \vec{\mu}}$$

$$\frac{}{\mathcal{T},\Delta \vdash \text{exit}(x_1,\ldots,x_n) \rightsquigarrow (\mathcal{T}(x_1),\ldots,\mathcal{T}(x_n))}\ \forall \rho : \Delta(\rho) \neq \text{local}$$

Figure 9.1: Typing rules for RP programs, part 1. Compare with Figure 6.1 on page 93.

$$\frac{\mathcal{T}, \Delta\{\rho \mapsto \mathsf{local}\} \vdash \mathfrak{b} \rightsquigarrow \vec{\mu}}{\mathcal{T}, \Delta \vdash \mathsf{makeregion} \rightarrow \rho \;\&\; \mathfrak{b} \rightsquigarrow \vec{\mu}} \Delta(\rho) = \mathsf{unknown}$$

$$\frac{\mathcal{T}, \Delta\{\rho \mapsto \mathsf{dead}\} \vdash \mathfrak{b} \rightsquigarrow \vec{\mu}}{\mathcal{T}, \Delta \vdash \mathsf{killregion}\ \rho \;\&\; \mathfrak{b} \rightsquigarrow \vec{\mu}} \Delta(\rho) = \mathsf{local}$$

$$\frac{\mathcal{T}, \Delta \vdash \mathfrak{b} \rightsquigarrow (\mu'_1, \ldots, \mu'_{\lfloor pr \rfloor})}{\begin{array}{l}\mathcal{T} \vdash pr[\rho_1,\ldots,\rho_{\lfloor pr \rfloor}] \;\text{:-}\\ \quad \mathsf{enter} \rightarrow (x_1,\ldots,x_{\lceil pr \rceil}) \;\&\; \mathfrak{b}.\end{array}} \begin{cases} \mathcal{T}(pr) = \left\langle [\rho_1, \ldots, \rho_{\lfloor pr \rfloor}]\ \frac{\mu_1,\ldots,\mu_{\lceil pr \rceil}}{\mu'_1,\ldots,\mu'_{\lfloor pr \rfloor}} \right\rangle \\ \forall i : \mathcal{T}(x_i) = \mu_i \\ \Delta = \lambda\rho. \begin{cases} \mathsf{parm} \text{ if } \exists \rho_i = \rho \\ \mathsf{dead} \text{ if } \exists \mu_i : \rho \in \mathrm{rgns}(\mu_i) \\ \mathsf{dead} \text{ if } \exists \mu'_i : \rho \in \mathrm{rgns}(\mu'_i) \\ \mathsf{unknown} \text{ otherwise} \end{cases} \end{cases}$$

$$\frac{\mathcal{T} \vdash pr[\rho_1,\ldots,\rho_{\lfloor pr \rfloor}] \;\text{:-}\; c_1. \quad \cdots \quad \mathcal{T} \vdash pr[\rho_1,\ldots,\rho_{\lfloor pr \rfloor}] \;\text{:-}\; c_n.}{\mathcal{T} \vdash pr[\rho_1,\ldots,\rho_{\lfloor pr \rfloor}] \;\text{:-}\; c_1;\ldots;c_n.}$$

Figure 9.2: Typing rules for RP programs, part 2

$\Delta(\rho) = \mathsf{parm}$ means that ρ is in scope and is a formal region parameter. We know that the region exists; killing it is not allowed.

$\Delta(\rho) = \mathsf{local}$ means that ρ is in scope; the region it is bound to exists but must be killed before exiting from the predicate.

$\Delta(\rho) = \mathsf{dead}$ means the region register ρ is not in scope, but nevertheless may occur in the type of a value register. The difference between dead and unknown is that it is not allowed to use a dead region register in a makeregion instruction.

Except for the Δ, the interpretation of the typing judgements is as for TGP.

The rule for the call instruction implements region polymorphism. The θ that appears in the side condition is a **region renaming**, that is, a mapping from region registers to region registers. This mapping naturally generalises to entire types: θ simply gets applied to the region component of each type node. Note that there is no requirement that θ is injective.

Most of the instructions need only the "topmost" regions in the their operands' types to exist. This means that a code sequence such as

```
(...)
  & construct at r0 blarf( ) → x
  & construct at r1 foo(x) → y
  & killregion r0
  & destruct y as foo → (xx)
  & construct at r2 bar(xx) → z
(...)
```

is perfectly legitimate. The two latter destruct and construct instructions will be pull the `xx` pointer (which used to point to the `blarf` structure but is now a dangling pointer) out of the `foo` structure and reinstall it in

a `bar` structure. This is operationally safe, so the TRP type system only makes sure that the program does not try to read what that dangling pointer points to.[3]

The exception to this general rule is the unify instruction. Because a unification operation may need to traverse the entire terms bound to its operands, the typing rule for unify requires that every region register in the type of the operands exist.

Note also that unify requires the types of the operands, inclusive of the region annotations, to be identical. That is a conservative way of protecting against an uninstantiated variable in one term being instantiated to a part of the other term which is in a different region. This rule could be made less conservative by employing mode information, but we leave that as a possible further extension (see Section 12.1.9).

The rule for predicate definitions (on Figure 9.2) contains some subtleties related to region polymorphism. Unlike Tofte and Talpin [1993] we do not require that all of the regions in the types of the input and output parameters are among the region parameters. Instead, those region registers in the types that are not region parameters get marked as dead in the initial Δ. That means that the body of the predicate must assume that those regions have been deallocated; it can neither inspect values of the corresponding types nor create new values of them. Whether the regions are *in fact* deallocated depends on the caller of the predicate. It is possible for the same predicate to be called from places where the regions do exists as well as from places where they don't. Aiken et al. [1995] call this property **state polymorphism**.

9.3 Well-typed TRP programs do not go wrong_4

The usefulness of the TRP type system depends on the following theorem:

Theorem 9.3 *When a TRP program is executed according to the semantics in Section 4.4, the error state* wrong_4 *never arise.*

Recall that wrong_4 is the kind of error that occurs if the program tries to reference a dangling pointer. In other words, the theorem states that the TRP type rules prevents a program from allocating a value in a region that will be deallocated before the last possible reference to the value.

Due to time constraints we only sketch how we think Theorem 9.3 can be proved.

The proof has two main parts. The first part consists of proving that if any RP program goes wrong_4, it does so because it tries to dereference a

[3] This feature is in contrast to Tofte and Talpin [1993]'s region type system. The most direct RP analogue of their rule for `letregion` would be to require that all region registers in the type of a value register exists as long as the value register is "live". Our rule—which can be seen as incorporating some of the post-processing suggested by Aiken et al. [1995] into the type system—is obviously stronger than this; we also found it easier to implement.

$$\frac{}{\mathcal{T}, \sigma, \xi, F, [] \vdash () \looparrowright []}$$

$$\frac{\mathcal{T}, \sigma, \xi, F, \mathcal{R}' \vdash \mathcal{R}, \mathcal{K}, \mathcal{E}, b \looparrowright \kappa \qquad \mathcal{T} \vdash \delta \text{ ok}}{\mathcal{T},\sigma,\xi,F \vdash (\tilde{\mu}_1, ..., \tilde{\mu}_n) \looparrowright (\mathcal{R},\mathcal{K},\mathcal{E},(x_1, ..., x_n),b,\delta) :: \kappa} \begin{cases} \exists\, \mathcal{R}' \\ \forall i : \mathcal{R}'[\mathcal{T}(x_i)] = \tilde{\mu}_i \end{cases}$$

$$\frac{\mathcal{T}, \sigma, \xi, F' \vdash (\mathcal{R}'[\mu_1], ..., \mathcal{R}'[\mu_n]) \looparrowright \kappa}{\mathcal{T}, \sigma, \xi, F, \mathcal{R}' \vdash \mathcal{R}, \mathcal{K}, \mathcal{E}, b \looparrowright \kappa} \begin{cases} \exists\, \Delta \\ \operatorname{Codom} \mathcal{R}' \cap F = \varnothing \\ \mathcal{R} = \mathcal{R}'|_{\Delta^{-1}\{\mathsf{parm},\mathsf{local}\}} \\ \operatorname{Codom} \mathcal{R} \subseteq \operatorname{Dom} \sigma \\ \mathcal{K} = \Delta^{-1}\{\mathsf{local}\} \\ \forall x \in \operatorname{Dom} \mathcal{E} : \xi(\mathcal{E}(x)) = \mathcal{R}'[\mathcal{T}(x)] \\ \mathcal{T}, \Delta \vdash b \rightsquigarrow (\mu_1, ..., \mu_n) \\ F' = F \cup \mathcal{R}'(\Delta^{-1}\{\mathsf{unknown}, \mathsf{local}\}) \end{cases}$$

$$\frac{\begin{array}{c}\mathcal{T}, \sigma, \xi, \varnothing, \mathcal{R}' \vdash \mathcal{R}, \mathcal{K}, \mathcal{E}, b \looparrowright \kappa \\ \mathcal{T} \vdash \delta \text{ ok} \qquad \mathcal{T} \vdash fs \text{ ok}\end{array}}{\mathcal{T} \vdash (\sigma, \mathcal{R}, \mathcal{K}, \mathcal{E}, b, \delta, \kappa) :: fs \text{ ok}} \begin{cases} \exists\, \xi, \mathcal{R}' \\ \text{consistent}(\sigma, \xi) \end{cases} \qquad \frac{}{\mathcal{T} \vdash [] \text{ ok}}$$

Figure 9.3: Inductive definition of the "$\vdash$ ok" relation used in Lemma 9.4. The side conditions of the form "$\exists X$" have no formal meaning but draw attention to the fact that X does not occur in the conclusion of the rule. The variable F range over finite sets of region numbers (intuitively interpreted as region numbers that are "forbidden" in the proofs). The relation consistent(σ, ξ) *is defined in the lemma.*

pointer (r, o) in a store σ that does not include the region r. In other words, whenever the programs knows of a pointer (r, o), either $\sigma(r)$ does not exist, or it does exist and its domain includes o. This part of the theorem can be proved directly from the RP semantics in Section 4.4.

The second part of the proof is the hard one. It consists of proving that whenever a TRP program tries to dereference a pointer (r, o), the region r will still exist. For this, a couple of new concepts are needed:

Run-time types are types that have the same structure as TRP types except that each μ contains a region number r instead of a region register name ρ (recall that region numbers are a run-time concept; the RP semantics uses region environments $\mathcal{R}$ to map region registers to region numbers). Let $\tilde{\mathbb{T}}$ be the set of all run-time types.

A region environment $\mathcal{R}$ can, in the obvious way, be applied to a TRP type μ, resulting in a run-time type $\tilde{\mu} = \mathcal{R}[\mu]$, if all the region registers found in μ are in $\operatorname{Dom} \mathcal{R}$.

A **TRP store typing** is a function $\xi : \langle \mathit{Addr} \rangle \xrightarrow{\text{fin}} \tilde{\mathbb{T}}$. In contrast to the store typings we used for TP in Chapter 7, we do not need to let a store typing map each address to a *set* of types; it is enough with a single type.

With these concepts we can formulate a working induction lemma:

Lemma 9.4 *When a TP program with well-typing $\mathcal{T}$ is executed according to the semantics in Section 3.7, "$\mathcal{T} \vdash$ fs ok" can be derived by the inference rules in Figure 9.3 for every $\langle State \rangle$ fs that appears during the computation.*

The relation consistent(σ, ξ) *that appears in the rules is defined to be true iff*

- $\mathrm{Dom}\,\xi = \mathrm{Dom}\,\sigma$
- *For each* $(r, o) \in \mathrm{Dom}\,\sigma$, $\xi(r, o) = r.\tau$ *for some* τ.
- *For each* $(r, o) \in \mathrm{Dom}\,\sigma$ *with* $\sigma(r)(o) = \alpha'$, $\xi(r, o) = \xi(\alpha')$.
- *For each* $(r, o) \in \mathrm{Dom}\,\sigma$ *with* $\sigma(r)(o) \in \mathbb{Z}$, $\xi(r, o) = r.\mathsf{int}$.
- *For each* $(r, o) \in \mathrm{Dom}\,\sigma$ *with* $\sigma(r)(o) = f(\alpha_1, ..., \alpha_{|f|})$, $\xi(r, o) = r.f(\tilde{\mu}_1, ..., \tilde{\mu}_{|f|}) + ...$ *such that* $\forall i : \alpha_i \in \mathrm{Dom}\,\sigma \Longrightarrow \xi(\alpha_i) = \mu_i$.

The lemma can be proved by induction on the length of the computation. It is easy to see that the initial state defined in Section 4.4.1 satisfies the required property. The induction step consists of proving that if "$\mathcal{T} \vdash$ fs ok" can derived and the successor state of fs is fs′, then a derivation of "$\mathcal{T} \vdash$ fs′ ok" can be constructed from the derivation of "$\mathcal{T} \vdash$ fs ok".

The detailed constructions for each kind of RP instruction are very verbose, so due to time constraints we do not write them down. Instead we give a few general hints about how they can be constructed.

The most important step is showing that killregion instructions are safe. Here the F's are used to see that the *number* of the region that is deallocated is not used in any other region environments than the one of the currently executing predicate.

When a construct or makevar instruction allocates a new cell in the store, the (run-time) type of the new cell can generally just be added to the store typing ξ. There can be a problem, however, if the type of the new cell contains region registers that are not defined in the current region environment. In that case we simply enter the unknown region register into the $\mathcal{R}'$ region environment, mapping to a fresh "tentative" region number[4]. When (and if) the unknown region register is eventually created by a makeregion, we substitute the new real region number for the tentative region number in the region environment *and* everywhere in the store typing ξ. This does not break the consistency of ξ because the tentative region number does not actually appear in the store[5], and we can use the F mechanism to see that the tentative number does not occur in other region environments than the current one.

The unify instruction may change the store, but the new store is consistent with the previous store typing. This can be seen by induction on the

[4]We assume, without loss of generality, that the set of region numbers has an infinite subset which is never used by makeregion instructions, so that it is always possible to select a tentative region number that does not risk colliding with a later makeregion.

[5]Auxiliary lemma: none of the numbers we use for tentative region numbers ever get defined in any store.

number of "$\triangleright\!\!-$" steps (Definition 3.9ff) in the unification process. The induction hypothesis is that ξ is consistent with σ and that $\xi(\alpha_1) = \xi(\alpha_2)$ for each $(\alpha_1, \alpha_2) \in \mathcal{C}$.

The construction for the remainder of the cases are straightforward, although there is a lot of detail to take care of in connection with predicate calls and returns.

Chapter 10

A region inference algorithm

In this chapter we describe how a TP-typeable P program[1] can be transformed to a good equivalent region-safe RP program.

The translation proceeds in three overall phases:

A. Translate the P program to a preliminary TRP-typeable RP program whose region annotations are as fine-grained as possible—that is, the RP program does not use the same region for any two allocations, unless that is absolutely necessary for the program to be TRP typeable. Phase A does not decide whether it is desirable to *use* this extreme granularity, and it does not even decide when regions should be created and deallocated, except for inserting trivial default decisions to be changed in later phases.

