Verification of Pointer Programs Using Regions and Permissions

Romain Bardou

ORSAY
N° d'ordre: 2011PA112220

UNIVERSITÉ DE PARIS-SUD 11
CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

le grade de docteur en sciences DE L'UNIVERSITÉ PARIS XI

PAR

ROMAIN BARDOU

—⋇—

SUJET :

Vérification de programmes avec pointeurs à l'aide de régions et de permissions

Verification of Pointer Programs Using Regions and Permissions

soutenue le 14 octobre 2011 devant la commission d'examen

MM. Peter Müller
François Pottier
Jean Goubault-Larrecq
Burkhart Wolff
Claude Marché

Acknowledgements

This thesis is the result of more than four years of work with my advisor Claude Marché, to who I wish to express my deepest gratitude. Claude was always available for advice and guidance, was always abundantly helpful and supportive, and knocking on his door to abuse his time was a favorite passtime of mine.

I of course wish to thank my reviewers Peter Müller and François Pottier, who dedicated so much of their time to try and understand even the most obscure parts of this thesis. They also offered invaluable advice. My gratitude also goes to Jean Goubault-Larrecq and Burkhart Wolff for accepting to be part of the supervisory committee.

The atmosphere at ProVal is unique. Hearing Jean-Christophe and Sylvain arguing joyfully in the corridor while eating a cake baked by Sylvie or Evelyne is a rather common scene that I'm sure Guillaume, Louis, Kim and Andrei have and will continue to enjoy. I'll also keep a fond memory of all other PhD students or other non-permanent-without-children with who I shared many restaurants: François, Florence, Yannick, Johannes, Stéphane, Wendy, Nicolas, Nicolas and Nicolas (*sic*), Alain, Tuyen, Matthieu, Thierry, Alexandre, Asma, Mohamed... The list goes on.

Last but not least, I would like to thank my father for letting me watch him program in Pascal, on his knees late in the evening, twenty years ago. And I would like to thank my mother for pretending not to know despite the fact that I should have already been in bed.

Contents

Chapter 1

Introduction

Software is invading your everyday life. Your cellphone for instance, or *smart*phone, grants you access to many applications which are not so phone-related. Those applications are themselves managed by a single program: the operating system. Similarly, your car is no longer just a car: you cannot put on your mechanic clothes and open it to understand why this black smoke emerges from the engine. You also need programmer clothes, as your car is driven not only by gears and wheels, but also by complicated pieces of software such as GPS or speed limiters.

Some programs can be considered *critical*. Programs which control your car or your plane might cause injuries or deaths if they ever were not to behave as expected. Other examples include software in health-related systems such as X-ray devices, which should not send more radiation than needed. Critical programs also include programs which entail huge losses of money if they fail. Software which controls rockets is a good example.

All those programs, in particular critical ones, must be trusted to behave correctly if they are to be used. A good question is: what is the condition for a program to be trusted? Current methods to certify critical programs involve a lot of tedious work from programmers. Their code has to be written in very specific ways to ensure it will be easily readable by other programmers who will then be able to say whether the program is safe or not. But humans are very good at failing to detect simple programming errors. And even changing a single character in a ten million lines-of-code software can be disastrous. So these methods are not sufficient. *Testing* programs on some well-chosen inputs is currently the primary certification method in software industry. Testing has become a wide area of research in computer science. Unfortunately, by definition testing cannot prove that a program behaves correctly on all inputs.

While humans may easily fail at detecting programming errors, machines are very good at executing tedious tasks... as long as you tell them what to look for, and how. This observation has led to several fields of research for *static analysis* of programs such as model-checking, abstract interpretation, proof assistants and deductive verification. All of them have in common that the verification is done on the program code *before* it is executed.

1.1 Static Analysis

Model-checking considers a program as involving a finite number of states, even if each individual state represent an infinite number of actual program states. One says which of

these states are unsafe, and then one computes whether these states are reachable. The set of states is computed from the program and unsafe states are given by a logic formula. If these logic formulas can be decided, since there is a finite number of states, these computations terminate and the property: "can my program enter an unsafe state?" is decidable.

Abstract interpretation [Cousot77] deals with infinite numbers of states by abstracting over them. The program is then executed symbolically in the abstract space. We obtain an over-approximation of the reachable states of the program. We can then check whether this over-approximation contains unsafe states. Computing the over-approximation is decidable, but if it does contain unsafe states, it does not mean that the program will reach them. Moreover, we cannot prove *any* property: the abstract domain must obey certain rules.

Proof assistants such as Coq [Coq] provide a very rich logic in which to write programs, their specification and their proofs. The assistant will check proofs for you, but as the logic used is largely undecidable, you will have to do most of the proof by hand.

Deductive verification is based on Hoare logic [Hoare69]. Using this logic, one can specify the behavior of programs using *pre-conditions* and *post-conditions*. The program is valid with respect to this specified behavior if, from any state in which the pre-condition holds, executing the program leads to states in which the post-condition holds as well. By computing the *weakest pre-condition* of a program given its post-condition [Dijkstra76], and then proving that the user-specified pre-condition implies the weakest pre-condition, one proves that the program verifies its specification. Deductive verification tools thus provide: a specification language in which to write pre-conditions and post-conditions, a tool which computes weakest pre-conditions, and tools to prove that this weakest pre-condition is implied by the specified pre-condition. The Why platform [Filliâtre07] and Spec# [Barnett04a] are examples of existing deductive verification tools. Those tools include proof assistants and automatic theorem provers such as Alt-Ergo [AltErgo], Z3 [Z3], CVC3 [Barrett07] or Simplify [Detlefs05].

There is a trade-off between *automation* and *expressiveness*. In model-checking, expressiveness is low as the state space is finite and it must be decidable whether a given state verifies a logic specification. On the other hand and thanks to these limitations, automation is high: once the program and its specification are written, the verification is automatic, although it can take *some* time. Abstract interpretation allows more expressiveness for programs, but the properties that can be checked depend on the abstract state space being used. The level of automation is comparable to the one of model-checking. The language used by proof assistants both to write programs and to specify them is very expressive. As a consequence, proof assistants provide almost no automation.

Deductive verification is an attempt at increasing expressiveness while keeping some automation. Programs may be written in mainstream languages such as C or Java, and we consider the memory to be unbounded so the state space is infinite. Program specifications are written using first-order logic, which is not decidable. The proofs themselves can be done by hand, using proof assistants, or discharged by automatic provers if they are simple enough.

1.2 Deductive verification

This thesis contributes to deductive verification, which we now detail further. We want to prove some properties for every program, such as: no memory error (segmentation fault) will occur, no division by zero will occur, arrays won't be read outside of their bounds. These are generic *safety* properties. Note that some of them, in some languages, are ensured by typing.

For instance, a well-typed Java or OCaml [OCaml] program will not produce a segmentation fault [Damas82, Pottier05], at least when using the default compiler options.

Some properties, however, are program-specific. They say something about the *behavior* of the program. As an example, consider the following C function:

```
int max(int i, int j)
{
  if (i >= j)
    return i;
  else
    return j;
}
```

A behavior of this function is that it always returns an integer greater or equal than its arguments `i` and `j`. Another behavior is that the result is always either `i` or `j`. Those two combined behaviors tell us that this function is the mathematical `max` function.

Now let's use our `max` function in a context where safety is involved:

```
void main()
{
  printf("%d", 10 / max(1, 2));
}
```

To prove the safety of the `main` function, we need to know that `max(1, 2)` won't return `0`. This can be seen as a *pre-condition* of the division operator `a / b` stating that `b` must be different than `0`.

To prove our program in a modular fashion, we want to hide the implementation of `max`. Instead of checking that `max(1, 2)` is not `0` in this particular implementation of `max`, we specify its behavior as a *post-condition*:

```
int max(int i, int j)
  /*@ ensures \result >= i && \result >= j @*/
{
  if (i < j)
    return j;
  else
    return i;
}
```

This example uses a syntax close to ACSL [Baudin09]. The **`ensures`** clause in the comment will be interpreted by the verification tool as a post-condition stating that the result is greater or equal than both `i` and `j`. Note that this post-condition does not describe the full behavior of the `max` function, but it is sufficient to prove that the call to `max(1, 2)` will not return `0`. Also note that this implementation of `max` is different, but both verify this same post-condition.

Verifying the `max` function consists in proving the following *proof obligation*:

$$\forall i, j.\ (i < j \Rightarrow j \geq i \wedge j \geq j) \wedge (\neg(i < j) \Rightarrow i \geq i \wedge i \geq j)$$

Intuitively, this is the post-condition of `max` where `\result` is replaced by the actual returned value, and the conjunction is introduced by the **`if`** statement. This logic formula is

obtained by computing the *weakest pre-condition* [Dijkstra76, Leino05] of the function, based on Hoare logic [Hoare69]. We will detail both Hoare logic and the computation of weakest pre-conditions in Section 2.2. This formula is valid, so the `max` function verifies its specification. This particular proof obligation is simple enough to be discharged automatically by automatic theorem provers.

Verifying the `main` function consists in proving the following proof obligation:

$$\forall r.\ r \geq 1 \wedge r \geq 2 \Rightarrow r \neq 0$$

Intuitively, the integer r represents the value returned by the call to `max(1, 2)`. So r verifies the post-condition of `max` where `\result` is replaced by r, `i` by 1 and `j` by 2. The proof obligation itself comes from the pre-condition of the division operator. It can also be discharged automatically.

1.3 Data Invariants

Data invariants are a handy tool for specifying programs [Barnett04b, Drossopoulou08, Summers09]. Here are some examples of data invariants. A sorted list is always sorted: being sorted is an invariant of the sorted list data structure. In a search tree, the contents of each node is greater than all nodes in the left subtree, and less than all nodes in the right subtree: this is an invariant of the search tree data structure. In a balanced tree, each node also has about as many nodes in the left subtree than in the right subtree: this is again an invariant of balanced trees. Invariants capture design intentions of the programmer. For instance, a type named `pos_int` may be implemented using integers **`int`**, but the programmer may intend all variables of type `pos_int` to be positive.

As invariants are assumed to hold, it is necessary to maintain them. For instance, the function which inserts a new node in a search tree assumes the invariant of the tree and maintains it. Then, the function which looks for a node in a search tree assumes the invariant and uses it to avoid having to look everywhere in the tree. If this function is given a tree which is not a search tree, it may not return the expected result.

Let's see how invariants can be specified in the context of deductive verification. Assume a `set` data structure with a given invariant predicate `inv`. Thus, `inv(s)` states that the invariant of set `s` holds. For instance, if sets are implemented using trees, this invariant can state that the tree is a balanced search tree. We can annotate the `add` function using pre- and post-conditions to specify that it maintains the invariant:

```
void add(set s, int i)
  /*@ requires inv(s);
      ensures inv(s) */
```

The **`requires`** clause states the pre-condition of `add`: the invariant of the set argument `s` must hold when calling `add` on `s`; and the **`ensures`** clause states the post-condition: the invariant of the set argument `s` holds after calling `add` on `s`. In a similar fashion, a function `mem(s, i)` which tests whether integer `i` is in set `s` will require the invariant as a pre-condition.

One could argue that invariants are simply automatic pre- and post-conditions on all arguments, but this methodology raises several issues. The first issue is that invariants must then be proved before each call. One could argue that the proof obligations coming

from the invariant in pre-conditions could be ignored, using the following wrong argument: as invariants are added as post-conditions everywhere, they will hold after each call, and thus will hold everywhere. But the following counter-example shows that this policy is inconsistent:

```
void f(set s)
  /*@ requires inv(s);
      ensures inv(s) */

void g(set s)
  /*@ requires inv(s);
      ensures inv(s) */
{
  ...   /* do something to break the invariant of s */
  f(s); /* we cannot ignore the pre-condition of f here */
}
```

This example makes this issue obvious, but in object-oriented languages the `s` argument would be implicit and thus this problem would appear less clearly. To sum up, if we don't want to have to prove invariants everytime we call a function, we need to use another methodology.

Another issue is that function arguments are not the only available data structures in the function context. For instance, consider the following data structure:

```
typedef int positive;
  /*@ invariant(positive p) = p > 0 */

typedef struct {
  positive first;
  positive second;
} pair;
  /*@ invariant(pair p) = p.first < p.second */
```

Type `positive` is the type of positive integers: it is equal to type **`int`**, but we specify that all `positive` values have an invariant, which is that they are positive. Type `pair` is the type of sorted positive pairs. It is composed of two `positive` values, the `first` and the `second` one. It also has one invariant: the `first` field must be lesser than the `second` field for all `pair` values. Now consider the following function:

```
void h(pair p)
```

What are the pre- and post-conditions that must be added to `h` to take invariants into account? If we strictly follow the idea that invariants are automatic pre- and post-conditions on all arguments, then we obtain the following specification for `h`:

```
void h(pair p)
  /*@ requires p.first < p.second;
      ensures p.first < p.second */
```

But what about the invariants of `p.first` and `p.second`? They are values of type `positive`, so they should be required and ensured to be positive. So we add these requirements in the specification:

```
void h(pair p)
```

```
/*@ requires
      p.first < p.second
      && p.first > 0
      && p.second > 0;
    ensures
      p.first < p.second
      && p.first > 0
      && p.second > 0 */
```

This is fine for finite structures. But what about linked lists or trees? What about containers such as lists of `pair` values? It quickly becomes impossible to decide which values are accessible to `h`.

This brings us to the third issue: modularity. What if we want to hide the `first` and `second` fields of the `pair` data structure to make it an abstract type? It is then even more difficult to decide which pointers are accessible to `h`, as `h` does not even know about `p.first` and `p.second`. Moreover, as the invariant of `pair` values is unfolded in the **`requires`** and **`ensures`** clauses of `h`, the `first` and `second` fields are exposed and modularity is broken.

1.4 Contributions

The main contribution of this thesis is the following:

A type system using regions and permissions to structure the heap in a modular fashion, control pointer aliasing and data invariants and produce proof obligations where pointers are separated.

This thesis introduces a language called *Capucine* which uses this type system (Chapter 4). It also introduces a model and a semantics for this language and proves its soundness. The Capucine language has been implemented in a prototype tool and experimented on some examples (Chapter 3).

The type system is based on the already-existing notion of *regions* to separate pointers statically. It is extended with region parameters and polymorphism for more expressiveness.

The notion of region is extended with *region ownership* to construct a region hierarchy. One region may own, i.e. contain, other regions. This allows modular reasoning.

Permissions, i.e. affine pieces of information about regions, are used to track the state of regions. This also allows easy tracking of the state of data invariants.

Capucine programs are *interpreted as Why programs* to compute proof obligations (Chapter 5). Using information from the type system, it is possible to avoid some of the proof obligations needed to maintain data invariants in existing approaches. Moreover, pointers can be separated in proof obligations using typing information to simplify them.

This thesis also introduces an *inference algorithm* for region annotations and operations (Chapter 6). It is quite helpful in practice.

I claim that this methodology alleviates the need for logic specification annotations about pointer aliasing and data invariants. Indeed, this kind of information can often be handled by the type system instead, and inference helps to ensure that the burden of annotating the program with region operations is light. This also reflects on proof obligations. I also claim that this methodology structures the program in a modular fashion and thus has a chance to scale.

Chapter 2

Technical Background

In this chapter, we give technical background which we need in this thesis. The type system of Capucine borrows parametric polymorphism from ML which we detail first. We then present the foundations of deductive verification. Finally, we detail existing work on data invariants, regions and permissions from which the thesis is largely inspired.

2.1 The ML Type System

The type system of Capucine features many properties inspired by the ML language family, including OCaml [Pottier05, OCaml]. The features which we borrow are: type parameters, polymorphism with implicit prenex quantification and inference. In this section, we introduce those features informally. All examples are written using the OCaml syntax.

Polymorphism Consider the following function which takes a pair as an argument and returns the first component of the pair:

let $f\ (x,\ \_) = x$

One possible type for this function is:

$$int \times int \to int$$

In other word, it is a function which takes a pair of two integers as an argument and returns an integer. Another possible type is:

$$float \times string \to float$$

In fact, function f can accept any type for the first component x of the pair, and any type for the second component. It returns a value which is of the same type than x. A more general type for f is thus:

$$\forall \alpha, \beta.\ \alpha \times \beta \to \alpha$$

This type is *polymorphic*: α and β can be *instanciated* with any type. For instance, we can instanciate α with *float* and β with *string*, and we obtain $float \times string \to float$, which is one of the type we had given to f above.

In ML, and also in Capucine, type variables are always quantified at the head of types. For instance, $int \times (\forall \beta.\ \beta) \to int$ is not a valid ML type. Because of this, we actually omit the quantification on type variables, which are implicitely universally quantified at the head of types. Thus, the type of f is actually written:

$$\alpha \times \beta \to \alpha$$

Polymorphism allows the programmer to avoid duplicating code. It thus allows to avoid duplicating errors as well, and is a key feature for modularity [Hughes89].

Type Parameters Consider the following linked list data structure.

type *list* =
| *Nil*
| *Cons* **of** *int* × *list*

This defines the type of *integer lists*, with two constructors. The first constructor is *Nil*, which is the value denoting the empty list. The second constructor is *Cons*, with two arguments. The first argument has type *int*: it is the value stored in the first node of the list. The second argument has type *list*: it is the remaining of the list, also called its *tail*. For instance, *Cons*(42, *Nil*) is the list with one element: the integer 42. The remaining of the list is empty.

The first argument of constructor *Cons* can be replaced by any type: *float*, *string*... This does not change the structure of the list itself, only the type of its contents. Instead of copy-pasting the list for every type we need, we may use a *type parameter*.

type α *list* =
| *Nil*
| *Cons* **of** $\alpha \times \alpha$ *list*

Type α *list* can be instanciated with any type for α. For instance, *int list* is the type of lists of integers, *float list* is the type of lists of floats, and *string list list* is the type of lists of lists of strings.

Type parameters are especially useful when combined with function polymorphism. Indeed, one can write functions which operate on any list. For instance, the *length* function takes a list and computes its length. It does not have to read the contents of the nodes of the list, and thus *length* operates in exactly the same way on lists of integers, lists of strings and so on. The type of function *length* is:

$$\alpha\ list \to int$$

It is polymorphic in the type α of the contents of the list.

Inference Any ML expression e has a *principal type* τ. If e also has type τ', then τ' is an instance of τ. In other words, we can obtain all types of e by instanciating the type variables of τ. For instance, type $\alpha \times \beta \to \alpha$ is the principal type of function f above, and type *float* × *string* → *float* is an instance of this principal type.

Inference algorithms such as the so-called Algorithm W [Damas82] of Luis Damas and Robin Milner, also called the Hindley-Milner algorithm, are capable of automatically computing the principal type of any ML expression. A simple inference algorithm consists in assigning every sub-expression a type variable, and then applying the type system rules to extract *constraints* on type variables. We then apply a *unification algorithm* on those constraints to obtain a substitution σ from type variables to types, which solves the constraint

system. If the type variable given to expression e was α, then the inferred type of e is $\sigma(\alpha)$. The unification algorithm can be written in such a way that $\sigma(\alpha)$ is the principal type of e.

2.2 Hoare Logic

In Section 1.2, we introduced deductive verification using an example. We showed pre- and post-conditions and said that proof obligations could be computed from them, proof obligations being logic formulas which imply that the program verify its specification. This methodology is based on Hoare logic [Floyd67, Hoare69], which we introduce quickly in this section to show how proof obligations are actually computed.

2.2.1 Hoare Triples

A *Hoare triple* is a triple of the form:

$$\{P\}C\{Q\}$$

where C is a *command*, i.e. an operation of the programming language being considered, and P and Q are logic formulas. Formula P is the *pre-condition* and formula Q is the *post-condition*. These logic formulas depend on the *program state*. Note that Hoare logic can be applied to many programming languages, as well as many logic languages, including but not limited to first-order logic. This also means that the program state can take different forms. Here, we consider it to be a simple map from variable names to their values. A Hoare triple is *valid* if, for any program state S such that P holds in S, if C executed on S leads to the new program state S', then Q holds in S'. Hoare logic defines rules to build Hoare triples. Hoare logic is shown sound: triples built from these rules are always valid.

Skip The first Hoare logic rule is trivial:

$$\{P\}\textsf{skip}\{P\}$$

where P is any predicate and **skip** is the command which does nothing. Indeed, the program state is not modified by **skip**, so if P holds before **skip** is executed, it still holds after.