B. Use the fine-grained region annotations to place killregion instructions in the program as early as possible, such that every value is deallocated as soon as the TRP type system allows.

C. Remove unnecessary generality and granularity from the region annotations. The goal here is to make the the program uses the region-based memory manager more efficiently (*e.g.*, by using fewer regions), without compromising the decisions about deallocation times made in phase B.

The fact that a TRP-typeable program occurs already in phase A may lead to the impression that that the "real work" happens in phase A and that phases B and C are just finishing touches. That would not be fair, however, because the task of the region inference is not just to produce *some* region-safe RP equivalent of the input program but to produce a *good* one, where "good" means that the RP program should make efficient use of the region-based memory manager.

It is easy to produce a bad RP equivalent of a P program—you just let everything happen in a single big region. Producing a good one is a much more intricate problem, and all of our 3 phases contribute importantly to this task. Phase A is the most complex indivisible step in the region inference

[1] Or rather, a P program that would be TP-typeable if sufficiently many alias instructions were inserted.

```
embla:~/2del/speciale/impl$ ./reginfer -d -v 8queens.P
Reading 8queens.P
Checking variable scopes
Dumping to 8queens.dump.b0
Doing primary region annotation
Dumping to 8queens.dump.b1
Doing removal of unused region parameters
Dumping to 8queens.dump.b2
Doing killregion instructions
Dumping to 8queens.dump.b3
Doing removal of excess region annotations
Dumping to 8queens.dump.b4
Doing extra region parameter insertation
Dumping to 8queens.dump.b5
Doing region merges
Dumping to 8queens.dump.b6
Doing removal of unused region parameters
Dumping to 8queens.dump.b7
Doing main predicate replacement
Dumping to 8queens.dump.b8
Doing makeregion instructions
Checking final RP program
Writing to 8queens.RP
embla:~/2del/speciale/impl$
```

Figure 10.1: A sample run of the `reginfer` *program.*

process, but it would need to be a lot more complex if not phases B and C were present to make the decisions that *can* be made one by one.

The difference between phases B and C is somewhat technical of nature yet also theoretically important. The transformations in phase B work on TRP-typeable programs and do not make sense at all for programs that are not. The argument for their correctness is that they preserve TRP typeability. On the other hand, the transformations in phase C preserve only region safety, not TRP typeability (which is a conservative approximation to region safety). It would be possible to design a type system such that typeability was preserved by the latter transformations, but such a type system would be much more complex to specify and reason about. Distinguishing between the two groups of transformations allows us to use a simple type system and still get reasonably efficient RP code out of the transformation.

The existence of phase C emphasises that TRP does not occur in a black-box specification for the region inference. As long as the resulting RP code is region safe, the rest of a region-based Prolog implementation does not care whether a TRP typing has been used to obtain that property or not.

We have implemented the region inference described here in the `reginfer` program which is part of our prototype region-based Prolog implementation [Makholm 2000]. When `reginfer` is invoked with the switches

`-v` and `-d` it announces the individual phases of the translation and dumps the result of each to auxiliary files for inspection by curious users (see Figure 10.1).

The organisation of this chapter does not match the phase division in `reginfer`'s output as closely as Chapter 8 does that of `pro2P`. The reason for this is that we want to stress our overall three-phase design; such a grand design was not present for `pro2P`.

10.1 Phase A: Fine-grained region annotation

As we have stated, the purpose of phase A is to find a TRP equivalent of the input that only puts two values in the same region if absolutely necessary. We can do that by finding a TRP program where no region register appears in more places in the TRP typing than necessary.

The first step in this analysis is to compute the general shape of a TRP typing, that is, a TRP typing where the region annotations are ignored. This is easy to do with well-known techniques (see Section 8.6 for a short discussion of the few pitfalls).

Next, fresh copies of all of the types are made such that there is no sharing between the types of different registers.[2]

Then, we have to compute which type nodes need to be annotated with identical region registers to fulfil the type equalities in the TRP rules on pages 123 and 124. We can use a standard union-find data structure (with type nodes as the elements) for most of this job, as most of the constraints take the form "the region annotations of this type must equal the region annotations of that type".

A slight extension of the algorithm is necessary to implement the region-polymorphic rule for call. The call rule states that there must be a functional relationship between the region registers mentioned in the predicate type (*i.e.*, type of the formal input and output parameters) and the regions mentioned in the types of the actual parameters. In other words, if type nodes μ_1 and μ_2 appear in the predicate type and μ_1 and μ_2 are annotated with a common region register, their counterparts μ_1' and μ_2' in the types of the actual parameters also need to be annotated with a common (but possibly different from the one in the predicate types) region register.

It is easy to achieve this when analysing the program by hand, and it can be implemented efficiently in a union-find–based algorithm as follows: We distinguish between **lead** nodes and **nonlead** nodes. The nodes that make up a type in a predicate types are all lead nodes; all other nodes are nonlead nodes. Each lead node has attached a list of **dependent** nodes[3]. There is one dependent node for each call of the predicate in whose type the lead node

[2]Sharing inside the type of each register is preserved in these copies, so that recursive types are supported. This sometimes harms the precision of the analysis; see Section 12.1.8 for a discussion of that.

[3]A dependent node can itself be either a lead node or a nonlead node

occurs—namely the type node that occupies the corresponding position in the type of the actual parameter.

During the analysis the invariant holds that all type nodes in the same partition of the union-find structure come from the types of registers that belong to the same predicate. This implies that whenever two lead nodes occur in the same partition, they have the same number[4] of dependent nodes, matched in pairs by coming from the same call instructions.

We can also easily maintain the invariant that if a partition contains any lead nodes at all, one of the lead nodes will be the "root" node of the partition.

Now replace the *union* algorithm of the union-find structure with the following *reg-union* algorithm:

1. If the two partitions to be unified are the same, then stop.
2. If at most one of the partitions to be unified contain lead nodes, do a normal *union* operation and stop.
3. Otherwise, the "root" node of either partition is a lead node. Save a local copy of the two dependent-node lists of the root node.
4. Do a normal *union* operation.
5. Recursively, do a *reg-union* operation on each matching pair of nodes from the saved dependent-node lists. (It is important that this happens after step 4, to prevent infinite recursion).

When all the necessary *reg-union* operations have been performed, a fresh region register is created for each partition and used to annotate the type nodes in the partition.

Then only little remains to create a preliminary TRP-typed RP program:

- Construct a formal region parameter list for each predicate containing all region variables that appear in the predicate's formal parameter types.
- Construct the corresponding actual region parameter lists for each call statement.
- For each region register that is not a formal parameter, insert a makeregion instruction at the beginning of the clause where it appears, and a killregion instruction at its end.
- Annotate construct, makevar, and builtin instructions with region annotations as selected by the result register's type.

In our implementation the internal representation of the program is then written to the dump *name*`.dump.b1`.[5]

[4] A dependent node can occur more than once in a lead node's dependent-node list; then it counts more than once here.

[5] In the internal representation the makeregion and killregion instructions are implicit at this point. makeregion and killregion instructions for every region register in each clause that is not a region parameter are supposed to be clustered at just after the enter pseudoinstruc-

10.2 Phase B: TRP-based region optimizations

Recall that the characteristic of the transformations in phase B is that their soundness follow from preservation of TRP typeability.

In our design, phase B comprises two optimizations that work together to make each region in the program be deallocated as early as allowed by the TRP type system.

10.2.1 Removal of unused region parameters

The first of the optimizations consists of removing as many region parameters as TRP permits. The reason for doing this is that regions must not be used as actual region parameters after they have been deallocated; thus removing regions from the parameter lists might allow some regions to be deallocated earlier.

The optimization works by computing which of the region parameters has any reason *not* to be removed:

- A region parameter ρ cannot be removed if it is used by a builtin, construct, destruct, makevar, deref, or unify statement, in the sense that the typing rule for the statement (on page 123) requires $\Delta(\rho) \in \{\mathsf{parm}, \mathsf{local}\}$.
- A region parameter ρ cannot be removed if it is used by a call statement to initialise another region parameter that cannot be removed.
- Any other region parameter can be removed.

In our implementation, the set of region parameters that cannot be removed are computed by a simple fixpoint iteration. Then, in a single pass over the program, all the removable region parameters are removed from predicate definitions and call statements.

The dump *name*`.dump.b2` shows the program after unused region parameters have been removed.[6]

10.2.2 Moving killregion instructions backwards

When unused region parameters have been removed, the killregion instructions are moved as far backwards as possible (where "backwards" means "towards the enter pseudoinstruction") using the rules that

- "killregion ρ" commutes with "killregion ρ'"
- "killregion ρ" commutes with "makeregion ρ'" if $\rho \neq \rho'$.

tion and just before the exit pseudoinstruction, respectively.

The actual TRP types are also implicit, but the important information in the types is retained as annotations on destruct, deref, and unify which name the regions that each instruction may need to read values from.

[6] makeregion and killregion instructions are still implicit in this dump.

- "makeregion $\rightarrow \rho$ & killregion ρ" may be removed from the program.
- "killregion ρ commutes with some other instruction if the typing rule (on page 123) for that instruction does not require that $\Delta(\rho) \in \{\mathsf{parm}, \mathsf{local}\}$.

Because this phase comes after the removal of unused region parameters, it allows killregion instruction to move before call instructions whenever the region to be killed is not used as an actual region argument. The region may still occur in the type of one of the value parameters.

The dump *name*`.dump.b3` shows the program after unused region parameters have been removed.[7]

10.2.3 Removal of excess region annotations

In our implementation, the boundary between phases B and C is marked by a phase that discards the type annotations from the internal representation of the RP program.

In fact, the type annotations that have been used in phase B are simplified annotations on each instruction that enumerate the regions ρ for which $\Delta(\rho)$ must be parm or local. These annotations are now removed from all instructions but the ones where they are part of the official RP syntax, namely call, construct, makevar, and the output operands of builtin.

The dump *name*`.dump.b4` shows the program after these annotations have been removed.

10.3 Phase C: Other region optimizations

The transformations in phase C are general optimizations that can be applied to any RP program independently of its origin. They do not expect their input to be TRP well-typed, nor do they guarantee their output to be even if the input is.

Phases A and B have concentrated on the abstract region concept with the ultimate goal of releasing heap memory blocks as soon as our techniques can show statically that they are not necessary anymore.

The optimizations of phase C fall in two groups, both of which are motivated by properties of the region-based memory manager we developed in Chapter 5, rather than abstract concern over object lifetimes.

The first group of optimizations is motivated by the observation in Section 5.1.3 that a few large regions are in general more space efficient than a lot of small regions. The goal of phase A was to use as many and as small regions as possible, which was a worthy goal by then because it allowed us to reason more precisely about the lifetimes of each regions in phase B. Now,

[7] In this and all of the following dumps, makeregion but not killregion instructions are implicit. Each killregion instruction implies an implicit makeregion at the very beginning of the clause.

however, the lifetimes have been fixed, so it is time to *merge* regions to the greatest possible extent that does not cause any memory blocks to live too much longer than before.

The second group of optimizations is motivated by the fact that the *make-region* primitive allocates a page of heap space where it can create a region management structure. It therefore makes sense to allocate the region as late as possible. Ideally that would mean immediately before the first instruction that allocates something in the region, but because RP does not allow a region to outlive the clause that contains its makeregion instruction we have to compromise a little here.

10.3.1 Merging regions

The basic transformation for region merging is simple: whenever a sequence such as

$$\ldots \;\&\; \mathsf{killregion}\ \rho_1 \;\&\; \ldots \;\&\; \mathsf{killregion}\ \rho_2 \;\&\; \ldots$$

occurs in the program *and* there is no call instruction between the two killregion instructions, the first killregion is removed[8] and any references to ρ_1 in the clause are replaced with ρ_2.

We do this even if there is an allocating instruction such as construct between the two killregion statements. It might be argued that that is not wise, because before the merge the construct could reuse the memory freed by the killregion statement. However, eliminating ρ_1 in itself frees 6 words for the region's management record and—on average—half a page of slack space, so we believe that in most cases the merge causes a net improvement in the memory efficiency of the program.

This is, however, not the whole story. Our experience with an earlier prototype of the region inference algorithm showed that clauses that ended in something like

$$\ldots \;\&\; \mathsf{call}\ \mathfrak{pr}[\mathtt{r17}](\ldots) \rightarrow (\ldots) \;\&\; \mathsf{killregion}\ \mathtt{r17} \;\&\; \mathsf{exit}(\ldots)$$

where "call $\mathfrak{pr}$" is a recursive call, are common at this stage. The first observation we can make here is that the region inference has caused the call to cease being a tail call. But what is worse is that each recursive invocation of the predicate has its own instance of the r17 region. All of these regions are deallocated simultaneously when the innermost recursive instance eventually returns. If each r17 contains only a few words (which is common), there is a lot of memory to save by merging all these regions which are already deallocated at practically the same time.

This calls for an *inter-procedural* region merging transformation. We can get that almost for free by the following trick:

[8] To be precise, the corresponding makeregion should also be removed. In the representation we work on in this phase, all makeregion instructions are not present but implicitly supposed to be clustered at the beginning of each clause, so the removal of the corresponding makeregion instruction is also implicit.

Before the real region merging transformation, we do an auxiliary pass over the program that inserts an extra, "artificial" region parameter at the beginning of every region parameter list. At the same time, every

(...)
  & call pr $[\rho_1, ..., \rho_{\lceil pr \rceil}]$ $(x_1, ..., x_{\lceil pr \rceil}) \rightarrow (x'_1, ..., x'_{\lfloor pr \rfloor})$
(...)

gets replaced with

(...)
  & call pr $[\tilde{\rho}, \rho_1, ..., \rho_{\lceil pr \rceil}]$ $(x_1, ..., x_{\lceil pr \rceil}) \rightarrow (x'_1, ..., x'_{\lfloor pr \rfloor})$
  & killregion $\tilde{\rho}$
(...)

(which also entails an implicit "makeregion $\rightarrow \tilde{\rho}$" at the beginning of the clause) where $\tilde{\rho}$ is a fresh region register.

The dump *name*`.dump.b5` shows the program after the insertation of extra region parameters.

After this operation, we know that the first region parameter of every predicate always gets deallocated immediately after it returns. Therefore we can let the region merge transformation merge a local region with that region parameter if there are no call instructions after the killregion instruction; and the new $\tilde{\rho}$s surrounding the call instructions partake in the region merge phase.

We do not allow anything to be merged with the region parameter in clauses where there are allocating instructions after the last call instruction, lest the lifetime of a local region at the bottom of a stack of recursive calls gets prolonged past arbitrarily many allocations.

The dump *name*`.dump.b6` shows the program after the region merging.

10.3.2 Removal of unused region parameters

Now we remove all region parameters that are not (directly or indirectly) used for allocations. Apart from reducing the overhead of passing around unnecessary region parameters[9], it also puts a finishing touch to the region merge transformations (by removing those of the "artificial" region parameters that turned out not to be used for anything) and prepares for the makeregion placement (which has to assume that a predicate may use any of its region parameters to allocate something).