Sequence A rule allows to compose Hoare triples:

$$\frac{\{P\}C_1\{Q\} \qquad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

where $C_1;C_2$ is the sequence of C_1 followed by C_2. Indeed, if C_1 transforms a state where P holds into a state where Q holds, and if C_2 transforms a state where this same Q holds into a state where R holds, then C_1 followed by C_2 transforms a state where P holds into a state where R holds.

Consequence The consequence rule allows to strengthen the pre-condition or weaken the post-condition:

$$\frac{P' \Rightarrow P \qquad \{P\}C\{Q\} \qquad Q \Rightarrow Q'}{\{P'\}C\{Q'\}}$$

Assignment A most fundamental rule is the rule for variable assignment:

$$\{Q[x \mapsto v]\}x := v\{Q\}$$

where $Q[x \mapsto v]$ denotes predicate Q where all free occurences of x are replaced by v. Here is an example of this rule being applied:

$$\{x + y = 3 \wedge y = 2\}x := x + y\{x = 3 \wedge y = 2\}$$

which, using the consequence rule, is equivalent to:

$$\{x = 1 \wedge y = 2\}x := x + y\{x = 3 \wedge y = 2\}$$

It is important to note that we need to know whether x and y are *separated*, i.e. whether they are the same variable. In this example we assume x and y are different variables.

Test Now let's show the rule for the **if** statements:

$$\frac{\{e \wedge P\}C_1\{Q\} \qquad \{\neg e \wedge P\}C_2\{Q\}}{\{P\}\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2\{Q\}}$$

Note how the **if** condition e is added to the pre-condition of C_1 and C_2. Here is an instance inspired by function `max` of Section 1.2, where r is the returned value:

$$\frac{\{i \geq j\}r := i\{r \geq i \wedge r \geq j\} \qquad \{i < j\}r := j\{r \geq i \wedge r \geq j\}}{\{\top\}\textbf{if } i \geq j \textbf{ then } r := i \textbf{ else } r := j\{r \geq i \wedge r \geq j\}}$$

Note that we also applied the consequence rule to replace $i \geq i \wedge i \geq j$ into $i \geq j$ and to weaken $j \geq i \wedge j \geq j$ into $i < j$ in the post-condition of each premise, respectively.

Loops The rule for *while* loops is the following:

$$\frac{\{P \wedge e\}C\{P\}}{\{P\}\textbf{while } e \textbf{ do } C\{P \wedge \neg e\}}$$

Predicate P is called the *loop invariant*: it is preserved by the body C of the loop. Thus, if P holds at the beginning of the loop, then it still holds at the end of the loop. We prove that C preserves P if the condition e holds, as the body of the loop is only run if e evaluates to *true*. At the end of the loop, e evalutes to *false*, so the negation of e is available in the post-condition.

Note that the above version does not check that the loop terminates. To check termination, one can tweak the rule to add a *variant* requirement. A variant is an expression which evaluates in a well-founded order and which strictly decreases with each iteration.

Function Calls Assume a function f with body C and such that the following Hoare triple is valid:

$$\{P\}C\{Q\}$$

Predicate P is called the *pre-condition*, and predicate Q is called the *post-condition*. They depend on the formal parameters $x_1, \ldots, x_n$ of f.

The rule for function call is the following:

$$\frac{Dom(\sigma) = \{x_1, ..., x_n\} \qquad \text{forall } i,\ \sigma(x_i) = e_i}{\{\sigma(P)\} f(e_1, \ldots, e_n) \{\sigma(Q)\}}$$

We use substitution σ to replace, in the pre-condition and post-condition, the formal parameters of f with the actual arguments.

2.2.2 Weakest Pre-Conditions

Given a command C and a logic formula Q, the *weakest pre-condition* $wp(C, Q)$ is a logic formula such that:

$$\{wp(C, Q)\} C \{Q\}$$

and forall P such that:

$$\{P\} C \{Q\}$$

then $P \Rightarrow wp(C, Q)$. In other words, the weakest pre-condition is the unique pre-condition which is implied by all other pre-conditions.

To prove $\{P\}C\{Q\}$ it is sufficient to prove $P \Rightarrow wp(C, Q)$, as we can then apply the consequence rule:

$$\frac{P \Rightarrow wp(C, Q) \qquad \{wp(C, Q)\} C \{Q\} \qquad Q \Rightarrow Q}{\{P\} C \{Q\}}$$

Thus, one way of verifying a program deductively is to compute its weakest pre-condition. If the user annotated program C with a pre-condition P and a post-condition Q, we generate $P \Rightarrow wp(C, Q)$ as a proof obligation.

Sequence The algorithm to compute weakest pre-conditions can be deduced from Hoare logic rules. For instance, from the composition rule:

$$\frac{\{P\} C_1 \{Q\} \qquad \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}}$$

we deduce the rule to compute the weakest pre-condition of a sequence:

$$wp((C_1; C_2), Q) = wp(C_1, wp(C_2, Q))$$

Assignment From the assignment rule:

$$\{Q[x \mapsto v]\} x := v \{Q\}$$

we deduce the rule to compute the weakest pre-condition of an assignment:

$$wp((x := v), Q) = Q[x \mapsto v]$$

Test The weakest pre-condition of an **if** statement is not straightforward from the Hoare logic rule:

$$wp((\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2), Q) = (e \Rightarrow wp(C_1, Q)) \wedge (\neg e \Rightarrow wp(C_2, Q))$$

It can easily be shown using the consequence rule that this is a valid pre-condition. It is also the weakest. Note that this is not the only method to compute the weakest pre-condition. In particular, *efficient weakest pre-condition* avoids duplicating the post-condition Q [Leino05].

While To the weakest pre-condition of a **while** statement is only define if a *loop invariant* is given. Guessing the loop invariant is undecidable in general, so we usually require the programmer to give one. Here is one way to compute the weakest pre-condition of a loop, without checking termination, with loop invariant I and where $\overline{x}$ is the set of variables being assigned by the body C:

$$wp((\textbf{while } e \textbf{ do } C), Q) = I \wedge \forall \overline{x},\ (e \wedge I \Rightarrow wp(C, I)) \wedge (\neg e \wedge I \Rightarrow Q)$$

Function Calls Assume a function f with body C and such that the following Hoare triple is valid:

$$\{P_f\}C\{Q_f\}$$

Assume the formal parameters of f are $x_1, \ldots, x_n$. Assume n expressions $e_1, \ldots, e_n$. Assume a substitution σ such that for all i, $\sigma(i) = e_i$. Assume the call to $f(e_1, \ldots, e_n)$ assigns variables $\overline{x}$. Here is how we compute the weakest pre-condition of a call to f with arguments $e_1, \ldots, e_n$:

$$wp(f(e_1, \ldots, e_n), Q) = \sigma(P_f) \wedge (\forall \overline{x},\ \sigma(Q_f) \Rightarrow Q)$$

Example Let's apply this algorithm on the `max` function:

$$\begin{array}{ll} & wp((\textbf{if } i \geq j \textbf{ then } r := i \textbf{ else } r := j), r \geq i \wedge r \geq j) \\ = & (i \geq j \Rightarrow wp(r := i, r \geq i \wedge r \geq j)) \wedge (i < j \Rightarrow wp(r := j, r \geq i \wedge r \geq j)) \\ = & (i \geq j \Rightarrow i \geq i \wedge i \geq j) \wedge (i < j \Rightarrow j \geq i \wedge j \geq j) \end{array}$$

2.3 Alias-Free Program Verification

In Section 2.2 was introduced the Hoare logic rule for assignment:

$$\{Q[x \mapsto v]\}x := v\{Q\}$$

As we already noted, this rule is trickier than it seems. Indeed, it relies on *variable substitution* $Q[x \mapsto v]$, which can only be defined when all variables of Q can clearly be said to be equal or different to x. In this section, we present two languages which implement Hoare logic in a context where two variables with different names are always separated. References, pointers, objects are not part of these languages and must be encoded.

2.3.1 The Why Intermediate Language

Why [Filliâtre07] is a tool and an *alias-free* programming language for deductive verification based on weakest pre-condition computation. There is no distinction between statements and expressions, as in the ML language family. Specification language is first-order logic, and the user may define or declare his own logic types, logic functions and predicates to use in the program specification annotations.

To specify his program, the user may annotate it using:

- pre-conditions for functions, which are assumed true when proving the function and must be showned to hold at call site;
- post-conditions for functions, which must be proved when proving the function and can be assumed to hold after a call to the function;
- loop invariants, which are predicates which must be shown to hold at the beginning of the loop and at the end of each iteration, and will be thus also hold at the end of the loop, and which cannot be inferred by the weakest pre-condition algorithm;
- loop variants (optionally), which are terms which must decrease in a well-founded order after each iteration of a loop, and can be used to prove total correctness;
- assertions, which are predicates which must hold at a given program point.

Assertions are a way to set "breakpoints" in the resulting proof obligation. They may help automatic provers and make the proof obligation more readable.

The Why tool can produce proof obligations for several automatic theorem provers, including Alt-Ergo [AltErgo], Z3 [Z3], CVC3 [Barrett07], Simplify [Detlefs05] or Gappa [Gappa]. It can also produce proof obligations for a proof assistant such as Coq [Coq]. The Why tool can also apply transformations to proof obligations. For instance, it can split conjunctions, resulting in several simpler proof obligations. This is very handy to understand which part of the proof obligation is not proved by automatic theorem provers.

Why features mutable variables, which are called *references* in the Why language, but the type system ensures that they may not be aliased. In other words, if two mutable variables x and y are in the context, then because the names "x" and "y" are different we know that x and y are not the same references. Thus, if we modify x, then y is untouched.

To enforce the absence of aliases, the type system imposes some restrictions. In particular:

- when calling a function which takes two references as arguments, the references given at call site must not be aliased;
- a reference cannot contain another reference.

The former ensures that functions can assume their arguments to be separated. The later is quite restrictive as complex data structures such as linked lists cannot be written without some encoding of a memory model.

Translating From Imperative To Functional The Why tool can be seen as a tool which transforms an imperative program into a pure program. The transformation introduces a new immutable variable x_i for each assignment to a variable x. Variable x_i will be used when reading from x until x is assigned again. Let's illustrate the idea using an example:

```
int x = 1;
int y = 2;
x = x + y;
y = 3;
x = x + y;
```

Here is what we obtain:

```
let x1 = 1 in
let y1 = 2 in
let x2 = x1 + y1 in
let y2 = 3 in
let x3 = x2 + y2 in
()
```

This is a purely functional program which simulates the original imperative one. Note that this also looks a lot like the hypotheses that would be available to prove a post-condition on the program, using deductive verification: we can view the successive values of mutable variables as successive immutable variables.