This transformation is similar to the one we described in Section 10.2.1—in fact our implementation uses the same code for both phases. The difference is how regions that cannot be removed are defined:

- A region parameter cannot be removed if it is mentioned by a builtin, construct, or makevar instruction. (In Section 10.2.1, a region parameter was prevented from being removed just by a value being read from the region).

[9]The ML Kit does a similar optimization, which is documented by Birkedal et al. [1996] under the heading "Removal of get-regions". For unknown reasons, the optimization is described as being part of the multiplicity analysis...

- A region parameter cannot be removed if it is used by a call statement to initialise another region parameter that cannot be removed.
- Any other region parameter can be removed.

The dump *name*`.dump.b7` shows the program after regions parameters have been removed.

10.3.3 Placing makeregion instructions

The final set of transformations in our region inference place makeregion instructions at appropriate places in the program.

The first step is mostly technical in nature. Our implementation of RP assume that the `main` predicate has neither value nor region parameters. However, the region merge phase has inserted a region parameter for `main`. If it has not been removed by the the second unused-parameter removal we need to create a new main predicate which creates an appropriate region and calls the old one:

```
new-main[] :- enter → ()
            & makeregion → reg
            & call old-main [reg] () → ()
            & killregion reg
            & exit ().
```

The dump *name*`.dump.b8` shows the program after this step.[10]

The second and final step actually inserts makeregion instructions into each clause. The makeregion is inserted immediately before each mention of each region register which is not a formal region parameter. If the first (and only) mention of a region register happens to be its killregion instruction, the region register is removed entirely.

[10] If you actually inspect the `b8` dump of a program where `main` replacement was necessary you'll find that the makeregion instruction is still implicit and that the exit is actually a fail. The latter is as good as exit because the boundary conditions for xP programs (Section 2.4) already say that if the initial call of `main` ever returns it immediately backtracks.

Chapter 11

Experimental results

In this chapter we report some experimental findings with our prototype of a region-based Prolog implementation. The components that make up the prototype have been described in Chapters 8, 10, and 5.

It is well known that programming style has a large influence on the space efficiency of a program with a region-based implementation. We have not performed systematic experiments that investigate this influence. The effects are already relatively well-understood for them ML Kit (many of them are described by Tofte et al. [1997]) and our system is sufficiently close to the one used in the ML Kit that we are certain that this understanding is also applicable to Prolog.

Another reason for not conducting an extensive study on the interaction between programming style and our prototype is that the prototype does not really represent the state of the art in efficient region usage. In particular, our prototype has problems with loops such as

```
loop(Data) :- process(Data,Newdata),
              ( finished(Newdata) -> true
              ; loop(Newdata)
              ).
```

which it implements as

```
loop(Data) :- process(Data,Newdata),
              auxiliary(Newdata).
auxiliary(X) :- finished(X), !.
auxiliary(X) :- loop(X).
```

The problem here is that the region(s) where `Newdata` live must be allocated in `loop`. Because `Newdata` is used by `auxiliary`, the region can only be deleted after the call to `auxiliary` which is to say after the entire loop has ended. Thus with our current prototype, code such as this will build a big pile of intermediate results that are only deallocated after the entire computation terminates.

This **tail recursion problem** occurs also in Tofte and Talpin's system for ML. The known implementations of that system (including the prototype used by Tofte and Talpin in 1993 and the ML Kit) have all circumvented the

problem by using a **storage-mode analysis** [Birkedal et al. 1996; Tofte et al. 1997]. The storage mode analysis works by using the same region for `Data` and `Newdata` but arranging for the region to be **reset** (*i.e.*, entirely deallocated and then reallocated) at an appropriate point during the execution of `process`.

However, due to time constraints our prototype does not contain a storage-mode analysis. We judge that the principles behind the storage mode analysis are entirely compatible with the extensions to the region model we have made to accommodate Prolog's special characteristics, so it should be possible (given time to do the work) to use storage-mode analysis in a region-based Prolog implementation.

Therefore, we feel that it would not be fair to use our prototype to assess which constraints region-based memory management puts on Prolog programming style.

What we *can* do, however, is to compare the performance of region-based memory management to the performance of other memory management paradigms for programs where the absence of a storage-mode analysis does not handicap our prototype.

The goal of such a comparison is to try to refute the pessimistic hypothesis that region-based memory-management primitives are so much slower than WAM-like stack allocation or garbage collection that it is futile to try to develop region-based memory management for Prolog further.

Earlier experience shows that region-based management does not incur prohibitive time costs in ML [Birkedal et al. 1996]. However, this experience does not necessarily carry over to our Prolog setting, because we have needed to add extra administrative burdens to the region-based memory-management primitives so that they support backtracking.

11.1 The reference implementations

It would not be conclusive to compare the performance of our prototype against an existing production-quality Prolog implementation. The prototype uses very naïve techniques for most tasks that do not concern memory management, so if the comparisons showed that it was slower than an existing Prolog implementation we would never know whether the reason was region-based memory management or some unrelated optimization used by the other Prolog implementation.

Therefore we compare performance against two reference implementations that we have derived from our prototype. The **WAM-like reference implementation** uses pure stack management for the heap (and is thus not well suited for long non-backtracking computations). It is meant to be representative of non-garbage-collecting Prolog implementations such as Visual Prolog [Prolog Development Center 2000]. The **garbage-collecting reference implementation** is meant to be representative of Prolog implementations such as BinProlog [Demoen et al. 1996] or SICStus. It uses WAM-like

heap management as far as possible but resorts to garbage collection if the heap approaches a maximal size selected by the experimenter.

Either of the reference implementations consists of

- The same Prolog-to-P translator as we use in the region-based implementation (described in Chapter 8).
- A translator from P to C which is exactly the same as the region-based implementation's translator from RP to C, except that when applied to P code it of course does not create code for region manipulation.

  The C code generator always emits special statements to support the garbage-collecting reference implementation (especially to help maintain a minimal "root set"). We use the C preprocessor to remove these special instructions when the C code is compiled for use with the WAM-like reference implementation or the region-based implementation.
- A run-time module (written in C) which replaces the region-based run-time module we described in Chapter 5.

  The WAM-like reference implementation's run-time module is very simple: it allocates memory by extending the heap and never deallocates anything except at backtracking, where the top-of-heap pointer is reset to a value stored in the choice point.

  The garbage-collecting reference implementation's run-time module is similar, except that it starts a garbage collector when the heap threatens[1] to grow too big. We have implemented a simple two-space copying garbage collector. The garbage collector is interfaced with the stack-like heap manager such that heap words allocated *after* the collection can still be deallocated quickly by backtracking. The words that survive the collection can only be removed by the next collection[2].

  Each run-time module (including the one in the region-based implementation) can be compiled with or without statistics collection enabled. We actually run each test case twice: once with statistics disabled, to find running-time figures, then once with statistics enabled, to find precise values for the maximal heap size and the number of various operations performed

With this design we feel reasonably confident that any observed difference in behaviour between our region-based prototype and the reference implementations will be due to the different heap-management strategies.

[1] The P-to-C translator inserts heap-check operations at the start of every block of straight-line code which contains allocation instructions. Each heap-check operations checks that there is enough free space on the heap to accommodate the longest possible sequence of allocation instructions (whose length has been determined statically by the P-to-C translator).

[2] This is not quite the state of the art—Demoen et al. [1996] and others have described garbage collectors which do not interfere with WAM-like deallocation of collected words. However, Bevemyr and Lindgren [1994] have found that in practise the amount of deallocation enabled by this tends to be small.

The reference implementations can be obtained as part of the region-based prototype implementation [Makholm 2000].

11.2 Experimental procedure

The timing experiments were carried out on the machine thor.diku.dk which (according to the computing department) is a HP Apollo 9000/735 workstation with a 99 MHz PA-RISC1.1 processor, running HP-UX 10.20.

Test programs (generated C code as well as run-time modules) were compiled with GCC 2.8.1 with optimization level `-O2`. We used `/bin/time` to time them with their standard output redirected to `/dev/null`. Each running-time figure is the median of the "user time" reported in 5 consecutive runs.

All memory usage measurements are given in machine words. The word size on the test machine is 32 bits.

The Prolog sources for the test programs, as well as `puzzles` file used as input for the `puzzle.pro` example, are included with the prototype implementation [Makholm 2000].

11.3 Shallow backtracking search

Our first set of test programs are meant as examples of programs that do not really need region-based memory management, because their memory behaviour in the stack-based WAM model is acceptable. This is true for two classes of programs: programs that do a backtracking search over a relatively shallow search space, and programs that do few enough allocations that a monotonically growing heap is not a problem. The latter class is not interesting in our experiment, because either they terminate so quickly that comparisons are impossible, or their running times are dominated by other effects than memory management.

We have measured the performance of two programs that do shallow backtracking searches.

`10queens.pro` finds all solutions to the "10-queens problem", which is the familiar problem of placing eight mutually nonattacking queens on a chessboard, extended to 10 queens and a 10×10 square chessboard. The purpose of the extension is to make the running time larger, compared to the 0.1 second resolution of the timing measurements.

The searching code is taken from Bratko [1990, page 117], but has been adapted to fit into the Prolog subset supported by our prototype implementation and generalised to allow easy adjustments to the size of the problem. We added an output routine which outputs each answer using P's primitive I/O built-ins.

`puzzle.pro` finds solutions to cryptoarithmetic addition puzzles such as "find different decimal digits to substitute for each of the letters in

	10queens.pro			puzzle.pro		
	GC	WAM	regions	GC	WAM	regions
Running time (s)	19.0	18.3	20.0	7.1	6.1	7.3
Max heap used		347	162		20103	3903
Max heap size		347	368		20103	4512
Max stacks size		197	307		683	681
Max regions alive			12			13

Figure 11.1: Measurements from the "shallow backtracking search" experiments. The "Max heap used" row shows the maximum number of words that the client program has allocated on the heap at any time. The "Max heap size" show the maximum actual heap size, inclusive of region management data and unused space in pages. For the WAM-like reference implementation, these two figures are always the same. The "Max stacks size" is the sum of the maximum size of each of the auxiliary stacks used by each implementation model. They include the local stack, the choice-point stack, the snapshot stack (for the region-based implementation), and the trail (for the WAM-like implementation).

```
  SEND
+MORE
------
 MONEY
======
```

such that the addition is correct".

The search code is taken from Barker [1999]. We added an output routine to format the solutions nicely, and an ad-hoc parser which reads problem descriptions from the input stream.

The input for the experiments is a file `puzzles` containing contains 8 different problem instances, most of which are also from Barker [1999]. Each problem instance is repeated 4 times to give larger running times.

The results of the experiments are tabulated in Figure 11.1. We see that region-based memory management causes a slowdown of 10 to 20 percent compared to the WAM-like implementation and about 5 percent compared to the garbage-collecting implementation. A garbage collection is never actually triggered in either of the programs, but the extra work of checking for heap overflow and maintaining a minimal root set make the garbage-collecting implementation use more time than the pure WAM-like one.

The table of results also lists key memory usage figures. The example programs are too small to draw any firm conclusions, but we observe that even though the region-based manager uses over half of the heap for administrative data in `10queens.pro`, the actual space need remains comparable to that of the WAM model, because the region model can free intermediate data before backtracking occurs.

In the `puzzle.pro` example, the region-based memory model's large improvement in space efficiency is due to the fact that the test program's main

loop is tail recursive such that the WAM-like model never gets to free the intermediate data allocated while parsing the problem descriptions. The decrease in the stacks size for this example is due to the trail in the WAM model which is superseded by administrative data on the heap in the region-based model.

11.4 Longer functional computations

Our second set of test programs are examples of programs that backtrack so seldom that it is not realistic to use the pure WAM model for them at all.

We have three examples in this category.

`ack.pro` computes ackermann(3, 8) using the standard recursive definition of Ackermann's function. The result is 2045.

This example is selected for being extremely friendly to the garbage collector. During the computation, eight million heap words are allocated one by one, but each time a garbage collection is triggered at most 5 words are live. The running time of the copying garbage collector in our reference implementation is proportional to the amount of live data rather than to the total size of the heap. Therefore one would expect garbage collection for this example to be as good as cost-less.

On the other hand, in the region-based model the average number of words ever allocated in each region is 2, meaning that the complex *makeregion* and *killregion* operations are performed often.

Tofte and Talpin [1993] also used Ackermann's function as a benchmark, but with a point totally different from ours. In their implementation a major consumer of memory was auxiliary pair objects that contained the arguments for each call of the `acker` function, because their language only allowed one argument to each function. This problem does not exist at all in our context, because Prolog (and P) natively supports multiple arguments to a predicate.

`quick.pro` sorts a list of 20000 pseudorandom numbers[3] using a list-processing implementation of quicksort. Quicksort is a classic benchmark for region-based memory management, used since Tofte and Talpin [1993] who proudly reported that they could sort a list of 5000 numbers "in less than five times the memory needed to represent the list". If our test program is changed to sort 5000 numbers it indeed uses approximately 5 times the memory needed to represent a list of 5000 numbers (a little less if only the payload data in regions is counted—which matches Tofte and Talpin's way of counting—a little more when region-management data is also counted).

We sort 20000 numbers rather than 5000 numbers because 5000 numbers can be sorted in 1.2 seconds which we find is too little given

[3] A built-in predicate `random` has been added to the implementations. It uses the C library function `lrand48()` to produce the same sequence of pseudorandom numbers between 0 and $2^{24} - 1$ in all implementations.

	ack.pro		quick.pro		filerev.pro	
	GC	reg	GC	reg	GC	reg
Running time (s)	19.3	18.3	6.3	5.7	⊥	1.3
Max heap used		2054		287686		94120
Max heap size	32736	32736	307584	307584	108928	108928
Max stacks size	6152	12288	60020	80029	2283	3658
Max regions alive		2046		56		726

ack.pro			quick.pro			filerev.pro		
heap	#coll.	time	heap	#coll.	time	heap	#coll.	time
98208	170	19.3	922752	6	5.5	326784	10	2.0
65472	255	19.2	615168	10	5.7	217856	38	3.0
32736	510	19.3	307584	30	6.3	200000	67	4.0
16374	1021	19.6	233802	54	6.9	190000	119	5.8
8194	2042	20.2	196910	97	8.2	179562	467	17.8
14	∞?	∞?	160014	11427	359.9	108928	⊥	⊥

Figure 11.2: Measurements from the "longer functional computation" experiments. In the main comparison for each of the examples, the garbage-collecting implementation's heap size limit has been set to equal the maximum heap size needed by the region-based implementation. The ⊥ entries for the `filerev.pro` experiment means that the garbage-collecting implementation could not work at all in that little memory.
The lower table shows running times and number of collections for the garbage-collecting implementation with different heap sizes.
The heap sizes for the garbage-collecting implementations is measured as the sum of the sizes of the semispaces.

that `/bin/time`'s precision is 0.1 second. Interestingly, when sorting 20000 numbers the quotient between the memory usage of quicksort with regions, and the memory needed to store the list, *drops* from 5 to about 3.7.