To translate functions with side-effects, we encode them as functions which return not only their return value but also the new values of the variables they modify. For instance, the following imperative function:

```
int f(int* x, int* y)
{
  *x = *x + *y;
  return *x - *y;
}
```

becomes the following pure function:

```
let f (x: int, y: int) =
  let x1 = x + y in
  (x1 - y, x1)
```

The first component of the returned pair is the return value of the function, i.e. the translation of `*x - *y`. The second component is the new value of `x`. Now consider the following caller:

```
int x = 1;
int y = 2;
int z = f(&x, &y);
return x + y + z
```

This call is translated into the following pure program:

```
let x = 1 in
let y = 2 in
let (z, x1) = f(x, y) in
x1 + y + z
```

The new value of `x` is retrieved from the pair returned by the call.

2.3.2 The Boogie Intermediate Language

Boogie [Barnett05] is similar to Why in that it is an intermediate programming language: C# programs are encoded in the Boogie programming language (Boogie PL), and programs in Boogie PL are themselves encoded in the lower-level Boogie language. The Boogie tool then computes proof obligations from these intermediate programs. Proof obligations can be discharged automatically by the Z3 automatic prover. Boogie PL is also alias-free.

However the Boogie language itself, contrary to Why, is not an expression language but a statement language. Its main instructions are **havoc**, **assume** and **assert**. It is remarkable that these three operations are quite expressive. In this section we describe their semantics and show how they can be used to encode basic constructions such as function calls and loops.

The **havoc** `x` operation takes a variable `x` as an argument and assigns it to *some* value. This value is unspecified: this operation is non-deterministic. If we see the execution of the program as a *path*, **havoc** is a crossroads: execution can continue on any path where nothing has changed but, maybe, variable `x`.

The **assert** `P` operation takes a predicate `P` as an argument. Predicate `P` must hold at the current program point: **assert** `P` produces `P` as a proof obligation. If we see the execution of the program as a path, **assert** is a checkpoint. It does not restrict or add new paths to the execution.

The **assume** `P` operation takes a predicate `P` as an argument and inserts it as an hypothesis in the proof obligation. This predicate does not have to be proven. If we see the execution of the program as a path, **assume** restricts possible paths by removing all paths where `P` does not hold.

We can encode assignment `x := e`, if `x` does not appear in `e`, using **havoc** and **assume**:

```
havoc x
assume x = e
```

Indeed, the **havoc** operation allows the program execution to take any path where the value of variable x has changed, but these paths are immediatly restricted by **assume** to those where predicate $x = e$ holds. Thus, the program will take the unique path where value of `x` has changed and is now e.

We can encode the following **while** loop:

```
...
while (x > 0)
  /*@ invariant P */
{
  y = x;
  x = y - 1;
};
...
```

using two Boogie programs. The first program proves the body of the loop:

```
// assume loop invariant
assume P;
// assume we are in the loop
assume x > 0;
// assign y
```

```
havoc y;
assume y = x;
// assign x
havoc x;
assume x = y - 1;
// prove loop invariant is maintained
assert P;
```

The second program abtracts the loop as a black box and can be used to replace the loop in the remaining of the program encoding:

```
...
// loop invariant must hold
assert P;
// the loop modifies some variables x̄
havoc x̄;
// assume we are leaving the loop
assume x <= 0;
// the loop invariant holds
assume P;
...
```

Finally, we show how to encode functions and function calls. Say we have the following function:

```
void incr(int* x)
  //@ requires x >= 0
  //@ ensures x > 0
{
  int y = *x;
  *x = y + 1;
}
```

We can encode the function as:

```
// pre-condition
havoc x;
assume x >= 0;
// body
havoc y;
assume y = x;
havoc x;
assume x = y + 1;
// post-condition
assert x > 0;
```

Notice that the pre-condition is assumed, and the post-condition must be proved. Now say we want to have the following call:

```
z = 42;
incr(&z);
```

This can be encoded as:

```
havoc z;
assume z = 42;
// pre-condition
assert z >= 0;
// function modifies its argument
havoc z;
// post-condition
assume z > 0;
```

Notice that now, the pre-condition must be proved, and the post-condition is assumed. Notice also that it is necessary to know which variables the call may modify. Here, `incr` modifies its argument `z`, hence the **havoc** `z` instruction.

2.4 Memory Models for Programs with Alias

To take into account the fact that pointers may or may not point to the same address, we need to introduce a model of the heap in proof obligations. Several models are possible, and we discuss some of them in this section.

2.4.1 Heap as a Single Array

A simple model is to represent the whole heap as an array H indexed by pointer addresses. To this end, we introduce the *theory of arrays*. We model arrays using two functions *select* and *store*. Given an array a and an index i, $select(a, i)$ is the value of the cell of a at index i. Given an array a, an index i and a value v, $store(a, i, v)$ is a copy of the array a where the contents of the cell at index i is replaced by v. We pose two axioms to reason about *select* and *store*:

$$\forall a, i, v.\ select(store(a, i, v), i) = v$$

and:

$$\forall a, i, j, v.\ i \neq j \Rightarrow select(store(a, i, v), j) = select(a, j)$$

This is enough to describe the whole heap if we represent the address of pointers as indexes i in such an array a.

Consider the following function `f`:

```
void f(int* x, int* y)
  /*@ ensures *x == 1 @*/
{
  *x = 1;
  *y = 2;
}
```

Using the above model, the proof obligation for function `f` is:

$$\forall H, x, y.\ select(store(store(H, x, 1), y, 2), x) = 1$$

Intuitively, H is the heap at the beginning of the function. After the assignment to x, the heap is then $store(H, x, 1)$. After the assignment to y, the heap is then $store(store(H, x, 1), y, 2)$. This proof obligation cannot be discharged. Indeed, we cannot apply any of our two axioms as we don't know whether $x = y$.

Here is one way to fix the specification of `f`:

```
void f(int* x, int* y)
  /*@ ensures x <> y ==> *x == 1 @*/
{
  *x = 1;
  *y = 2;
}
```

The proof obligation becomes:

$$\forall H, x, y.\ x \neq y \Rightarrow select(store(store(H, x, 1), y, 2), x) = 1$$

Because we now have $x \neq y$ as an hypothesis, we can apply our second axiom and simplify the proof obligation as:

$$\forall H, x.\ select(store(H, x, 1), x) = 1$$

which is an instance of our first axiom with $v = 1$.

2.4.2 Need for Separation

Although tools such as Spec#, ESC/Java, VCC or Dafny have proven that the above approach can be used to verify quite large industrial programs, there are important issues regarding the formalization of the memory heap and in particular the way aliasing is handled. We review these issues below, and refer to Sascha Böhme and Michał Moskal [Böhme11] for an experimental comparison of various kinds of models.

Aliasing in Proof Obligations The first issue is that proof obligations sometimes become quite complicated. Indeed, consider what the programmer may have in mind for function `f`:

$$\forall x, y.\ x = 1 \wedge y = 2 \Rightarrow x = 1$$

Now consider the actual proof obligation:

$$\forall H, x, y.\ x \neq y \Rightarrow select(store(store(H, x, 1), y, 2), x) = 1$$

This is less readable and obvious, although it is still simple enough to be discharged by automatic theorem provers.

Now consider a program which assigns a lot of variables:

```
{
  *x0 = v0;
  *x1 = v1;
  ...
  *xn = vn;
}
```

To prove that `*x0` is equal to `v0`, we have to prove that `x0` is different than all variables `x1` to `xn`. This requires not only more work for the proof but also additional specification annotations in the program, to state that all variables are different.

Modularity Another issue is modularity. Say you have designed a library implementing some data structure `set` to represent sets of integers. Maybe you used linked lists, maybe you used trees, but you want to hide this implementation to ensure that you will be able to change it later if needed without adapting the whole program. Say your library provides a function to add a new element into the set:

```
void add(set s, int i)
```

This function assigns some pointers belonging to the internal representation of the set. Now let's use this function `add`:

```
void f(int* x)
  /*@ ensures x = 0 */
{
  *x = 0;
  add(l, 42);
}
```

Because `add` may modify any pointer of the heap H, including x, the proof obligation is not provable:

$$\forall x, H, H', H''.\ H' = store(H, x, 0) \Rightarrow select(H'', x) = 0$$

Here, H is the heap at the beginning of function `f`, H' is the heap after the assignment to `x`, and H'' is the heap after the call to `add`. Because we do not know which pointers `add` may assign, we know nothing about H''. To ensure the proof obligation for `f` is provable, we need to specify which pointers `add` may assign, or, in other words, which pointers it will leave unchanged. This can be specified using an **assigns** clause. For instance, if `add` only modifies a field named `contents` of set `s`:

```
void add(set s, int i)
  /*@ assigns s.contents */
```

This **assigns** clause is comparable to a post-condition stating that the value of all locations except `s.contents` has not changed. However, this **assigns** clause exposes the `contents` field of the set `s`, and thus modularity is broken.

This modularity issue has already been addressed by deductive verification tools. Some of them allow the user to abstract over the set of variables which are modified. We can view the set of locations representing the data structure as a *frame*. From outside of the data structure, it is sufficient to know that only the locations of the frame may be modified. The **assigns** clause is another instance of a frame, used when applying a function. When we apply function `add` above, we know that only the frame specified by the **assigns** clause may be modified by `add`. In logics with a notion of frames such as separation logic [Reynolds02], dynamic frames [Kassios06] or regional logic [Banerjee08], *frame rules* state that if the Hoare triple $\{P\}C\{Q\}$ is valid, if C only modifies frame f and if R mentions no location of f, then the Hoare triple $\{P \wedge R\}C\{Q \wedge R\}$ is valid as well.