`filerev.pro` reads in a file from the input stream, storing it as a list of lines. Then it reverses the list using the naïve quadratic-time reverse predicate of Figure 2.1 (page 22) and outputs the reversed list.

This example is selected to be extremely hostile to the garbage collector. Each time the heap is collected, the contents of all the lines has to be traced and copied. Region-based memory management, on the other hand, only needs to manage the memory used for cons cells in the intermediate lists of lines used by the reverse predicate.

As input to `filerev.pro` we use the output of the `10queens.pro` example. It consists of 724 lines totalling 21720 characters.

The results of the experiments are shown in Figure 11.2. With the garbage-collecting implementation, the running time depends on how big a

heap we allow. We think the most fair comparison is to give the garbage collector the same amount of memory that the region-based memory manager needs. The running times in the upper part of Figure 11.2 have been measured with this convention.

We have also measured running times of the garbage-collecting implementation for other heap sizes, shown in the lower part of Figure 11.2. The heap sized used have been selected as

- One, two, and three times the amount of memory needed by the region-based implementation.
- The smallest heap in which it is possible to run the example program. For heap sizes 12, 160012, respectively 179560, the examples die with an error message from the garbage collector that the heap cannot be compacted enough.
- The arithmetic mean between the "what the region-based implementation needs" and "smallest possible heap", and the arithmetic mean between that and the "smallest possible heap".

We are surprised to observe that the region-based implementation is faster than the garbage-collecting one in all three experiments—even `ack.pro` which was selected specifically to throw the odds in favour of garbage collection. The `quick.pro` example *can* be faster with garbage collection, but only if the garbage collector is allowed to use more than twice the memory needed by the region-based implementation.

Our only explanation for the behaviour of the `ack.pro` example is that the region-based version incurs fewer cache misses, because the region-based memory manager uses the most recently freed page when reusing memory, whereas the garbage collector always allocates the least recently used word available to it.

An alternative hypothesis is that the garbage-collecting `ack.pro` spends the extra time with procedure-call overhead, in the calls to the run-time module that check whether a garbage collection is due. This hypothesis can be refuted by inspecting the generated C code: The region-based version makes just as many calls to the *makeregion* and *killregion* primitives.

The `filerev.pro` example could not run with garbage at all collection without having more memory available than the region-based version needs. The reason for this is that the copying garbage collector needs all of the live data to fit in a semispace, so the total heap size must be at least twice the maximal amount of live data.[4]

We warn that the good results of these experiments must not be taken as evidence that region-based memory management *always* outperforms

[4]It has been debated whether it is fair to count both semispaces when measuring the space usage of a copying garbage collector. In the context of our experiments, however, we think that there can be little doubt that the most fair results come from counting both semispaces. The machine we use for the experiments has much more physical memory than any of the examples use, so the running times we compare assume that a garbage collection can be completed without swapping.

garbage collection. We have consciously selected the test examples so that they work reasonably well with the restricted region model we have implemented.

11.5 Conclusion

Our experiments have established that region-based memory management for Prolog is *not* inherently less efficient than garbage collection. Our results indicate that for programs where region-based memory management does not lead to catastrophic space usage behaviour, the region-based execution model is quite competitive compared to garbage collection.

The experiments also indicate that the region-based model is not prohibitively expensive compared to a purely stack-based model like the WAM. The cost of region-based memory management in that context is however not totally negligible, and it would probably pay to be able to switch to a more WAM-like memory use pattern in those parts of a program that uses backtracking search.

Chapter 12

Conclusion

The stated goal of this project is to investigate the hypothesis that

> *Region-based memory management can work just as well in Prolog as it works in Standard ML in the ML Kit today.*

To do this, we have shown how to reason about the memory use of Prolog programs by translating it to a simpler and more explicit intermediate language P (Chapters 2, 3, 6, 7, and 8). We have proposed a way to combine region-based management with backtracking and logical variables (Chapter 4) and developed algorithms for supporting it at run-time (Chapter 5). We have adapted existing theory and algorithms for automatically creating region annotations to our model (Chapters 9 and 10). Finally, we have produced a prototype implementation of these ideas and used it to compare the time efficiency of region-based memory management to other memory-management strategies (Chapter 11).

We have not done rigorous proofs that our techniques work as claimed, but we have provided strict semantic definitions of many key concepts and suggested how proofs of their most important properties could be structured.

Our work does not provide fully conclusive experimental evidence for the main hypothesis, because time constraints have prevented us from implementing a region system as advanced as that of the ML Kit. It does, however, constitute strong circumstantial evidence that the hypothesis is valid. We have not uncovered any inherent reasons why it would not be possible to add the missing features to our model; indeed our expectations are that such a task would be fairly routine.

We therefore feel justified in concluding that the project has met its goal.

12.1 Directions for further work

We now suggest a number of possible areas in which this work can be extended. They range from mundane tasks such as extending our system to handle more examples of real-life Prolog code, to problems and shortcomings with region-based memory management in general which it will require new creative research to solve satisfactorily.

12.1.1 The tail recursion problem

The **tail recursion problem** is the one described at the beginning of Chapter 11. In general terms the problem is that even though a parameter to a procedure (predicate, function) may only be used for a fraction of the time the entire procedure executes, the regions where the parameter lives can only be deallocated after the procedure terminates. The problem is not restricted to tail-recursive calls but shows itself most prominently when it prevents a tail-recursive procedure from having its expected constant space usage.

The storage-mode analysis in the ML Kit [Birkedal et al. 1996] and the enhancements proposed by Aiken et al. [1995] both provide partial solutions of this problem. We judge it would be relatively easy to add either or both of them to our RP model.

However, even the combination of the two partial solutions does not solve all instances of the problem. We are currently participating in a project also comprising Henning Niss (at the University of Copenhagen) and Fritz Henglein (at the IT University in Copenhagen), which promises to yield an elegant general solution to the tail recursion problems, superseding storage-mode analysis as well as Aiken et al.'s model. This solution will also be applicable in our Prolog context.

12.1.2 Special handling of small regions

As we described in Section 5.1.3, our run-time implementation of regions works best with regions that grow big. In practise, the region inference decides on many regions that are only used for a single or a few allocations. For these regions, the list-of-pages implementation wastes a lot of space.

The ML Kit contains a **multiplicity analysis** [Birkedal et al. 1996] which identifies provably **finite** regions, that is, regions that are only used for at most one allocation at run time. Space for objects allocated in finite regions are allocated directly on the stack, without any management overhead. Tag bits in region identifiers are used to allow a procedure to allocate its result in either a finite or an infinite region, according to which kind of region it is given as a region parameter.

A similar scheme could be used in our Prolog implementation. Finite regions would be allocated in the environment frames on the local stack. That would somewhat complicate the layout of environment frames, but not insurmountably.

It would also incur extra complexity to arrange for trailing of variables allocated in finite regions. With our current region inference it would not be necessary, however. An uninstantiated variable in a finite region would be the only thing that lived in the region at all, so there would never be anything to instantiate it to; thus it would not need trailing. (It would not need existing at all, for that matter).

It would be nontrivial (in our system as well as in the ML Kit) to apply multiplicity analysis *and* Aiken et al.'s partial solution for the tail recursion

problem at the same time. Storing finite regions on the call stack depends on each region being allocated and deallocated by the same procedure, which is precisely the constraint Aiken et al. propose removing. The net effect of combining both ideas would become similar to individual allocation and deallocation, except that the deallocation points would be automatically inferred. An auxiliary heap with a conventional memory manager would be necessary.

12.1.3 Support for a larger subset of Prolog

We have described region-based memory management for a small and reasonably pure Prolog subset. If region-based memory management is to be of any value in practise, it needs to support a much larger selection of the extra-logical features that Prolog programmers expect to have available.

We now list some of the more interesting features our subset is missing, and estimate what would be needed for supporting them in a region-based implementation.

- `var/1` and `nonvar/1`. These predicates test whether or not a term is an uninstantiated variable. They can already be easily expressed in P using the deref instruction.
- `atom_length/2`, `atom_concat/3`, `atom_chars/2`, and similar predicates that allow atoms (nullary functors) to be used as strings.

  We do not think region-based memory management add complications to the way these features are usually implemented. For our type-based analysis passes we would need to add a way to specify "any nullary functor" in the type systems.

  Region-based memory management might make it a worthwhile option to store the strings that identify each functor on the heap instead of in a separate "symbol table". The ML Kit can represent arbitrarily long strings efficiently in a region composed of constant-size pages. A similar technique could be used here.
- `functor/3`, `arg/3`, and `=../2`. These predicates allow constructing and analysing structures using a dynamically-determined atom as the functor and a dynamically-determined number of arguments.

  Once the heap representation of structures has been changed to allow such dynamically-built structures (which it must regardless of whether regions are used or not) it should be possible to add these constructions to P and RP.

  They are fundamentally hostile to type-based analyses, however, and while a well thought-out type system might be able to resolve some cases to more benign constructs, it would still need to have a conservative fall-back option, which in turn could have devastating effects on the region inference's precision.
- `call/1`. This is Prolog's equivalent of "eval", a predicate that takes a term and interprets it as code to execute. Such constructions are

necessarily hard to reason about statically and hard to implement in a compiler. Advanced type and mode analysis techniques might be able to convert some cases of `call/1` to more benign primitive constructs, but any compiler (region-based or not) will probably always need a very conservative (and inefficient) fall-back option to duplicate the intended interpretative semantics.

- `assert/1` and `retract/1`. These predicates allow a program to modify itself by adding and removing clauses in its Prolog source. In general, this is of course terminally destructive to any efficient implementation technique, region-based or not.

  Many programs, however, restrict their use of these predicates to manipulating simple facts (that is, clauses without subgoals and without variables) for a small statically-determined set of predicates. This use implements what is known as the **database**, a global, updateable storage area that is persistent to backtracking.

  Region-based memory management not only tolerates the database but can actually help optimize its implementation. Implementations that reclaim heap memory when backtracking need to copy asserted facts to a separate "persistent heap" lest they get deallocated by backtracking before a database query that retrieves them. In a region-based implementation the region inference may arrange for every value that might end up being used in an asserted fact to be allocated on the persistent heap from the beginning, modelled by a special pseudo-region. That way time-consuming copying of asserted facts can be avoided.

  (This optimization works in the common case that the asserted fact does not contain any uninstantiated or trailable variables. A mode analysis such as our TP type inference from Section 8.8.1 can be used to identify when this applies and use a copying implementation of `assert/1` otherwise).

- `findall/3`, `bagof/3`, and `setof/3` which create lists of the results that a programmer-specified piece of code produces when it is forced to backtrack after delivering each result.

  Again, region-based memory management may optimize the implementation of these in the common case where the results cannot contain uninstantiated or trailable variables. Conventional implementations have to copy each result out of the heap so that it will not be destroyed by the backtracking that follows. A region-based implementation may simply (with some cooperation from the memory manager) exempt the region(s) containing the result from shrinking in that backtracking operation, thus still deallocating intermediate results that the computation may have left in other regions.

- *Blocking goals* are supported in some Prolog implementations. They are goals that are only executed immediately if "enough" of their arguments is instantiated (for some programmer-specified definition of "enough"). Otherwise they lie dormant until enough instantia-

tions have been performed, at which point they figuratively insert a call to themselves in front of whatever code happens to be executing.

It appears to be very difficult to reason about when a blocked goal might suddenly wake up and need the arguments that were passed to it to still exist. We do not know whether it is possible at all to combine blocking goals and region-based memory management.

12.1.4 Programmer feedback from the region inference

Experience with the ML Kit shows that it is generally hard for programmers to predict how region inference reacts to their code. Minor changes to the code may often result in dramatic improvements of the space behaviour of the programs, so it is important for the programmer to get feedback about how her program got annotated.

With our current prototype the only way of getting this feedback (save for simply observing how much memory the translated program happens to use) is to inspect the RP code produced by the region inference phase. Because P and RP look quite different from Prolog, this option is probably not attractive to most Prolog programmers.

It would therefore be interesting to develop a way to present the results of region inference in the context of the original Prolog program, or to develop a region inference algorithm which works on a representation closer to Prolog.

12.1.5 Estimation of memory-management overhead

The additions to the region-based model we made to support backtracking had the consequence that not all of the memory management primitives run in constant time. It would be desirable to have a static analysis that were able to pinpoint those primitive operations which risk taking arbitrarily long time.

12.1.6 Separate compilation

We have expressed our techniques as if the entire Prolog program was present right from the beginning. In practise, a region-based implementation would be much more convenient if it supported **separate compilation** of program modules.

The ML Kit can do region inference during separate compilation [Elsman 1999], so we do not think that it will be a problem to do the same for P.

It will not be as easy to translate Prolog to P with separate compilation. Programmer-supplied mode annotations for the predicates exported by each module will most probably be necessary.

12.1.7 Integration of garbage collection with regions

Not all programs react well to pure region-based memory management. Consider, for example, a program which creates a list containing a lot of data and only afterwards decides which few elements of the list it is going to need for the remainder of a long computation. With region-based memory management, the program would need to allocate all of the elements in the same region and keep that region alive during the entire computation.

Now, one could simply conclude that such a program is not suited for region-based memory management and use a garbage-collecting implementation to run it instead. But regions could still be a useful memory-management principle for other pieces of data in the program.

Tofte et al. [1997] suggest that such program be rewritten to create fresh copies of those elements that are eventually chosen to be long-lived. The fresh copies can be made in a separate, long-lived region, and the region containing the original list elements can be deallocated.

We think that another strategy is more likely to appeal to programmers (which in reality is as pragmatically important as efficiency and elegance, if not more): that garbage collection can be used for some data and region-based memory management for other data. For example, the programmer should be able to specify that all data that needs to be alive at some specific point in the program's main loop must be garbage collected; other intermediate data can be allocated in regions.

Region inference could easily be extended to accept such annotations. It might be challenging, however, to design a garbage-collection strategy that could co-exist peacefully with a region inference that allows dangling pointers.

12.1.8 Precision problems with type-based analyses

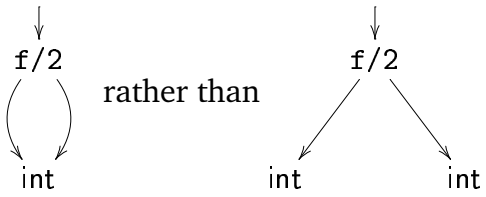
During the preparations for the experiments in Chapter 11 we discovered a strange behaviour of the algorithms we use to compute TP and TRP typings. Consider, for example, this little program:

```
foo(X,Y) :- Z=f(X,Y), W is X + Y.
bar(T) :- TT is T + T.
main :- P=3, foo(P,P),
        Q=3, R=5, foo(Q,R),
        bar(R).
```

Most surprisingly, our region inference elects to place `Q` and `R` in the same region—even though `Q` could easily have been deallocated before the call to `bar/1` and it is easy to exhibit a TRP typing that proves this!