2.4.3 Component-as-Array: One Array Per Field

The Burstall-Bornat component-as-array [Bornat00] model is a simple way to achieve some separation when records with multiple fields (C structures or Java objects, for instance) are involved. The idea is that instead of using one single array for the whole heap, we use one

array per field. For instance, say the programmer declared a `pair` record with two fields `first` and `second`:

```
typedef struct {
  int first;
  int second;
} pair;
```

Assume two `pair` pointers `p` and `q`. There is no way for `p->first` to denote the same memory cell as `q->second` unless type casts are allowed. This static information is used to separate the heap into one array per declared field. This way, if `p->first` is modified, the only part of the heap which is changed in the proof obligation is the array corresponding to the `first` field. The array for `second` is *statically untouched*.

Let's see an example of a program using the `pair` structure and compute its proof obligation using the component-as-array model.

```
void incr(pair* p)
  /*@ ensures p->first == \old(p->first) + 1 */
{
  p->first = p->first + 1;
  p->second = p->second + 1;
}
```

In the post-condition, `\old(e)` denotes the value of `e` before the call.

Without the component-as-array approach, the program is transformed into the following:

```
void incr(pairarray* h, pair* p)
  /*@ ensures
        select(*h, p).first == select(\old(*h), p).first + 1 */
{
  *h = store(*h, p,
    setfirst(select(*h, p), select(*h, p).first + 1));
  *h = store(*h, p,
    setsecond(select(*h, p), select(*h, p).second + 1));
}
```

where `pairarray`, `select` and `store` form the array theory with type `pointer` for indexes and type `pair` for values, and where `setfirst` and `setsecond` return a new `pair` with field `first` or `second`, respectively, being changed to the given value. Variable `h` denotes the heap. To prove the post-condition we need to know that `setsecond` does not modify field `first`.

Using the component-as-array approach, we transform this program into the following:

```
void incr(intarray* first, intarray* second, pointer p)
  /*@ ensures select(*first, p) == select(\old(*first), p)+1 */
{
  *first = store(*first, p, select(*first, p) + 1);
  *second = store(*second, p, select(*second, p) + 1);
}
```

where `intarray`, `select` and `store` form the array theory with type `pointer` for indexes. Heap `h` has been split into `first` and `second`. This version is compatible with the Why

no-alias restriction, as to assume `first` and `second` are separated is exactly what we want. Here is the proof obligation which we can compute using Hoare logic rules given in Section 2.2:

$$\begin{array}{ll} & \forall \mathit{first}, \mathit{second}, \mathit{first}', \mathit{second}', p. \\ & \mathit{first}' = \mathit{store}(\mathit{first}, p, \mathit{select}(\mathit{first}, p) + 1) \\ \wedge & \mathit{second}' = \mathit{store}(\mathit{second}, p, \mathit{select}(\mathit{second}, p) + 1) \\ \Rightarrow & \mathit{select}(\mathit{first}', p) = \mathit{select}(\mathit{first}, p) + 1 \end{array}$$

There is no longer a single heap H, as it is separated into two arrays: *first* and *second*. Because *first* and *second* are syntactically two different variables, when *second* is modified *first* is unchanged. Thus we do not have to prove that `p->second` and `p->first` represent different memory cells.

Note that this model does not allow finer pointer arithmetic. It requires typing to ensures record fields are separated from each other, which is not true in C if *casts* are allowed, but is true in higher-level languages such as Java or OCaml. For instance, if `p` is a pointer on a `pair` structure, the expression:

```
(int*) (((int) p) + sizeof(int))
```

may, in a finer memory model for the C language, actually be an integer pointer whose pointed memory cell is physically the same as `p->second`. Such casts are incompatible with the Burstall-Bornat component-as-array model and will not be considered in this thesis.

2.4.4 Regions: One Array Per Region

Another method we can use to split the heap into several arrays is *regions*, and has been used in the Why platform by Thierry Hubert and Claude Marché [Hubert07, Hubert08]. All pointer types are given a unique *region variable*. These variables are then unified using the same unification mechanism used in ML. After this unification is done, if two variables x and y still have different region variables, then we assume that x and y are separated, i.e. are denote different memory cells. We then split the heap using one array per region.

Here is a program which illustrates the region mechanism:

```
void incr3(int* i, int* j, int* k)
  /*@ ensures
        *i = \old(*i) + 1
     && *j = \old(*j) + 1
     && *k = \old(*k) + 1 */
{
  *i = *i + 1;
  if (i != j)
    *j = *j + 1;
  *k = *k + 1;
}
```

The three pointer variables `i`, `j` and `k` are each given a unique region variable, r_i, r_j and r_k. The unification algorithm is then executed, and because of pointer comparison `i != j`, region variables r_i and r_j are unified into a single variable r_{ij}. So after unification there are only two regions: r_{ij} and r_k, and the program is transformed into:

```
void incr3(intarray* rij, intarray* rk, pointer i,
    pointer j, pointer k)
```

```
  /*@ ensures
        select(*rij, i) = select(\old(*rij), i) + 1
     && select(*rij, j) = select(\old(*rij), j) + 1
     && select(*rk, k) = select(\old(*rk), k) + 1 */
{
  *rij = store(*rij, i, select(*rij, i) + 1);
  if (i != j)
    *rij = store(*rij, j, select(*rij, j) + 1);
  *rk = store(*rk, k, select(*rk, k) + 1);
}
```

and here is the proof obligation, slightly re-arranged to avoid the duplication due to the **if** statement:

$$
\begin{array}{ll}
 & \forall r_{ij}, r'_{ij}, r''_{ij}, r_k, r'_k, i, j, k. \\
 & r'_{ij} = \mathit{store}(r_{ij}, i, \mathit{select}(r_{ij}, i) + 1) \\
\wedge & i \neq j \Rightarrow r''_{ij} = \mathit{store}(r'_{ij}, j, \mathit{select}(r_{ij}, j) + 1) \\
\wedge & i = j \Rightarrow r''_{ij} = r'_{ij} \\
\Rightarrow & \mathit{select}(r''_{ij}, i) = \mathit{select}(r_{ij}, i) + 1 \\
\wedge & \mathit{select}(r''_{ij}, j) = \mathit{select}(r_{ij}, j) + 1 \\
\wedge & \mathit{select}(r'_k, k) = \mathit{select}(r_k, k) + 1
\end{array}
$$

To prove the `i` and `j` parts of the post-condition, because they are in the same region, we need to do a case split on whether `i = j`. To prove the `k` part we do not have to prove any pointer difference as `k` is in a fully separated region.

Note that this mechanism is not modular, as regions are unified globally. This is a limitation of the approach. In Capucine, we build upon this mechanism and add *region polymorphism* to ensure we can verify each function without knowing the body of the others.

2.5 Preservation of Data Invariants Using Ownership

In Section 1.3 we introduced the notion of data invariants and showed why they were a challenging problem. The key issue is to control which and when objects verify their invariants. To this end, methodologies have been proposed. Peter Müller introduced the idea of ownership-based invariants in his thesis [Müller02, Müller06]. It was used widely, for instance in VCC [Cohen10] and by Yi Lu and John M. Potter [Lu07]. In this section we present the ownership methodology used in the Spec# platform [Barnett04b, Leino04]. Boogie being an intermediate language of Spec#, this ownership methodology is sometimes refered to as the *Boogie methodology*.

The Spec# ownership methodology can be summarized as follows:

1. objects may *own* other objects;
2. an object can only be owned by one other object (its *owner*);
3. objects may be *open or closed*;
4. a closed object must verify its invariant, and when closing an object, its invariant must be shown valid;
5. a closed object cannot be modified;

6. the invariant of a closed object may only depend on the objects it owns;

7. a closed object may only own closed objects.

In other words, objects are boxes which can contain other boxes. Obviously a given box cannot be inside two boxes A and B at the same time, unless A is itself inside B or vice-versa. And we cannot leave an open box inside a closed box.

Items 1 and 2 create an *ownership tree* between objects. This tree is *dynamic* in that it may change during program execution.

Items 3, 4 and 5 allow to know which object verify their invariants. If an object is closed, it must verify its invariant. If an object is open, its invariant may or may not hold. Open objects are objects being modified: their invariant is temporarily broken and will be restored before closing the object.

Items 6 and 7 control what invariants may depend on. If the invariant of an object A depends on the value of an object B, then modifying B might break the invariant of A. To ensure this does not happen, the ownership methodology requires A to own B. If the invariant of A is supposed to hold (i.e. A is closed), then B is also closed and may not be modified. To modify B, it must be opened first, and to open B we must open A before.

Let's review a slightly modified version of the example of Section 1.3:

```
typedef struct {
  int value;
} positive;
  /*@ invariant(positive p) = p.value > 0 */

typedef struct {
  positive* first;
  positive* second;
} pair;
  /*@ invariant(pair p) =
        p.first->value < p.second->value */
```

Because the invariant of the `pair` data structure depends on its fields `first` and `second`, it must own them. These fields cannot be modified if the `pair` structure is closed. This also helps solving the problem of knowing which invariants hold. Indeed, if a `pair` is closed, then its `first` and `second` fields should be closed too, as they are owned by the `pair`. Thus, if a function takes a closed `pair` as an argument, we do not have to explicitly say that the `first` and `second` fields of the `pair` are closed too. This also applies to recursive data structures such as linked lists or trees.

In its original presentation [Barnett04b], the Spec# methodology is implemented in a purely *dynamic* way: information about the open or closed state of objects is added as an actual field, named `inv`, in every object.

The `inv` field can have three states: `open`, `closed` or `committed`[1]. If `inv` is `open`, then the object is open; if `inv` is `closed`, then the object is closed and is not owned by a closed object; if `inv` is `committed`, then the object is closed, and is owned by a closed object.

[1] The original presentation [Barnett04b] actually uses two boolean fields: `inv` which states whether the object is closed, and `committed` which states whether the object is owned by a closed object. Moreover, `inv` is actually a class name. All invariants of the class of `inv` and its ancestors hold.

This field can then be referred to in the program specification, in particular pre- and post-conditions. For instance, a pre-condition of a function `f` which takes an argument `x` can require `x.inv` to be `closed` or `committed` if the function requires the invariant of `x` to hold. When calling `f`, it is usually easier to show this than to show the invariant of `x` holds. Moreover, the caller does not have to know what the invariant actually is.

A `pack` operation can be used to change the state of an object from `open` to `closed`. The invariant of the object being packed must be proven before. Thus, if `x` owns one field `f`, `pack x` can be encoded as:

```
assert x.inv == open;
assert x.f.inv == closed;
assert invariant(x);
x.inv = closed;
x.f.inv = committed;
```

Note the third **assert** instruction which requires to prove the invariant of `x` actually holds. The opposite operation, `unpack x`, can be encoded as:

```
assert x.inv == closed;
x.inv = open;
x.f.inv = closed;
```

We do not need to prove that `x.f.inv` is `committed` before, as the methodology will ensure this always holds.

2.6 Permissions: Linear Information About Pointers or Regions

Permissions, or *capabilities*, are a way to statically reason about variables. Unlike usual type systems, which ensure that the type of a given expression will never change throughout the reduction of the expression, permissions keep track of changes. At any given point of the program, a set of permissions is available. Operations consume some of them and produce some others. Because *permissions cannot be duplicated*, one cannot both consume it and keep it for the rest of the program.

Garbage Collection A first application of permissions is to keep track of which memory cells are being used and which can be *garbage collected*, i.e. re-assigned for some other purpose. For example, in the following C program:

```
int* x = malloc(sizeof(int));
int* y = malloc(sizeof(int));
*x = 42;
*y = *x + 69;
free(x);
```

the memory cell denoted by pointer `x` is no longer used after the assignment to `*y`. Thus, `x` can be safely freed.

To keep track of which memory cells can be freed, we associate a permission to each variable. Here is the same program annotated with permissions:

```
// No permission is available here
int* x = malloc(sizeof(int));
// Permission x is available
int* y = malloc(sizeof(int));
// Permissions x and y are available
*x = 42;
*y = *x + 69;
free(x);
// Permission y is available
```

To sum up: the `malloc` operation *produces* a permission, the `free` operation *consumes* a permission, and access `*x` *requires* permission x.

Using some inference mechanism, one can insert the `free(x)` operation automatically. Indeed, if a permission is not required after a given program point, then its corresponding variable can be freed, i.e. *garbage collected*.

Permissions and Regions An important issue is *pointer aliasing*. In the above example, what if `malloc` returned the same pointer, i.e. the same memory address, to both `x` and `y` ? Then we would have two permissions on the same memory cell. After the `free(x)` operation, both pointers `x` and `y` would be freed and should no longer be read or written. But permission y would still be available, and thus `y` could still be accessed, which is unsafe. So we require the language to ensure that either aliases never happen, or they are controlled.

One way of controlling aliases is to use *regions*. We already introduced regions in Section 2.4.4 to control aliasing in proof obligations. We use a type system where the type of pointers is annotated by a region. Then, we use one permission per region: when the region is created, its permission is introduced; when a pointer is added to or accessed from the region, its permission is required; when a pointer is freed, all pointers of the region are actually freed at the same time and the permission is consumed.

Translating Imperative Programs to Functional Programs Recent work by Arthur Charguéraud and François Pottier [Charguéraud08] uses permissions on regions to encode imperative programs into a purely functional language. The idea is that because permissions denote information about variables, and because permissions are capable of following changes of such information during the execution of the program, such information can actually represent the *knowledge of the value of the variable*.

The translation is based on the one we presented in Section 2.3.1, and extends it to handle aliasing. This is done by representing regions as functional *maps* from locations to values. Each time a permission is produced, a new map is produced to represent the new value of the region. For instance, if the region of x is r, the assignment:

```
x = 1;
```

consumes the existing permission on r. In other words, the current value of the map representing r is not used anymore. The operation produces the exact same permission on r. This new permission is translated as a new map representing the new value r' of r. So the assignment is translated to the following pure program:

```
let r' = store(r, x, 1) in
...
```

As usual, `store(r, x, 1)` builds a map where each location has the same value as in `r`, except `x` which now has value `1`.

Also noteworthy are the allocation and deallocation operations. Allocation produces a new permission, and thus a new map to represent the region in which the pointer is allocated. Deallocation consumes the permission of a region, thus disabling the corresponding map, and the region and all its locations are no longer accessible.

Chapter 3

Informal Presentation of the Capucine Approach

3.1 Separation Using Regions and Permissions

In Section 2.3, we presented alias-free languages which require that, given two variables x and y, those two variables are *separated* if, and only if the *names* x and y are different. In Section 2.4 we presented how *maps* can encode aliasing in proof obligations, and how pointers can be statically separated into different such maps. In Section 2.4.4 we showed how those two approaches can be combined using *regions*. Say pointer p_1 belongs to region r_1 and pointer p_2 to region r_2. The approach of Thierry Hubert and Claude Marché [Hubert07, Hubert08] uses the *names* of regions r_1 and r_2 to know whether p_1 may be aliased with p_2 or not. If the names r_1 and r_2 are different, then p_1 and p_2 are *statically separated*: we know thanks to typing that they will never denote the same memory cell. Thus we can use two different maps to encode these pointers: one for each region. However, their approach is limited as the region of a pointer cannot change during execution.

Capucine is based on the approach of Section 2.4.4, extended with permissions as introduced in Section 2.6 to allow greater flexibility. In this section, we show the region language of Capucine and how permissions are used to specify separation. We also present region operations which can be used to move pointers from their current region to another, with some restrictions expressed using permissions.

Example of Capucine Program In Capucine, regions are explicit in pointer types. Let's review the example of Section 2.4.4 in the Capucine language. First, let's introduce the *Long class*[1]:

class *Long*
{

[1]We use term *class* to denote records associated with owned regions and invariants. Owned regions and invariants are introduced in Section 3.2 and Section 3.3 respectively. In this thesis, term class is thus not related to object-oriented programming. In particular, inheritance is not considered. However, as this thesis is based on work about object invariants where invariants are attached to classes, we choose to stick with this terminology.

```
  value: int;
}
```

This class is a simple record with one integer field named *value*. This is how integer references are encoded in Capucine. This is similar to OCaml references, as they are actually mutable records with one field.

Now let's see how function *incr3* is written:

```
fun incr3[ρ1: Long, ρ2: Long](i: [ρ1], j: [ρ1], k: [ρ2]): unit
  consumes ρ1 ρ2
  produces ρ1 ρ2
{
  i.value ← i.value + 1;
  if (i ≠ j)
  {
    j.value ← j.value + 1;
  }
  else {};
  k.value ← k.value + 1;
}
```

Function *incr3* takes three pointers as arguments: i, j and k. Pointers i and j have type $[\rho_1]$ while pointer k has type $[\rho_2]$. Type $[\rho]$ reads "at ρ" and is the type of pointers of region ρ. Here, ρ_1 and ρ_2 are parameters of function *incr3*, of class *Long*. This means that i, j and k are objects of class *Long*.[2]

Separation Because i and j belongs to the same region ρ_1, they may be the same pointer. However, k belongs to ρ_2 and will never be equal to i or j.

To ensure ρ_1 and ρ_2 are different regions, function *incr3* consumes two permissions: one on ρ_1 and one on ρ_2. The type system ensures that *permissions cannot be duplicated*. Function *incr3* cannot be called with $\rho_1 = \rho_2$, as then two permissions on the same region ρ_1 would be required.

The Why language introduced in Section 2.3.1 ensures absence of aliases by requiring that a reference cannot appear as two arguments of the same function call. In Capucine we use the same kind of restriction, but on regions instead of pointers.

To sum up, to call *incr3* we not only need three pointer arguments i, j and k. We also need their regions ρ_1 (for i and j) and ρ_2 (for k) and permissions on these two regions. These permissions are consumed by the call, but are also produced: in fact, they are merely required.

Singleton and Group Permissions In the above example, we denoted permission on ρ_1 by ρ_1, and permission on ρ_2 by ρ_2. But in Capucine the language of permission is actually richer and there is no such thing as a permission ρ. Let's introduce two possible permissions for a given region ρ: permissions $\rho^\times$ and ρ^G.

Permission $\rho^\times$ not only denotes that we have a permission on region ρ, but also that ρ is a *singleton region*, while permission ρ^G denotes that ρ is a *group region*. A singleton region contains exactly one pointer. It is thus exactly the same as a reference in a language such

[2] As we use term *class*, it seems natural to use term *object* for values of class types. Again, this is not necessarily related to object-oriented programming.

as Why. A group region may contain any number of pointers, zero and one included. Thus permission $\rho^\times$ is strictly stronger than permission ρ^G. The header of function *incr3* should actually be written:

fun *incr3*[ρ_1: *Long*, ρ_2: *Long*](i: [ρ_1], j: [ρ_1], k: [ρ_2]): *unit*
consumes ${\rho_1}^G$ ${\rho_2}^\times$
produces ${\rho_1}^G$ ${\rho_2}^\times$

Indeed, ρ_1 may contain at least two pointers i and j. If ${\rho_1}^\times$ was required instead of ${\rho_1}^G$, function *incr3* could only be called with $i = j$. Permission ${\rho_2}^G$ could be required instead of ${\rho_2}^\times$, but as *incr3* does not expect several pointers in ρ_2 we might as well require ρ_2 to be singleton.

Empty Permission Permission $\rho^\emptyset$ denotes the fact that region ρ is empty, i.e. it contains exactly zero pointers. As $\rho^\times$, it is strictly stronger than ρ^G. Permission $\rho^\emptyset$ is the permission which is produced when introducing a new region using the following operation:

let region ρ: *Long*

This operation introduces a new region ρ, with permission $\rho^\emptyset$, containing pointers of class *Long*. The scope of ρ is the current statement block. In particular, ρ and its current permission is no longer available outside the function it is declared in.

Because $\rho^\emptyset$ and $\rho^\times$ are stronger than ρ^G, Capucine provides *weakening operations* which weakens permissions $\rho^\emptyset$ and $\rho^\times$ into ρ^G:

weaken empty ρ
weaken single ρ

The first operation consumes $\rho^\emptyset$ and produces ρ^G. The second operation consumes $\rho^\times$ and produces ρ^G. This is sometimes useful, for instance if a function consumes a group region and all you have is a singleton region.

Allocation To create a new object, Capucine provides the following operation:

let x = **new** [ρ]

where ρ is an empty region, i.e. permission $\rho^\emptyset$ is consumed. A pointer is allocated in region ρ which thus becomes singleton: permission $\rho^\times$ is produced. The address of the pointer is put in variable x, which has type [ρ]. There is no *null* pointer as in, for instance, Java or C.

Assignment An assignment such as $x.f \leftarrow e$ can only be done if x is a pointer in a singleton region. The intuition behind this restriction is that if a pointer is potentially aliased, i.e. is in a group region, we don't know exactly which variables of the program are actually modified when the pointer is assigned to another value. Whereas if a pointer is in a singleton region ρ, we know that the only variables of the program which may be modified are exactly all variables of region ρ.