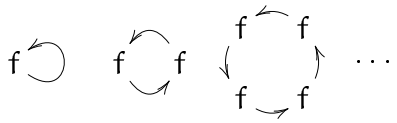
The problem is that our TRP type inference begins with a monomorphic region-less type inference using the standard unification-based algorithm. In the first call to `foo/2` means that the types of `X` and `Y` get unified, so `Z`'s type

becomes



After the region-less type inference a fresh copy is made of each of the program's types in preparation for region annotations—but the "fresh copy" of `Z`'s type has the same structure as its original, so it has only room for a single region annotation on the `int`.

In this particular case the problem could have been avoided by specifying that type nodes must not be shared internally in each fresh copy. That would, however, not be possible for recursive types. If instead we require the fresh copies to be the "least shared" representation we encounter the problem that recursive types have no least shared representation either. For example, the type `f(f(f(f(...))))` has a chain of less and less shared representations



which has no limit.

The ML Kit does not have this problem because the polymorphic type inference of ML shields the types inside `foo` from being influenced by irrelevant sharing among the actual argument types at the call sites.

Nevertheless other type-based analyses that are based on annotating an already-existing type assignment of the program[1] are liable to related problems. The general problem is how to find a representation with "less enough" sharing for a type-based analysis to be maximally precise. We feel that this problem recurs often enough that it ought to be investigated in a more general context than a specific analysis, but we are not sure what a suitable formal generalisation would be.

12.1.9 The list problem

Figure 12.1 shows the Prolog code for a simple expression evaluator. The clause of `eval/3` that implements a `let` expression adds an element to the environment in which `Exp2` is evaluated. After `Exp2` has been evaluated, the new element is clearly not necessary anymore.

The **list problem**, which occurs in our region inference algorithm as well in that of the ML Kit, is that the extra element does *not* get deallocated immediately. Rather, all elements ever added to any environment get allocated in the same region which is deallocated only in the outermost `evaluate/2`.

The cause of the problem is that region-annotated type systems such as TRP or the Tofte–Talpin system think of the type of `Env` as "a list of pairs that

[1] Such as, for example, the binding-time analysis of Makholm [1999].

```
lookup(X,[(X,V)|_],V).
lookup(X,[_|L],V) :- lookup(X,L,V).

eval(var(X),Env,V) :- lookup(X,Env,V).
eval(plus(Exp1,Exp2),Env,V)
  :- eval(Exp1,Env,V1),
     eval(Exp2,Env,V2),
     V is V1 + V2.
eval(let(X,Exp1,Exp2),Env,V)
  :- eval(Exp1,Env,V1),
     eval(Exp2,[(X,V1)|Env],V).

evaluate(Exp,V) :- eval(Exp,[],V).
```

Figure 12.1: A simple expression evaluator

are allocated in ρ" for some ρ, which means that all of the pairs have to be in the same region. The type systems does not support such a notion as "a list of pairs allocated in various regions".

Originally we intended that this report would describe an advanced region type system "TRP-2" which would solve the list problem. It was not included in the report due to time constraints, but we still think the basic idea is sound. TRP-2 was to be a combination of TRP and TP, such that a TRP-2 type would have the shape "$\rho.\mathfrak{m}.\tau$". A subtyping rule similar to the one we describe in Section 7.4 would be used for TRP-2 types, but it would also allow the region annotations in a type to change, guided by a global partial order on region register names. The relation "$\rho_1 \trianglelefteq \rho_2$" would intuitively mean, "whenever the regions ρ_1 and ρ_2 both exist, ρ_2 is going to exist at least as long as ρ_1". Then the type system could express such a type as "a list of pairs each of which will exist at least as long as ρ does". The `eval/3` rule for `let` could let ρ in this type be a local region where it allocated its extra element.

We think it would be interesting to see this idea developed in practise.

Appendix A

Some mathematical digressions

In this appendix we present some mathematical arguments used to argue for the the soundness of the design of P when unification is allowed to produce infinite (or regular, or circular, according to one's favourite nomenclature) terms.

A.1 Infinite terms

We assume the notion of **terms** built with a certain selection of function and constant symbols (and possible variable symbols) is known.

The most popular way to represent terms in the computer is as trees: each subterm is represented by a piece of memory containing a symbolic representation of a function symbol and pointers to the representations of its subterms. This idea is also well known.

Now, if one of the pointers are redirected to point to a node that represents a superterm of the original subterm, a **circular** term has been created. Such a structure does not correspond to a "term" according to the mathematical meaning of that word, but it is natural to view it intuitively as a representation of an infinitely deep term tree.

We now present our own little theory of machine representations of certain infinite terms. The end product is isomorphic to what various people have been calling *regular trees*, but the details have been chosen to support the rigorous proofs later in this appendix.

The central concepts of this section are those of a **termlike algebra** and of the **graph algebra**.

A.1.1 Signatures and algebras

For the sake of generality we parameterise the theory on a (many-sorted) **algebraic signature**. What this means is probably known to most readers, but to set notation straight we summarise the main definitions and properties here. Most of the definitions are from Mitchell [1996, Chapter 3], but many textbooks on programming language semantics contain similar material.

Definition A.1 *A **signature** $\Sigma = (S, F)$ consists of a set S whose elements are called **sorts**, and a a collection F of pairs $(f, s_1 \ldots s_k \rightarrow s)$ with $k \geq 0$, $s_1, \ldots, s_k, s \in S$, and no **function symbol** f occurring in two distinct pairs.*

In the sequel a fixed but arbitrary signature $\Sigma = (S_\Sigma, F_\Sigma)$ is assumed given except where something else is explicitly stated.

We shall sometimes assume that Σ is single-sorted (*i.e.*, that S_Σ is a one-element set) when it is notationally convenient and the generalisation to many-sorted signatures is obvious.

Definition A.2 $\mathsf{Sr} : F_\Sigma \rightarrow S_\Sigma$ *is the function defined by*

$$\mathsf{Sr}(f, s_1 \ldots s_k \rightarrow s) = s$$

Definition A.3 *A **Σ-algebra** $\mathcal{A}$ consists of an S_Σ-indexed family of sets A^s called **carriers**, and an **interpretation** map assigning a function*

$$f^{\mathcal{A}} : A^{s_1} \times \cdots \times A^{s_k} \rightarrow A^s$$

to each function symbol $f : s_1 \ldots s_k \rightarrow s \in F_\Sigma$. (When $k = 0$, $A^{s_1} \times \cdots \times A^{s_k}$ is taken to be some canonical one-element set).

Definition A.4 *A **homomorphism** h from Σ-algebra $\mathcal{A}$ to Σ-algebra $\mathcal{B}$ is a S_Σ-indexed family of functions $h^s : A^s \rightarrow B^s$ such that*

$$h^s(f^{\mathcal{A}}(a_1, \ldots, a_k) = f^{\mathcal{B}}(h^{s_1}(a_1), \ldots, h^{s_k}(a_k))$$

for every function symbol $f : s_1 \ldots s_k \rightarrow s$.

Definition A.5 *An **isomorphism** is a homomorphism h where the functions h^s are all bijections. Clearly, the inverse of a isomorphism is also an isomorphism. When an isomorphism from $\mathcal{A}$ to $\mathcal{B}$ exists, $\mathcal{A}$ and $\mathcal{B}$ are called **isomorphic**.*

Definition A.6 *A Σ-algebra $\mathcal{A}$ is **initial** if for every Σ-algebra $\mathcal{B}$ there is exactly one homomorphism from $\mathcal{A}$ to $\mathcal{B}$.*

Theorem A.7 *Any two initial algebras are isomorphic.*

Definition A.8 *The **term algebra** $\mathrm{Term}(\Sigma)$ is the algebra where the $\mathrm{Term}(\Sigma)^s$ are sets of strings of function symbols defined by*

- *For each $(f, s_1 \ldots s_k \rightarrow s) \in F_\Sigma$, whenever $w_i \in \mathrm{Term}(\Sigma)^{s_i}$ for $1 \leq i \leq k$, $f w_1 \ldots w_k$ is in $\mathrm{Term}(\Sigma)^s$.*
- *Nothing else is in $\mathrm{Term}(\Sigma)^s$.*

and the interpretation functions are the obvious ones.

The properties of the term algebra are assumed to be known, most importantly:

Theorem A.9 *The term algebra is an initial algebra.*

Theorem A.10 (structural induction principle) *If a subset*[1] B *of the term algebra* $\mathcal{A} = \mathrm{Term}(\Sigma)$ *meets the condition that for each function symbol* $f : s_1 \ldots s_k \to s$*,*

$$(\forall i \leq k : a_i \in B^{s_i}) \Rightarrow f^{\mathcal{A}}(a_1, \ldots, a_k) \in B^s$$

then $B^s = A^s$ *for all* s*.*

Lemma A.11 *If the structural induction principle holds for a* Σ*-algebra* $\mathcal{A}$*, then any homomorphism* $\mathcal{A}' \to \mathcal{A}$ *is surjective.*

Proof. Let B in the structural induction be the image of $\mathcal{A}'$ in $\mathcal{A}$. □

A.1.2 Grammars as signatures

We often notate an algebraic signature as a context-free grammar. Then, each sort of the signature corresponds to a non-terminal of the grammar, and each function symbol corresponds to a production rule. For example, the grammar

$$\begin{array}{rcl}
\langle \textit{stmt} \rangle & ::= & \langle \textit{var} \rangle \texttt{ := } \langle \textit{expr} \rangle \mid \langle \textit{stmt} \rangle ; \langle \textit{stmt} \rangle \\
& \mid & \texttt{if } \langle \textit{expr} \rangle \texttt{ then } \langle \textit{stmt} \rangle \texttt{ else } \langle \textit{stmt} \rangle \\
\langle \textit{expr} \rangle & ::= & \texttt{cons } \langle \textit{expr} \rangle \langle \textit{expr} \rangle \mid \texttt{nil} \\
& \mid & \texttt{car } \langle \textit{expr} \rangle \mid \texttt{cdr } \langle \textit{expr} \rangle \\
& \mid & \langle \textit{var} \rangle \\
\langle \textit{var} \rangle & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots
\end{array}$$

corresponds to the signature

$$\begin{array}{rcl}
(S & = & \{\texttt{stmt}, \texttt{expr}, \texttt{var}\}, \\
F & = & \{(1, \texttt{var expr} \to \texttt{stmt}),\ (2, \texttt{stmt stmt} \to \texttt{stmt}), \\
& & (3, \texttt{expr stmt stmt} \to \texttt{stmt}), \\
& & (4, \texttt{expr expr} \to \texttt{expr}),\ (5, \to \texttt{expr}), \\
& & (6, \texttt{expr} \to \texttt{expr}),\ (7, \texttt{expr} \to \texttt{expr}), \\
& & (8, \texttt{var} \to \texttt{expr}), \\
& & (9, \to \texttt{var})),\ (10, \to \texttt{var})),\ (11, \to \texttt{var}))\ldots\}
\end{array}$$

Observe that the terminal symbols of the grammar do not show up in the signature; in this context they only serve to suggest a convenient alternative notation for applications of function interpretations.

The intuitive idea behind this way of notating signatures is that parse trees for the grammar correspond exactly to elements of the term algebra. If the grammar is unambiguous (which the above example is not), there is a natural correspondence between the term algebra and the set of words in the language defined by the grammar.

[1] formally, a S_Σ-indexed family of subsets of the term algebra's carriers

A.1.3 Termlike algebras

We aim at exhibiting a construction for an algebra of cyclic terms with the given signature Σ. Now we define an important property that we want it to have in common with the term algebra.

Definition A.12 *A Σ-algebra $\mathcal{A}$ is **termlike** if every function interpretation $f^{\mathcal{A}}$ is injective and each element a of any carrier A^s is in the image of the interpretation of exactly one function.*

In other words, each element of a termlike algebra $\mathcal{A}$ can be written as $f^{\mathcal{A}}(a_1, ..., a_k)$ in exactly one way. This means that one can define properties on $\mathcal{A}$ by pattern matching and prove facts about $\mathcal{A}$'s elements by case analysis, just as for the term algebra.

It is immediately clear that the term algebra is termlike.

What one *can't* do with termlike algebras in general is induction. On the contrary, we have

Theorem A.13 *If the structural induction principle holds for a termlike algebra $\mathcal{A}$, then $\mathcal{A}$ is isomorphic to the term algebra.*

Proof. We show that the unique homomorphism $h_{\mathcal{A}}$ from the term algebra to $\mathcal{A}$ (recall that the term algebra is initial) is bijective hence an isomorphism. $h_{\mathcal{A}}$ is surjective because of Lemma A.11 and injective because of the following lemma. □

Lemma A.14 *Let $\mathcal{A}$ be any termlike algebra. The unique homomorphism from the term algebra to $\mathcal{A}$ is injective.*

Proof. Call the term algebra $\mathcal{B}$ and the homomorphism from $\mathcal{B}$ to $\mathcal{A}$ h. We prove

$$\forall s \in S_\Sigma,\ b, b' \in B^s :\ h^s(b) = h^s(b') \Rightarrow b = b'$$

by structural induction on b.

Because the term algebra $\mathcal{B}$ is termlike, there is a $f : s_1 ... s_k \to s$ and some b_is such that $b = f^{\mathcal{B}}(b_1, ..., b_k)$. Then

$$h^s(b) = h^s(f^{\mathcal{B}}(b_1, ..., b_k) = f^{\mathcal{A}}(h^{s_1}(b_1), ..., h^{s_k}(b_k)).$$

Similarly $b' = f'^{\mathcal{B}}(b'_1, ..., b'_{k'})$ and

$$h^s(b) = f'^{\mathcal{A}}(h^{s_1}(b'_1), ..., h^{s_{k'}}(b'_{k'}))$$

Now, because $\mathcal{A}$ is termlike, f must be identical to f' (hence $k = k'$ and $s_i = s'_i$) and $h^{s_i}(b_i) = h^{s_i}(b'_i)$ for $1 \leq i \leq k$.

The induction hypothesis now gives us $b_i = b'_i$, and then

$$b = f^{\mathcal{B}}(b_1, ..., b_k) = f'^{\mathcal{B}}(b'_1, ..., b'_{k'}) = b'$$

as required. □

Corollary A.15 *Every termlike algebra has an initial algebra as subalgebra.*

A.1.4 Recursive function definitions over termlike algebras

Let Σ_0 be the signature

$$(\{\mathbb{D}\}, \{(\texttt{nil}, \rightarrow \mathbb{D}),\ (\texttt{cons}, \mathbb{DD} \rightarrow \mathbb{D})\})$$

(which describes atomless Lisp-like data), and consider the following recursive definition of the length of a list:

$$\begin{aligned} \text{length}(\texttt{nil}) &= 0 \\ \text{length}(\texttt{cons}(d_1, d_2)) &= 1 + \text{length}(d_2) \end{aligned}$$

This defines a nice total function $\text{Term}(\Sigma_0) \rightarrow \mathbb{N}$. Which hopes can we have of making this kind of definition work for other termlike algebras than $\text{Term}(\Sigma)$? To answer this, we must first state what we mean by "this kind of definition". The general form we're concerned about is

$$\begin{aligned} \psi(f_1(a_1, ..., a_{k_1})) &= g_1(\psi(a_1), ..., \psi(a_{k_1})) \\ &\vdots \\ \psi(f_n(a_1, ..., a_{k_n})) &= g_n(\psi(a_1), ..., \psi(a_{k_n})) \end{aligned}$$

where $f_1, ..., f_n$ are the function symbols of Σ and g_i are functions that are specified as part of the recursive definition. In the concrete example above, we have $g_0() = 0$ and $g_1(x, y) = y + 1$.

We can make ψ's codomain C into a Σ-algebra $\mathcal{C}$ by setting ${f_n}^{\mathcal{C}} = g_n$. Clearly, the g_n functions can be immediately recovered from the algebra: Σ-algebra and recursive definition are equivalent concepts.

When we view the recursive definition this way, it simply states that ψ must be a homomorphism into $\mathcal{C}$. Now we can easily argue that a recursive definition always defines a unique, total function $\psi : \text{Term}(\Sigma) \rightarrow \mathcal{C}$, no matter what kind of internal structure $\mathcal{C}$ has in addition to the g_i functions. Because $\text{Term}(\Sigma)$ is an initial algebra, ψ is simply the unique homomorphism from $\text{Term}(\Sigma)$ to $\mathcal{C}$.

We now want to extend the definition so we can get a homomorphism ψ from an arbitrary termlike algebra $\mathcal{A}$ to $\mathcal{C}$. We rephrase the problem in fixed-point terms: ψ is a homomorphism iff it is a fixed point of the functional Ψ:

$$\Psi(\psi) = \lambda a.\textbf{case } a \textbf{ of } \left[f_i(a_1, ..., a_k) \mapsto {f_i}^{\mathcal{C}}(\psi(a_1), ..., \psi(a_k))\right]_i$$

(the case analysis here is acceptable exactly because $\mathcal{A}$ is termlike). Ψ does not in general have a fixed point, but there are some common cases where we can find one:

- If we can order $\mathcal{C}$ as a *Scott domain* so that the g_i functions are continuous, then these properties carry over to the function space $\mathcal{A} \rightarrow \mathcal{C}$ and the Ψ functional, respectively. We can then use the fixed point theorem

to find a minimal fixed point for Ψ which is a natural candidate to use for ψ.

Note that the "trivial" application of this idea does not work well. Formally we *might* take any odd algebra $\mathcal{C}$, add an artificial bottom element to make it a flat domain $\mathcal{C}_\perp$, and extend the $f^{\mathcal{C}}$ functions to $\mathcal{C}_\perp$ in the canonical strict way. However, ψ will end up being non-$\perp$ exactly on the elements of the initial algebra (which is embedded in $\mathcal{A}$ by Corollary A.15), so it does not gain anything we did not have in advance.

A more interesting example is the list-length definition above. We can take $C = \mathbb{N} \cup \{\infty\}$ which with the usual ordering is a Scott domain. If we define $\infty + 1 = \infty$, the function $g_2(x, y) = y + 1$ becomes (Scott) continuous. The least fixed point we get then assigns the correct length to finite lists—even if the list elements are infinite—and makes the length of infinite lists be ∞.

- If we can order $\mathcal{C}$ as a *complete lattice* so that the g_i functions are order-preserving, then these properties carry over to the function space $\mathcal{A} \to \mathcal{C}$ and the Ψ functional, respectively. We can then use Tarski's fixed point theorem[2] to find either a minimal or a maximal fixed point. Those two are not in general identical, so it is important that the definition states which of them is meant.

A.1.5 Construction of the graph algebra

Let a signature Σ be given. We now construct the **graph algebra** for Σ, which directly models the use of circular data structures to represent certain infinite terms.

Definition A.16 *A* Σ***-graph*** *is a triple* $\gamma = (V, \varphi, \eta)$ *where*

- V *is a finite set of* ***vertices****. Formally we may require that* $V \subseteq \mathbb{N}$ *so that the class of all* Σ*-graphs is a set.*
- $\varphi : V \to F_\Sigma$ *is any map from vertices to function symbols.*
- $\eta : V \to \mathbb{N} \xrightarrow{\text{fin}} V$ *is the* ***edge map***

such that for each $v \in V$ *with* $\varphi(v) = f : s_1 \ldots s_k \to s$, $\eta(v)(i)$ *is defined precisely for* $1 \leq i \leq k$ *and* $\mathsf{Sr}\, \varphi(\eta(v)(i)) = s_i$.

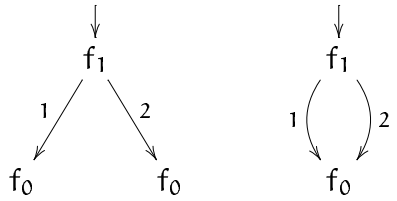
Intuitively, the nodes in a Σ-graph model records in a store and the edges model pointers stored in the records.

Definition A.17 *A* ***pointed*** Σ***-graph*** *is a pair* $\pi = (\gamma, v_0)$ *such that* $v_0 \in V_\gamma$. *We call the set of all pointed* Σ*-graphs* Π.

A pointed Σ-graph models a pointer into some given store.

[2] Theorem 4.11 of Davey and Priestley [1990]

The elements of the graph algebra will be pointed Σ-graphs. Or rather, they will be *equivalence classes* of pointed Σ-graphs, as we want graphs such as



to represent "the same" term. Therefore, we need to define a suitable equivalence relation on Π. That is done by the following series of definitions.

Definition A.18 *For any $\pi = (\gamma, v_0) \in \Pi$, define*

$$\begin{aligned} \mathrm{case}(\pi) &= \varphi_\gamma(v_0) \\ \mathrm{sub}_i(\pi) &= (\gamma, \eta_\gamma(v_0)(i)) \end{aligned}$$

Intuitively, $\mathrm{case}(\pi)$ tells us the topmost function symbol in π, and $\mathrm{sub}_i(\pi)$ extracts the pointers that represent the subterms.

Definition A.19 *A relation $R \subseteq \Pi \times \Pi$ is called **valid**[3] if for any $(\pi_1, \pi_2) \in R$ it holds that*

- $\mathrm{case}(\pi_1) = \mathrm{case}(\pi_2)$
- $(\mathrm{sub}_i(\pi_1), \mathrm{sub}_i(\pi_2)) \in R$ *for all relevant* i.

Lemma A.20 *Any (finite or infinite) union of valid relations is valid.*

Proof. Let $\Theta : \mathcal{P}(\Pi \times \Pi) \to \mathcal{P}(\Pi \times \Pi)$ be the map defined by

$$\Theta(R) = \{(\pi, \pi') \mid \mathrm{case}(\pi) = \mathrm{case}(\pi'), \forall i : (\mathrm{sub}_i(\pi), \mathrm{sub}_i(\pi')) \in R\}$$

It is easy to see that a relation R is valid precisely if $R \subseteq \Theta(R)$.

Now, given any family $(R_i)_i$ of relations with $R_i \subseteq \Theta(R_i)$, we show that $\bigcup_i R_i \subseteq \Theta(\bigcup_i R_i)$. Let (π, π') be an element of $\bigcup_i R_i$; then there is an i such that

$$(\pi, \pi') \in R_i \subseteq \Theta(R_i) \subseteq \Theta(\bigcup_i R_i)$$

where the last inclusion follows because Θ clearly preserves set inclusion. $\square$

Theorem A.21 (definition of $\cong$) *There is a unique largest valid relation $\cong$; it is the union of all valid relations in Π.*

[3] The word is from Paterson and Wegman [1978] which uses "valid" for a similar but not completely equivalent class of relations.

Proof. The union of all valid relations is itself valid by Lemma A.20; it is obviously the largest valid relation.[4] □

The intuition behind this definition of $\cong$ is that $\cong$ only considers two pointed Σ-graphs different if it would be invalid to identify them. Formally, this style of definition is known as a **co-inductive** definition.

Now we show a range of nice properties of $\cong$. We shall make extensive use of the **co-induction principle**: To prove that $\pi \cong \pi'$ it suffices to exhibit a valid relation R such that $\pi \; R \; \pi'$. The validity of this is obvious from the definition.

Theorem A.22 *$\cong$ is an equivalence relation.*

Proof. We must show that $\cong$ is

Reflexive: The identity relation $\{(\pi, \pi) \mid \pi \in \Pi\}$ is valid.

Symmetric: The definition of "valid" is symmetric, therefore the opposite of the valid relation $\cong$ is itself valid and thus contained in $\cong$.

Transitive: We show that the the relation $\cong^2$ defined by

$$\pi_1 \cong^2 \pi_3 \iff \exists \pi_2 : \pi_1 \cong \pi_2 \cong \pi_3.$$

is valid. Given any π_1, π_2, π_3 with $\pi_1 \cong \pi_2 \cong \pi_3$ we have $\mathrm{case}(\pi_1) = \mathrm{case}(\pi_2) = \mathrm{case}(\pi_3)$ and for any i: $\mathrm{sub}_i(\pi_1) \cong \mathrm{sub}_i(\pi_2) \cong \mathrm{sub}_i(\pi_3)$ hence $\mathrm{sub}_i(\pi_1) \cong^2 \mathrm{sub}_i(\pi_3)$.

□

Lemma A.23 *If for some $\pi_1, \pi_2 \in \Pi$ it holds that $\mathrm{case}(\pi_1) = \mathrm{case}(\pi_2)$ and $\mathrm{sub}_i(\pi_1) \cong \mathrm{sub}_i(\pi_2)$ for the relevant is, then $\pi_1 \cong \pi_2$.*

Proof. $(\cong) \cup \{(\pi_1, \pi_2)\}$ is a valid relation. □

Lemma A.24 *If vertices are added to a pointed Σ-graph without changing the existing ones, then $\cong$ identifies the original and the result. More precisely: if for $\pi_1 = (\gamma_1, v_0) = ((V_1, \varphi_1, \eta_1), v_0)$ and $\pi_2 = (\gamma_2, v_0) = ((V_2, \varphi_2, \eta_2), v_0)$ it holds that*

- $V_1 \subseteq V_2$
- $\varphi_1 = \varphi_2 \downarrow_{V_1}$
- $\eta_1 = \eta_2 \downarrow_{V_1}$

then $\pi_1 \cong \pi_2$

Proof. $\{((\gamma_1, v), (\gamma_2, v)) \mid v \in V_1\}$ is a valid relation. □

[4] We could also get this result by applying Tarski's fixed point theorem to the Θ in the proof of Lemma A.20, but we'll also need the lemma in subsequent proofs.

Lemma A.25 *If the vertices in a pointed Σ-graph are systematically renamed, then $\cong$ identifies the original and the result. More precisely: Let $\pi = (\gamma, v_0) = ((V, \varphi, \eta), v_0)$ be given, and let θ be any injective function with domain V. Let*

$$\gamma' = (\theta(V), \varphi \circ \theta^{-1}, \lambda v' i.\theta(\eta(\theta^{-1}(v'))(i))).$$

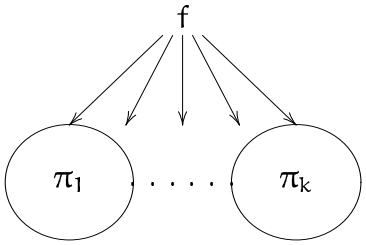
Then γ' is a Σ-graph, and $(\gamma', \theta(v_0)) \cong \pi$.

Proof. The first claim (that γ' is a Σ-graph) is immediate from the definition. To see that $(\gamma', \theta(v_0)) \cong \pi$, observe that $\{((\gamma', \theta(v)), (\gamma, v)) \mid v \in V\}$ is a valid relation. □

Theorem A.26 *For any $f : s_1 ... s_k \to s$ and any $\pi_1, ..., \pi_k$ with $\mathsf{Sr}\,\mathrm{case}(\pi_i) = s_i$ there is a $\pi \in \Pi$ such that*

- $\mathrm{case}(\pi) = f$
- *For $1 \leq i \leq k$:* $\mathrm{sub}_i(\pi) \cong \pi_i$

Proof. The intuitive idea is to take place the graphs π_1 through π_k along each other (after renaming vertices so they are disjoint) and add a single vertex labelled f:



Let $\pi_i = ((V_i, \varphi_i, \eta_i), v_i)$. Because of Lemma A.25, we can assume without loss of generality that the V_is are disjoint. Select a $v_0 \notin \bigcup_i V_i$. Then,

$$\begin{aligned}\gamma = (&\{v_0\} \cup \textstyle\bigcup_i V_i &,\\ &\{v_0 \mapsto f : s_1 ... s_k \to s\} \cup \textstyle\bigcup_i \varphi_i &,\\ &\{v_0, i \mapsto v_i \mid 1 \leq i \leq k\} \cup \textstyle\bigcup_i \eta_i)\end{aligned}$$

is well-defined because the functions we take the unions of are disjoint. γ is also clearly a Σ-graph.

Now let $\pi = (\gamma, v_0)$. It is clear that $\mathrm{case}(\pi) = f$, and for each i we have $\mathrm{sub}_i(\pi) = (\gamma, v_i) \cong (\gamma_i, v_i)$ because of Lemma A.24. □

Theorem A.27 *$\pi = (\gamma, v_0) \cong \pi' = (\gamma', v_0')$ if and only if there is a valid relation*

$$R \subseteq \{(\gamma, v) \mid v \in V_\gamma\} \times \{(\gamma', v) \mid v \in V_{\gamma'}\}$$

such that $(\pi, \pi') \in R$.

Proof. "If": trivial. "Only if": Assume $\pi \cong \pi'$ Let $R_0 = \{(\pi, \pi')\}$ and define

$$R_{n+1} = \{(\mathrm{sub}_i(\pi_1), \mathrm{sub}_i(\pi_2)) \mid (\pi_1, \pi_2) \in R_n\}$$

Set $R = \bigcup_n R_n$. We claim that R is valid.

By an easy induction on n we have $R_n \subseteq (\cong)$. Now take any (π_1, π_2) in any R_n. Because $\pi_1 \cong \pi_2$ we immediately have $\mathrm{case}(\pi_1) = \mathrm{case}(\pi_2)$. We also, by definition have $(\mathrm{sub}_i(\pi_1), \mathrm{sub}_i(\pi_2)) \in R_{n+1} \subseteq R$, so R is indeed valid. □

Corollary A.28 *$\cong$ is decidable.*

This follows directly from Theorem A.27, since there are only finitely many candidates for R and it is easy to check whether a candidate is indeed valid. The proof of the theorem also suggests an $O(n^2)$ algorithm: construct R by the indicated procedure; the relation holds iff $\mathrm{case}(\pi_1) = \mathrm{case}(\pi_2)$ for all $(\pi_1, \pi_2) \in R$.