Focus and Unfocus Function *incr3* uses several assignments, some in singleton region ρ_2:

$$k.value \leftarrow k.value + 1$$

and some in group region ρ_1:

$$i.value \leftarrow i.value + 1$$
$$j.value \leftarrow j.value + 1$$

However, we just said in the above paragraph that the only pointers which can be assigned are pointers belonging to singleton regions. To allow assignment of pointers in group regions, we introduce operations allowing to move pointers from their regions to other regions. This justifies the need for permissions instead of simply separating regions using their names.

The **focus** operation transfers a pointer p from a group region ρ to an *empty* region σ, which then becomes singleton. The transfered pointer then belongs to several regions at the same time: its original group region ρ and the target region σ. To ensure that pointers of region ρ are not accessed while pointer p is focused, region ρ is *temporarily disabled*: its permission ρ^G is consumed and permission $\sigma \multimap \rho$ is produced to replace it. Permission $\sigma \multimap \rho$ is inspired by linear implication of linear logic, and its intuitive meaning is: if you give $\sigma^\times$, you can get ρ^G back. To sum up, the following operation:

focus x: ρ **as** σ

assumes x has type $[\rho]$, consumes ρ^G and $\sigma^\emptyset$, and produces $\sigma \multimap \rho$ and $\sigma^\times$. The type of x is changed into $[\sigma]$.

The **unfocus** operation is the opposite of the **focus** operation: it takes a focused pointer and puts it back in its original region. The following operation:

unfocus x: σ **as** ρ

assumes x has type $[\sigma]$, consumes $\sigma^\times$ and $\sigma \multimap \rho$, and produces ρ^G. The type of x is changed into $[\rho]$. No permission on σ is produced: the region is no longer usable. Usually, focus operations are used to assign a pointer which is in a group region; target region σ is thus temporary.

Now that we have introduced singleton and group regions, as well as focus and unfocus operations, we can actually assign pointers in group regions. Here is how we can assign $i.value$ in the *incr3* example:

let region ρ_i: *Long*;	${\rho_i}^\emptyset$, ${\rho_1}^G$, ${\rho_2}^\times$
focus i: ρ_1 **as** ρ_i;	${\rho_i}^\times$, $\rho_i \multimap \rho_1$, ${\rho_2}^\times$
$i.value \leftarrow i.value + 1$;	${\rho_i}^\times$, $\rho_i \multimap \rho_1$, ${\rho_2}^\times$
unfocus i: ρ_i **as** ρ_1;	${\rho_1}^G$, ${\rho_2}^\times$

On the right side are the available permissions at the current program point.

Adoption The *adoption* operation can be used to transfer a pointer from a singleton region to a group region:

adopt x: σ **as** ρ

It is similar to the **unfocus** operation, except that pointer x was not originally in ρ. This operation consumes permissions $\sigma^\times$ and ρ^G, as pointer x of singleton region σ is transfered to group region ρ. Permission ρ^G is produced, but no permission on σ is available after the adoption: the region is no longer usable.

A typical sequence used in Capucine is allocation followed by adoption, to create a new pointer in a group region ρ:

let region σ: *Long*;	$\sigma^{\emptyset}$, ρ^{G}
let x = **new** $[\sigma]$;	$\sigma^{\times}$, ρ^{G}
$x.value \leftarrow 69$;	$\sigma^{\times}$, ρ^{G}
adopt x: σ **in** ρ;	ρ^{G}

After this sequence, region ρ contains a new *Long* pointer of value 69.

3.2 Modularity Using Region Ownership

Capucine features *region ownership*. Region ownership is similar to object ownership which can be used to allow object invariants to mention other objects, as was discussed in Section 2.5. However, while with object ownership, objects own other objects, with region ownership objects own regions. If an object owns a group region, it owns all the objects of the region. Capucine also uses ownership to extend the expressiveness of invariants, and we will discuss this in Section 3.3. In this section, we introduce region ownership as a way to improve modularity.

Motivation Let's motivate ownership using an example. Consider class *Pair*, which contains two *Long* fields:

```
class Pair [ρ1: Long, ρ2: Long]
{
  left: [ρ1];
  right: [ρ2];
}
```

Regions ρ_1 and ρ_2 are *region parameters* of class *Pair*. This is similar to the type parameters of ML, and is a form of polymorphism: class *Pair* can be used with several regions for its *left* and *right* fields.

Now consider function *incrPair*, which increments the two fields of a *Pair* object:

```
fun incrPair [ρl: Long, ρr: Long, ρp: Pair [ρl, ρr] ] (p: [ρp])
  consumes ρp× ρl× ρr×
  produces ρp× ρl× ρr×
{
  let l = p.left;
  l.value ← l.value + 1;
  let r = p.right;
  r.value ← r.value + 1;
}
```

As you can see, function *incrPair* must take three region parameters: not only the region ρ_p of its *Pair* argument p, but also the regions ρ_l and ρ_r of the *left* and *right* fields. On more complex data structures, it quickly becomes tedious to enumerate all involved regions. Moreover, it is impossible in some cases such as linked lists where the number of involved objects is not known statically.

But the most important motivation for ownership is being able to write data invariants which depend on more locations than just the fields of the class. Consider the following invariant for class *Pair*:

invariant $left.value < right.value$

To maintain such an invariant, we have to ensure that field *value* of pointers *left* and *right* cannot be modified without the invariant being checked. As we have shown in Section 2.5, ownership is one way to ensure this property.

Ownership It is natural to think of fields *left* and *right* as *belonging* to their *Pair* object. They are *owned* by it. Instead of using region parameters for fields *left* and *right*, Capucine allows class *Pair* to *encapsulate* their regions using region ownership:

class *Pair*
{
 single ρ_l: *Long*, ρ_r: *Long*;
 left: $[\rho_l]$;
 right: $[\rho_r]$;
}

This class declaration contains two *owned singleton regions* ρ_l and ρ_r, which are used for fields *left* and *right* respectively.

If given a *Pair* object, say a variable x of type $[\rho]$ with ρ being a *Pair* region, region expressions $x.\rho_l$ and $x.\rho_r$ denote the owned regions of x. Thus $x.left$ has type $[x.\rho_l]$ and $x.right$ has type $[x.\rho_r]$.

Permission $\rho^\times$ encapsulates permissions on owned regions $x.\rho_l{}^\times$ and $x.\rho_r{}^\times$ for x of type $[\rho]$. To access these permissions we need to *unpack* x using the **unpack** operation. This operation consumes $\rho^\times$ and produces ρ°, $x.\rho_l{}^\times$ and $x.\rho_r{}^\times$. Permission ρ° denotes that ρ is singleton, but is currently unpacked. The opposite operation is **pack**. Packing x consumes ρ°, $x.\rho_l{}^\times$ and $x.\rho_r{}^\times$ and produces $\rho^\times$.

Function *incrPair* can then be implemented as:

fun *incrPair* $[\rho_p$: *Pair*$]$ (p: $[\rho_p]$)
 consumes $\rho_p{}^\times$
 produces $\rho_p{}^\times$
{

unpack p;	$p.\rho_l{}^\times$, $p.\rho_r{}^\times$, $\rho_p{}^\circ$
let $l = p.left$;	$p.\rho_l{}^\times$, $p.\rho_r{}^\times$, $\rho_p{}^\circ$
$l.value \leftarrow l.value + 1$;	$p.\rho_l{}^\times$, $p.\rho_r{}^\times$, $\rho_p{}^\circ$
let $r = p.right$;	$p.\rho_l{}^\times$, $p.\rho_r{}^\times$, $\rho_p{}^\circ$
$r.value \leftarrow r.value + 1$;	$p.\rho_l{}^\times$, $p.\rho_r{}^\times$, $\rho_p{}^\circ$
pack p;	$\rho_p{}^\times$

}

To assign $l.value$ and $r.value$, we need permissions on $p.\rho_l$ and $p.\rho_r$. This is why we unpack p at the beginning of the function. We pack it at the end to restore permission $\rho_p{}^\times$.

Allocation Allocating a new *Pair* pointer p in a region ρ produces permissions ρ°, $p.\rho_l{}^\emptyset$ and $p.\rho_r{}^\emptyset$. Indeed, the object is not initialized, and so the ρ_l and ρ_r regions are empty. Region ρ is still open, as to be packed it requires permissions $p.\rho_l{}^\times$ and $p.\rho_r{}^\times$.

We extend permission ρ° to carry information about which fields are initialized. We denote by $\rho^\circ\{f_1, \cdots, f_k\}$ the permission stating that ρ contains a unique location to an

object whose fields are all initialized but $f_1, \cdots, f_k$. Permission ρ° is a shortcut for $\rho^\circ\{\}$. Assignment $x.f \leftarrow e$ removes f from the set of uninitialized fields.

Here is a *constructor* for a *Pair*:

```
fun create [ρp: Pair] (): [ρp]
  consumes ρp^∅
  produces ρp^×
{
  let p = new [ρp];                                  p.ρl^∅, p.ρr^∅, ρp^∘{left, right}
  let l = new [p.ρl];                                p.ρl^∘{value}, p.ρr^∅, ρp^∘{left, right}
  l.value ← 0;                                       p.ρl^∘, p.ρr^∅, ρp^∘{left, right}
  pack l;                                            p.ρl^×, p.ρr^∅, ρp^∘{left, right}
  p.left ← l;                                        p.ρl^×, p.ρr^∅, ρp^∘{right}
  let r = new [p.ρr];                                p.ρl^×, p.ρr^∘{value}, ρp^∘{right}
  r.value ← 0;                                       p.ρl^×, p.ρr^∘, ρp^∘{right}
  pack r;                                            p.ρl^×, p.ρr^×, ρp^∘{right}
  p.right ← r;                                       p.ρl^×, p.ρr^×, ρp^∘
  pack p;                                            ρp^×
  return p;                                          ρp^×
}
```

Modularity Ownership opens up abstraction possibilities. Indeed, objects encapsulate their owned objects. Functions which do not access the owned objects need not know about their existence. For instance, consider the following function:

```
fun twice [ρp: Pair] (p: [ρp])
  consumes ρp^×
  produces ρp^×
{
  incrPair [ρp] (p);
  incrPair [ρp] (p);
}
```

Function *twice* does not need to know anything about the implementation of *Pair* and of *incrPair*. This means that the *Pair* type could be abstract: fields *left* and *right*, as well as owned regions ρ_l and ρ_r, could be hidden and yet function *twice* would not need to be adapted.