Definition A.29 $\overline{\pi} = \{\pi' \in \Pi \mid \pi \cong \pi'\}$

Definition A.30 $\overline{\Pi} = \{\overline{\pi} \mid \pi \in \Pi\}$

Definition A.31 *The **graph algebra** $\mathcal{G}(\Sigma)$ is the Σ-algebra defined by*

- $G^s = \{\overline{\pi} \mid \mathrm{Sr}\,\mathrm{case}(\pi) = s\}$
- *For each $f \in F_\Sigma$, $f^{\mathcal{G}}(\overline{\pi}_1, \ldots, \overline{\pi}_k)$ is the $\overline{\pi}$ for which*

$$\mathrm{case}(\pi) = f \ \wedge\ \forall i : \mathrm{sub}_i(\pi) \cong \pi_i$$

Such a π exists by Theorem A.26; it is unique (up to $\cong$) by Lemma A.23. By the definition of $\cong$, if π has the requested property, then so has any $\pi' \in \overline{\pi}$.

Theorem A.32 *The graph algebra $\mathcal{G}$ is termlike.*

Proof. According to Definition A.12 we must prove that

1. *each function interpretation $f^{\mathcal{G}}$ is injective:* Assume that

$$f^{\mathcal{G}}(\overline{\pi_1}, \ldots, \overline{\pi_k}) = f^{\mathcal{G}}(\overline{\pi_1'}, \ldots, \overline{\pi_k'}) = \overline{\pi}.$$

   Then $\pi_i \cong \mathrm{sub}_i(\pi) \cong \pi_i'$, hence $\overline{\pi_i} = \overline{\pi_i'}$.
2. *each $\overline{\pi} \in \mathcal{G}$ is in the image of at most one function interpretation:* Assume $\overline{\pi} = {f_1}^{\mathcal{G}}(\ldots) = {f_2}^{\mathcal{G}}(\ldots)$. Then $\overline{\pi}$ contains a π_1 with $\mathrm{case}(\pi_1) = f_1$ and a π_2 with $\mathrm{case}(\pi_2) = f_2$. Because $\pi_1 \cong \pi_2$, it must be that $f_1 = f_2$.
3. *each $\overline{\pi} \in \mathcal{G}$ is in the image of at least one function interpretation:* Let $\mathrm{case}(\pi) = f$. Then $\overline{\pi} = f^{\mathcal{G}}(\overline{\mathrm{sub}_1(\pi)}, \ldots, \overline{\mathrm{sub}_k(\pi)})$

□

Theorem A.33 *The initial subalgebra of the graph algebra (cf. Corollary A.15) consists of those elements that have a representative* (γ, v_0) *where* γ *is acyclic.*

Proof. From the construction in the proof of Theorem A.26 it is clear that the set of elements with an acyclic representative are indeed a subalgebra. To show that this subalgebra is initial, we apply Theorem A.13.

The structural induction principle is valid for the set of acyclic γs because it can be reduced to "long" mathematical induction on the length of the longest possible path from the root of each graph. (This does not work for arbitrary γs, because when the graph is cyclic there may be arbitrarily long paths from the root node). □

A.2 Proof of Theorem 3.3

We're now in a position to prove Theorem 3.3 (on page 35). To do this we'll have to exhibit a notion of variables and terms, as well as a concrete $\sigma[\![\,]\!]$ function.

We choose to use finite subsets of $\langle Addr \rangle$ as variable names:

$$\mathbb{V} = \left\{ X \subseteq \langle Addr \rangle \;\middle|\; \#X < \infty \right\}$$

This choice may seem arbitrary, but it happens to simplify some of the proofs about unification.

Intuitively, a term is something made up from the grammar

$$\begin{array}{rcl} t & ::= & f(t_1, \ldots, t_{|f|}) \\ & | & X \\ & | & \bot \end{array}$$

where X is a variable name and $\bot$ is the "meaning" of a dangling pointer. To make this intuition formal and applicable to cyclic data structures we view the grammar as specifying a signature Σ_P:

- There is a single sort $\langle t \rangle$.
- For each functor name f there is a function symbol

$$f : \underbrace{\langle t \rangle \ldots \langle t \rangle}_{|f|} \to \langle t \rangle$$

- For each variable name $X \in \mathbb{V}$ there is a function symbol $X : \to \langle t \rangle$.
- There is one additional function symbol $\bot : \to \langle t \rangle$.

Then, a "term" is an element of the graph algebra $\mathcal{G}(\Sigma_P)$.

Now, a store is *almost* a Σ_P-graph – we just have to replace every uninst with an appropriately chosen variable name and prune away the instantiated variables:

Definition A.34 *Let σ be a store. $\mathsf{G}(\sigma)$ is the Σ_{P}-graph (V, φ, η) where*

- $V = \{\alpha \in \mathrm{Dom}\,\sigma \mid \sigma(\alpha) \notin \langle \mathit{Addr} \rangle\} \uplus \{\bot\}$
- $\varphi(\bot) = \bot$
- $\varphi(\alpha) = \textbf{case}\ \sigma(\alpha)\ \textbf{of} \begin{cases} \mathsf{f}(\alpha_1, \ldots, \alpha_i) & \mapsto & \mathsf{f} \\ \mathsf{uninst} & \mapsto & \{\alpha' \in \mathrm{Dom}\,\sigma \mid \mathsf{F}_\sigma(\alpha') = \alpha\} \end{cases}$
- $\eta(\alpha)(i) = \mathsf{F}_\sigma(\alpha_i)$ *if* $\sigma(\alpha) = \mathsf{f}(\alpha_1, \ldots, \alpha_{|\mathsf{f}|})$

Now we can define the meaning of an address α in the store σ:

Definition A.35 *Assume that σ has no local cycles. Then*

$$\sigma[\![\alpha]\!] = \overline{(\mathsf{G}(\sigma), \mathsf{F}_\sigma(\alpha))}.$$

Theorem 3.3 now follows directly from the various definitions.

Observe that if σ has no cycles, local or otherwise, then Theorem A.33 tells us that $\sigma[\![\alpha]\!]$ is a conventional finite term for each α.

We also note

Lemma A.36 *If $\sigma[\![\alpha]\!] \in \mathbb{V}$, then*

$$\sigma[\![\alpha]\!] = \left\{\alpha' \;\middle|\; \sigma[\![\alpha']\!] = \sigma[\![\alpha]\!]\right\}$$

which is an easy consequence of Definition A.34.

A.3 Proof of Theorem 3.11

We now prove the two parts of Theorem 3.11, which relates the unification algorithm defined in Definitions 3.9ff (on page 42) to the "most general unifier" concept defined in Section 3.6.1 (on page 41).

Throughout the subsection we let σ_0, α_0 and α_{00} be given and study the process of unifying α_0 and α_{00} in σ_0.

Lemma A.37 *For any $(\sigma, \mathcal{C})$ such that $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rightarrowtriangle^* (\sigma, \mathcal{C})$ it holds that $\sigma \blacktriangleright \sigma_0$ and $\mathrm{Dom}\,\sigma = \mathrm{Dom}\,\sigma_0$.*

Proof. By induction of the number of $\rightarrowtriangle$ steps in $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rightarrowtriangle^* (\sigma, \mathcal{C})$.

The base case is trivial ($\sigma_0 \blacktriangleright \sigma_0$).

The induction steps corresponding to $\rightarrowtriangle$ rules (a), (b), and (c) are also trivial: these rules do not change σ.

For $\rightarrowtriangle$ rule (d) we see immediately that $\sigma\{\alpha_1 \mapsto \alpha_2\} \blacktriangleright \sigma$. The conclusion then follows from the fact that $\blacktriangleright$ is transitive. □

Lemma A.38 *For any $(\sigma, \mathcal{C})$ such that $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rightarrowtriangle^* (\sigma, \mathcal{C})$ it holds that $(\alpha_0, \alpha_{00}) \in \mathcal{C}$.*

Proof. Trivial. □

Lemma A.39 *For any* $(\sigma, \mathcal{C})$ *such that* $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rightarrowtriangle^* (\sigma, \mathcal{C})$*, any unifier* σ_ω *of* α_0 *and* α_{00} *in* σ_0*, and any* $\alpha_1, \alpha_2 \in \langle Addr \rangle$*, it holds that*

$$\left[\sigma(\alpha_1) = \alpha 2 \ \vee \ (\alpha_1, \alpha_2) \in \mathcal{C}\right] \implies \sigma_\omega[\![\alpha_1]\!] = \sigma_\omega[\![\alpha_2]\!].$$

Proof. By induction of the number of $\rightarrowtriangle$ steps in $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rightarrowtriangle^* (\sigma, \mathcal{C})$.

The base case is trivial. There is an induction step for each of the $\rightarrowtriangle$ rules.

a. Trivial.

b. Because $\sigma_\omega \blacktriangleright \sigma_0$ and $\sigma \blacktriangleright \sigma_0$ with $\mathrm{Dom}\,\sigma = \mathrm{Dom}\,\sigma_0$, we know that

$$\sigma_\omega(\alpha_j) = \sigma_0(\alpha_j) = \sigma(\alpha_j) = f(\alpha_{j1}, \ldots, \alpha_{j|f|})$$

for $j = 1, 2$. This means that, by Theorem 3.3,

$$f(\sigma_\omega[\![\alpha_{11}]\!], \ldots, \sigma_\omega[\![\alpha_{1|f|}]\!]) = \sigma_\omega[\![\alpha_1]\!] = \sigma_\omega[\![\alpha_2]\!] = f(\sigma_\omega[\![\alpha_{21}]\!], \ldots, \sigma_\omega[\![\alpha_{2|f|}]\!])$$

hence (because $\mathcal{G}(\Sigma_P)$ is termlike) $\sigma_\omega[\![\alpha_{1i}]\!] = \sigma_\omega[\![\alpha_{2i}]\!]$.

c. By applying the induction hypothesis twice,

$$\sigma_\omega[\![\sigma(\alpha_1)]\!] = \sigma_\omega[\![\alpha_1]\!] = \sigma_\omega[\![\alpha_2]\!]$$

d. Trivial: (α_1, α_2) is already in $\mathcal{C}$.

□

Theorem A.40 (soundness) *If our unification algorithm produces a unifier* σ_ω *of* α_0 *and* α_{00}*, then* σ_ω *is a most general unifier.*

This is a special case of the following induction lemma:

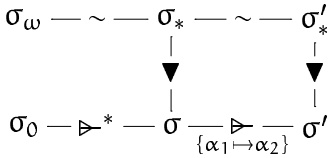
Lemma A.41 *For any* $(\sigma, \mathcal{C})$ *such that* $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rightarrowtriangle^* (\sigma, \mathcal{C})$ *and any unifier* σ_ω *of* α_0 *and* α_{00} *in* σ_0 *it holds that* σ_ω *is less general than* σ*, that is, there is a* σ_* *such that* $\sigma_\omega \sim \sigma_*$ *and* $\sigma_* \blacktriangleright \sigma$*.*

Proof. By induction of the number of $\rightarrowtriangle$ steps in $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rightarrowtriangle^* (\sigma, \mathcal{C})$.

The base case is immediate: $\sigma = \sigma_0$, $\mathcal{C} = \{(\alpha_1, \alpha_2)\}$, and the required properties follow directly from the definition of the involved concepts.

The induction cases corresponding to $\rightarrowtriangle$ rules (a) through (c) are also trivial: these rules do not change the store. Thus, assume that $(\sigma, \mathcal{C}) \rightarrowtriangle (\sigma', \mathcal{C})$ by rule (d), and that the statement of the lemma holds for $(\sigma, \mathcal{C})$.

We need to find σ'_* such that $\sigma_\omega \sim \sigma'_*$ and $\sigma'_* \blacktriangleright \sigma'$. The induction hypothesis gives us a σ_* such that $\sigma_\omega \sim \sigma_*$ and $\sigma_* \blacktriangleright \sigma$. Because $\sim$ is an equivalence relation we can forget about σ_ω and simply seek a $\sigma'_* \blacktriangleright \sigma'$ such that $\sigma'_* \sim \sigma_*$.



We know that there is $(\alpha_1, \alpha_2) \in \mathcal{C}$ such that

d1. $\sigma(\alpha_1) = \mathsf{uninst}$

d2. $\sigma(\alpha_2) \notin \langle \mathit{Addr} \rangle$

d3. $\alpha_1 \neq \alpha_2$

d4. $\sigma' = \sigma\{\alpha_1 \mapsto \alpha_2\}$

5. (from Lemma A.39) $\sigma_*[\![\alpha_1]\!] = \sigma_*[\![\alpha_2]\!]$

Now one of the following cases applies:

- $\sigma_*(\alpha_1) = \mathsf{uninst}$.
  Set $\sigma'_* = \sigma_*\{\alpha_1 \mapsto \alpha_2, \alpha_2 \mapsto \mathsf{uninst}\}$.
  Because $\sigma_*[\![\alpha_1]\!] = \sigma_*[\![\alpha_2]\!]$ it must hold that $\mathsf{F}_{\sigma_*}(\alpha_2) = \alpha_1$, hence $\sigma_*(\alpha_2) \in \langle \mathit{Addr} \rangle$. Therefore, $\sigma(\alpha_2)$ must be uninst, so $\sigma'(\alpha_2) = \mathsf{uninst}$. Thus $\sigma'_* \blacktriangleright \sigma'$ as required.
  Additionally, if we define $\theta(\alpha) = (\textbf{if } \alpha = \alpha_1 \textbf{ then } \alpha_2 \textbf{ else } \alpha)$, then $\mathsf{F}_{\sigma'_*}(\alpha) = \theta(\mathsf{F}_{\sigma_*}(\alpha))$, so we can see that $\sigma'_* \sim \sigma_*$ by invoking Lemma A.25 with this θ.
- $\sigma_*(\alpha_1) \neq \mathsf{uninst}$ and $\mathsf{F}_{\sigma_*}(\alpha_1) = \mathsf{F}_{\sigma_*}(\alpha_2)$.
  If $\sigma_*\{\alpha_1 \mapsto \alpha_2\}$ has no local cycles, then use that as σ'_*. Otherwise, α_2 must be part of the local cycle, hence $\sigma_*(\alpha_2) \in \langle \mathit{Addr} \rangle$ hence $\sigma(\alpha_2) = \sigma'(\alpha_2) = \mathsf{uninst}$ and we can use $\sigma_*\{\alpha_1 \mapsto \alpha_2, \alpha_2 \mapsto \mathsf{F}_{\sigma_*}(\alpha_2)\}$ as σ'_*.
  In either case we get $\mathsf{F}_{\sigma_*} = \mathsf{F}_{\sigma'_*}$ and $\mathsf{G}(\sigma_*) = \mathsf{G}(\sigma'_*)$, so clearly $\sigma_* \sim \sigma'_*$.
- $\sigma_*(\alpha_1) \neq \mathsf{uninst}$ and $\mathsf{F}_{\sigma_*}(\alpha_1) \neq \mathsf{F}_{\sigma_*}(\alpha_2)$.
  Set $\sigma'_* = \sigma_*\{\alpha_1 \mapsto \alpha_2\}$. This does not introduce any local cycles, and clearly $\sigma'_* \blacktriangleright \sigma'$. The trouble is to show that $\sigma'_* \sim \sigma_*$.
  By some unfolding of definitions, it suffices to show that the relation
  $$\mathsf{R} = \left\{ (\mathsf{G}(\sigma_*), \mathsf{F}_{\sigma_*}(\alpha))\ \mathsf{R}\ (\mathsf{G}(\sigma'_*), \mathsf{F}_{\sigma'_*}(\alpha)) \;\middle|\; \alpha \in \langle \mathit{Addr} \rangle \right\}$$
  is a subset of a valid relation. We take
  $$\mathsf{R}' = \left\{ (\mathsf{G}(\sigma_*), \mathsf{F}_{\sigma_*}(\alpha'))\ \mathsf{R}'\ (\mathsf{G}(\sigma'_*), \mathsf{F}_{\sigma'_*}(\alpha'')) \;\middle|\; \sigma_*[\![\alpha']\!] = \sigma_*[\![\alpha'']\!] \right\}$$
  as that valid relation. Clearly $\mathsf{R}' \supseteq \mathsf{R}$, but we must show that R' is valid.
  That is, for any α', α'' with $\sigma_*[\![\alpha']\!] = \sigma_*[\![\alpha'']\!]$, the conditions in Definition A.19 should hold for
  $$(\mathsf{G}(\sigma_*), \mathsf{F}_{\sigma_*}(\alpha'))\ \mathsf{R}'\ (\mathsf{G}(\sigma'_*), \mathsf{F}_{\sigma'_*}(\alpha'')).$$
  If $\mathsf{F}_{\sigma'_*}(\alpha'') = \mathsf{F}_{\sigma_*}(\alpha')$, then that follows easily because $\sigma'_*(\mathsf{F}_{\sigma'_*}(\alpha'')) = \sigma_*(\mathsf{F}_{\sigma_*}(\alpha''))$.
  So, assume that $\mathsf{F}_{\sigma'_*}(\alpha'') \neq \mathsf{F}_{\sigma_*}(\alpha')$. That can only happen if $\mathsf{F}_{\sigma'_*}(\alpha'') = \mathsf{F}_{\sigma'_*}(\alpha_2)$ and $\mathsf{F}_{\sigma_*}(\alpha'') = \mathsf{F}_{\sigma_*}(\alpha_1)$. We then have
  $$\sigma_*[\![\alpha']\!] = \sigma_*[\![\alpha'']\!] = \sigma_*[\![\alpha_1]\!] = \sigma_*[\![\alpha_2]\!]$$

and therefore,

$$(G(\sigma_*), F_{\sigma_*}(\alpha')) \; R' \; (G(\sigma'_*), F_{\sigma'_*}(\alpha_2))$$

is true and has the same validness condition than (α', α''). The validness condition now reduces to the previous case because $F_{\sigma_*}(\alpha_2) = F_{\sigma'_*}(\alpha_2)$.

This completes the proof of Theorem A.40 □

Theorem A.42 (completeness) *If there is any unifier σ_ω of α_0 and α_{00}, then our unification algorithm can find a unifier.*

Proof. Assume σ_ω exists, and select a σ from $\{\sigma \mid \exists \mathcal{C} : \; (\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rhd\!\!-^* (\sigma, \mathcal{C})\}$ that minimises $\#\{\alpha \mid \sigma(\alpha) = \mathsf{uninst}\}$. We claim that the σ thus selected unifies α_0 and α_{00} in σ_0.

Let $\mathcal{C}'$ be maximal among the $\mathcal{C}$s such that $(\sigma_0, \{(\alpha_0, \alpha_{00})\}) \rhd\!\!-^* (\sigma, \mathcal{C})$. This is always possible because the cardinality of any such $\mathcal{C}$ is bounded by $(\#\sigma)^2$.

I claim that

$$R = \left\{ (G(\sigma), F_\sigma(\alpha_1)) \; R \; (G(\sigma), F_\sigma(\alpha_2)) \;\middle|\; (\alpha_1, \alpha_2) \in \mathcal{C}' \right\}$$

is valid, hence σ unifies α_0 and α_{00}.

We show that the validness condition is satisfied for any $(\alpha_1, \alpha_2) \in \mathcal{C}'$. Because $\mathcal{C}'$ is maximal we have $(F_\sigma(\alpha_1), F_\sigma(\alpha_2)) \in \mathcal{C}'$. Therefore, without loss of generality we can assume that $\sigma(\alpha_i) \notin \langle \mathit{Addr} \rangle$ for $i = 1, 2$.

We now divide into two cases:

- At least one of $\sigma(\alpha_1)$ and $\sigma(\alpha_2)$ is uninst. In that case it must be that $\alpha_1 = \alpha_2$, because otherwise $\rhd\!\!-$ rule (d) would allow to construct a store with fewer uninsts than σ, contradicting the choice of σ.

  It is then trivial that the validness condition is satisfied: then $\varphi_{G(\sigma)}(\alpha_1)$ is a nullary function symbol in Σ_P.

- Both of $\sigma(\alpha_1)$ and $\sigma(\alpha_2)$ are number or structure. In that case, $\sigma \blacktriangleright \sigma_0 \blacktriangleleft \sigma_\omega$ means that $\sigma_\omega(\alpha_i) = \sigma_(\alpha_i)$ for $i = 1, 2$.

  Because $\sigma_\omega[\![\alpha_1]\!] = \sigma_\omega[\![\alpha_2]\!]$ by Lemma A.39, $\sigma(\alpha_1)$ and $\sigma(\alpha_2)$ must be the *same* number or structures with the *same* functor. That is the first part of the validness condition.

  The second part of the validness condition now follows from $\rhd\!\!-$ rule (b).

This completes the proof of the Theorem A.42. □

Corollary A.43 (a classic result) *If there is any unifier σ_ω of α_0 and α_{00} in σ_0, then there is a most general unifier σ_{MGU} which can be produced by our unification algorithm.*

Proof. This follows immediately from Theorems A.40 and A.42. □

Corollary A.44 (a less classic result) *Any most general unifier* σ_ω *of* α_0 *and* α_{00} *in* σ_0 *has the property that* $\operatorname{Dom}\sigma_\omega = \operatorname{Dom}\sigma_0$.

That means that no store-based unification algorithm that computes most general unifiers needs to do any (non-temporary) memory allocation.

Proof. Let an arbitrary most general unifier σ_ω be given. By Theorem A.42, our unification algorithm can find a unifier σ'. Because σ_ω is a most general unifier, there must be a σ'_ω such that $\sigma'_\omega \sim \sigma'$ and $\sigma'_\omega \blacktriangleright \sigma_\omega$. We now have

$$\operatorname{Dom}\sigma_0 = \operatorname{Dom}\sigma' = \operatorname{Dom}\sigma'_\omega \supseteq \operatorname{Dom}\sigma_* \supseteq \operatorname{Dom}\sigma_0$$

where the first equality is due to Lemma A.37 and the rest follow from the definitions of $\sim$, $\blacktriangleright$ and unifiers. □

References

Aczel, P. [1977]. An introduction to inductive definitions. In Barwise, J. (ed.), *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter 7, pages 739–782. North-Holland, Amsterdam, The Netherlands, ISBN 0-7204-2285-X.

Aiken, A., Fähndrich, M., and Levien, R. [1995]. Better static memory management: Improving region-based analysis of higher-order languages (extended abstract). In *Programming Language Design and Implementation (ACM SIGPLAN Conference, PLDI '95, La Jolla, CA, USA)*, special issue of *ACM SIGPLAN Notices*, 30(6):174–185.
⟨`http://http.cs.berkeley.edu/~aiken/ftp/region.ps`⟩.

Aït-Kaci, H. [1991]. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, MA, USA, ISBN 0-262-01123-9 (hardcover), 0-262-69146-9 (paperback).
⟨`http://www.isg.sfu.ca/~hak/documents/wam.html`⟩.

Barker, C. [1999]. LPA Win-Prolog goodies. A WWW collection of Prolog source snippets. ⟨`http://perso.wanadoo.fr/colin.barker/lpa/lpa.htm`⟩.

Bevemyr, J. and Lindgren, T. [1994]. A simple and efficient copying garbage collector for prolog. In Hermenegildo, M. and Penjam, J. (eds), *Programming Language Implementation and Logic Programming (6th International Symposium, PLILP '94, Madrid, Spain)*, volume 844 of *Lecture Notes in Computer Science*, pages 88–101. Springer-Verlag, Heidelberg, Germany.

Birkedal, L., Tofte, M., and Vejlstrup, M. [1996]. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages (23rd ACM SIGPLAN-SIGACT Symposium, POPL '96, St. Petersburg Beach, FL, USA)*, pages 171–183. ACM Press, New York, NY, USA, ISBN 0-89791-769-3.
⟨`http://www.diku.dk/users/tofte/publ/popl96.ps.gz`⟩.

Bratko, I. [1990]. *Prolog programming for artificial intelligence*. Addison-Wesley, Reading, MA, USA, 2nd edition, ISBN 0-201-41606-9.

Davey, B. A. and Priestley, H. A. [1990]. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, ISBN 0-521-36766-2.

Debray, S. and Mishra, P. [1988]. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91.

Demoen, B., Engels, G., and Tarau, P. [1996]. Segment preserving copying garbage collection for WAM based Prolog. In *Symposium On Applied Computing (SAC '96, Philadelphia, PA, USA)*, pages 380–386. ACM Press, New York, NY, USA, ISBN 0-89791-820-7. ⟨http://www.cs.unt.edu/~tarau/research/NewBinPrologPapers/CopyingGC.ps⟩.

Elsman, M. [1999]. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, Department of Computer Science, University of Copenhagen. Published as technical report DIKU-TR-99/3. ⟨http://www.itu.dk/research/mlkit/kit_general/phd.ps⟩.

Gabriel, J., Lindholm, T., Lusk, E. L., and Overbeek, R. A. [1984]. A tutorial on the Warren Abstract Machine for computational logic. Technical Report ANL-84-84, Argonne National Laboratory, Argonne, IL.

Henderson, F., Conway, T., and Somogyi, Z. [1995]. Compiling logic programs to C using GNU C as a portable assembler. In Somogyi, Z., Naish, L., Henderson, F., and Conway, T. (eds), *Sequential Implementation Technologies for Logic Programming (ILPS'95 Postconference Workshop, Portland, OR, USA)*, pages 1–15. ⟨http://www.cs.mu.oz.au/mercury/information/papers/mercury_to_c.ps.gz⟩.

Jones, N. D. and Mycroft, A. [1984]. Stepwise development of operational and denotational semantics for Prolog. In *International Symposium on Logic Programming (ISLP '84, Atlantic City, NJ, USA)*, pages 281–288. IEEE Computer Society, Los Alamitos, CA, USA, ISBN 0-8186-0522-7.

Kernighan, B. W. and Ritchie, D. M. [1988]. *The C Programming Language*. Prentice Hall, Englewood Cliff, NJ, USA, second edition, ISBN 0-13-110362-8.

Leroy, X. [1992]. *Polymorphic typing of an algorithmic language*. PhD thesis, University of Paris VII. ⟨ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/phd-thesis.ps.gz⟩.

Makholm, H. [1999]. Specializing C: an introduction to the principles behind C-Mix. Student project 99-1-2, Department of Computer Science, University of Copenhagen. To appear as a DIKU Technical Report in 2000. ⟨http://www.diku.dk/~makholm/cmixintro.ps.gz⟩.

Makholm, H. [2000]. A prototype implementation of RP. An electronic supplement to this thesis. ⟨http://www.diku.dk/students/makholm/rpsys.tar.gz⟩.

Manna, Z. and Waldinger, R. [1981]. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48.

Martelli, A. and Montanari, U. [1982]. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282.

Mellish, C. S. [1981]. The automatic generation of mode declarations for Prolog programs. DAI Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, UK. Also available in Workshop on Logic Programming for Intelligent Systems, 1981.

Mendelson, E. [1997]. *Introduction to Mathematical Logic*. Chapman & Hall, London, UK, fourth edition, ISBN 0-412-80830-7.

Mitchell, J. C. [1996]. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA, USA, ISBN 0-262-13321-0.

Mycroft, A. and O'Keefe, R. A. [1984]. A polymorphic type system for PROLOG. *Articificial Intelligence*, 23(3):295–307.

Paterson, M. S. and Wegman, M. N. [1978]. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167.

Pitts, A. M. [1994]. *Some Notes on Inductive and Co-Inductive Techniques in the Semantics of Functional Programs*, chapter 1. BRICS Note BRICS-NS-94-5. vi+135 pp, draft version.

Prolog Development Center [2000]. Visual Prolog. ⟨`http://vip.pdc.dk/`⟩.

Robinson, J. A. [1971]. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72.

Romanenko, S. and Sestoft, P. [1999]. Moscow ML version 1.44. A lightweight implementation of Standard ML.
⟨`http://www.dina.kvl.dk/~sestoft/mosml.html`⟩.

Somogyi, Z., Henderson, F., and Conway, T. [1996]. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64. ⟨`http://www.cs.mu.oz.au/research/mercury/information/papers/jlp.ps.gz`⟩.

Somogyi, Z. [1987]. A system of precise modes for logic programs. In Lassez, J.-L. (ed.), *4th International Conference on Logic Programming (ICLP '87, Melbourne, Australia)*, MIT Press Series in Logic Programming, pages 769–787. The MIT Press, Cambridge, MA, USA, ISBN 0-262-12125-5.
⟨`http://www.cs.mu.oz.au/~zs/papers/iclp87.ps.gz`⟩.

TIC [1998]. Leroy, X. and Ohori, A. (eds), *Types in Compilation (Second international Workshop, TIC '98, Kyoto, Japan)*, volume 1473 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-64925-5.

Tofte, M. and Birkedal, L. [1998]. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767.
⟨`http://www.itu.dk/research/mlkit/kit_general/toplas98.ps.gz`⟩.

Tofte, M. and Talpin, J.-P. [1993]. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-TR-93/15, Department of Computer Science, University of Copenhagen. The same work is reported in Tofte and Talpin [1994, 1997].
⟨http://www.irisa.fr/prive/talpin/Papers/diku-93-15.ps.gz⟩.

Tofte, M. and Talpin, J.-P. [1994]. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Principles of Programming Languages (21st ACM SIGPLAN-SIGACT Symposium, POPL '94, Portland, OR, USA)*, pages 188–201. ACM Press, New York, NY, USA, ISBN 0-89791-636-0.
⟨ftp://ftp.diku.dk/diku/semantics/papers/D-235.dvi.gz⟩.

Tofte, M. and Talpin, J.-P. [1997]. Region-based memory management. *Information and Computation*, 132(2):109–176.
⟨http://www.itu.dk/research/mlkit/kit2/infocomp97.ps⟩.

Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T. H., Sestoft, P., and Bertelsen, P. [1997]. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen. ⟨http://www.diku.dk/research-groups/topps/activities/kit2/diku97-12.a4.ps.gz⟩.

Velschow, P. and Christensen, M. V. [1998]. Region-based memory management in Java. Master's thesis, Department of Computer Science, University of Copenhagen.

Index