Several module systems could be considered to extend Capucine with abstraction: refinement and functors for instance are good candidates. The choice is orthogonal to the work of this thesis though, and we will not discuss it here.

3.3 Invariants Using Permissions and Ownership

Capucine features object invariants. Permissions are used to track which invariants hold, and ownership is used to allow invariants to depend on other objects.

Permissions Track Invariant States Let's illustrate the invariant mechanism on a simple example: positive integers. We define class *PosInt*:

```
class PosInt
{
  value: int;
  invariant value ≥ 0;
}
```

This class has only one integer field *value* which, according to the invariant, should always be positive (or zero).

To track the state of invariants, we use permissions ρ° and $\rho^\times$. They have already been introduced in previous sections. They both imply region ρ is singleton, but ρ° means that ρ is *open* while $\rho^\times$ means that ρ is *closed*. Until now, we only used this distinction because of ownership. Now we add a new meaning to permission $\rho^\times$: if $\rho^\times$ is available, the pointer of ρ must verify its invariant. If ρ° is available, the invariant may be broken. The invariant is checked when packing.

Moreover, we now require permission ρ° to assign the fields of the pointer of ρ. This is the same requirement than the one which was used in Section 2.5: only open pointers may be modified, as their invariant is not required to hold. However, instead of using a special `state` field we use permissions.

Here is an example of function which maintains the invariant of a *PosInt* object:

```
fun incr [ρ: PosInt] (p: [ρ]): unit
  consumes ρ^×
  produces ρ^×
{
  unpack p;                                  ρ^°
  p.value ← p.value + 1;                     ρ^°
  pack p;                                    ρ^×
}
```

Because *incr* consumes permission $\rho^\times$, the invariant of p must hold when *incr* is called. Because *incr* produces permission $\rho^\times$, the invariant is maintained by a call to *incr*.

Here is an example of function which does not require the invariant of a *PosInt* object, but which nevertheless ensures the invariant holds after the call:

```
fun abs [ρ: PosInt] (p: [ρ]): unit
  consumes ρ^°
  produces ρ^×
{
  if p.value < 0 then
  {
    p.value ← −p.value;
  };
  pack p;
}
```

Because this function consumes ρ°, and not $\rho^\times$, the invariant of p is not required to hold at the beginning. However, it holds at the end, and this is reflected by the fact that $\rho^\times$ is produced by *abs*.

A typical example of function which initializes the invariant of an object is its *constructor*, i.e. a function which allocates and initializes the object:

```
fun create [ρ: PosInt] (i: int): [ρ]
  pre i ≥ 0
  consumes ρ^∅
  produces ρ^×
{
  let p = new [ρ];                                   ρ°{value}
  p.value ← i;                                       ρ°
  pack p;                                            ρ^×
}
```

This constructor takes an empty region ρ and an integer i. The pre-condition requires i to be positive or null. The function returns a new pointer, allocated in region ρ, with initial value i. The invariant is thus verified, and this is reflected by the fact that $\rho^\times$ is produced. Note that allocation produces permission ρ°, and not $\rho^\times$. This is important in our system. Indeed, allocation does not initializes invariants.

An object of a group region is always closed. Permission ρ^G states that all objects of ρ verify their invariant.

Ownership Extends Invariant Expressiveness If the invariant of an object A depends on the value of an object B, then B should not be modified without checking that the modification does not break the invariant of A. As we have already discussed in Section 2.5, ownership is a means to achieve exactly this goal: if A owns B, and if the ownership methodology prevents B from being modified if the invariant of A is supposed to hold, then the invariant of A cannot be broken.

In Capucine, the invariant of a class may only depend on fields which are transitively owned by the class. A field is transitively owned by a class if either it is a field of the class, or it is the field of an object which is in a region owned by the class. For instance, if a class $\mathcal{C}$ owns a region r of class $\mathcal{C}'$, if $\mathcal{C}$ has a field f of type $[r]$, and if $\mathcal{C}'$ has a field g, then $\mathcal{C}$ transitively owns $f.g$ and its invariant may thus depend on $f.g$.

Then the region ownership system of Capucine prevents invariants from being broken unexpectedly. Indeed, to modify an object B owned by an object A, we require the open permission ${\rho_b}^\circ$ on the region ρ_b of object B. But this permission can only be available if A has been unpacked first, and thus if permission ${\rho_a}^\circ$ on the region ρ_a of object A is available. Thus the invariant of A is not expected to hold when B can be modified.

For instance, let's add an invariant to class *Pair*:

```
class Pair
{
  single ρ_l: Long, ρ_r: Long;
  left: [ρ_l];
  right: [ρ_r];
  invariant left.value < right.value;
}
```

The invariant only depends on the *value* field of the *left* and *right* fields, which are owned by the *Pair*, so the invariant is valid.

As we have introduced the new requirement that pointers should be open before their fields are modified, we have to rewrite *incrPair*:

```
fun incrPair [ρ_p: Pair] (p: [ρ_p])
```

```
  consumes ρ_p^×
  produces ρ_p^×
{
  unpack p;                                          p.ρ_l^×, p.ρ_r^×, ρ_p^°
  let l = p.left;                                    p.ρ_l^×, p.ρ_r^×, ρ_p^°
  unpack l;                                          p.ρ_l^°, p.ρ_r^×, ρ_p^°
  l.value ← l.value + 1;                             p.ρ_l^°, p.ρ_r^×, ρ_p^°
  pack l;                                            p.ρ_l^×, p.ρ_r^×, ρ_p^°
  let r = p.right;                                   p.ρ_l^×, p.ρ_r^×, ρ_p^°
  unpack r;                                          p.ρ_l^×, p.ρ_r^°, ρ_p^°
  r.value ← r.value + 1;                             p.ρ_l^×, p.ρ_r^°, ρ_p^°
  pack r;                                            p.ρ_l^×, p.ρ_r^×, ρ_p^°
  pack p;                                                          ρ_p^×
}
```

As you can see, before modifying $p.left.value$, we have to unpack $p.left$, and before unpacking $p.left$ we have to unpack p. So the invariant of p is not required to hold before the **pack** p instruction. In fact, this invariant can be temporarily broken if $p.left.value = p.right.value - 1$ at the beginning of the function. But the invariant is re-established before p is packed and its invariant is required to hold again.

3.4 Examples

3.4.1 Bounded Arrays

As a first example, we define the *bounded arrays* data structure in Capucine. Bounded arrays are arrays of *constent length*. Pre-conditions prevent reading or writing outside of the array.

Logic Arrays First we define *logic arrays* in Capucine. This is the theory of arrays which we introduced in Section 2.4.1. Here is the corresponding Capucine code:

type *larray* (α)

logic *store* (a: *larray* (α), i: *int*, v: α): *larray* (α)

logic *select* (a: *larray* (α), i: *int*): α

axiom *selectEq*:
$\forall a$: *larray* (α).
$\forall i$: *int*.
$\forall v$: α.
select(*store*(a, i, v), i) = v

axiom *selectNeq*:
$\forall a$: *larray* (α).
$\forall i$: *int*.
$\forall j$: *int*.
$\forall v$: α.

$i \neq j \Rightarrow$
$select(store(a,\ i,\ v),\ j) = select(a,\ j)$

This code illustrates how Capucine features logic type declarations using the **type** keyword, logic functions declarations using the **logic** keyword and axiomatizations of them using the **axiom** keyword. This allows the user to introduce new pure types using first-order logic axiomatizations. Here, arrays are polymorphic: the α argument of type *larray* is a type parameter as in ML.

Bounded Arrays: Class Definition We now define mutable *bounded arrays* in Capucine. A bounded array is a reference to a logic array and a size.

```
class array (α)
{
  length: int;
  contents: larray (α);
  invariant length ≥ 0
}
```

The invariant states that the size of an array is always positive or null.

Bounded Arrays: Creation Function *arrayCreate* returns a new bounded array. It takes a region argument ρ in which the new array will be created, and an integer *size* which will be the size of the array.

```
fun arrayCreate [ρ: array (α)] (size: int): [ρ]
  consumes ρ^∅
  produces ρ^×
  pre size ≥ 0
  post result.length = size
{
  let res = new array (α) [ρ];
  res.length ← size;
  return res
}
```

As the **consumes** and **produces** clause states, region ρ must be empty before the call and is singleton after. Indeed, in Capucine pointers are allocated in empty regions, which thus become singleton. Note that we omitted the **pack** *res* statement, which is inferred by Capucine just before the **return** statement.

Bounded Arrays: Reading Function *arrayGet* takes a bounded array a in a region ρ and an index i, and returns the item stored at index i:

```
fun arrayGet [ρ: array (α)] (a: [ρ], i: int): α
  consumes ρ^×
  produces ρ^×
  pre 0 ≤ i ∧ i < a.length
  post result = select(a.contents, i)
{
```

```
  return select(a.contents, i)
}
```

This function returns the stored value using logic function *select*. Function *arrayGet* also requires the index i to be inside the array. This is how we model array bound checks.

Bounded Arrays: Writing Similarly to the *arrayGet* function, *arraySet* encapsulates logic function *store* for bounded arrays:

```
fun arraySet [ρ: array (α)] (a: [ρ], i: int, v: α): unit
  consumes ρ^×
  produces ρ^×
  pre 0 ≤ i ∧ i < a.length
  post
    a.contents = store(a.contents@pre, i, v) ∧
    a.length = a.length@pre
{
  use invariant a;
  unpack a;
  a.contents ← store(a.contents, i, v);
  pack a;
}
```

The post-condition states that the *length* field has not changed. Term t@**pre** is term t at position **pre**, which is the position corresponding to the beginning of the function. It is thus the value of t before the call. The post-condition also states how *contents* is modified.

Statement **use invariant** a requires permission $\rho^\times$, which is indeed required by function *arraySet*, and introduces the invariant of a in the hypotheses of proof obligations. Here, there are two proof obligations: the post-condition and the invariant. Indeed, the **pack** statement generates the invariant of the new a as a proof obligation. To prove this invariant (size is positive or null) we use the hypothesis introduced by the **use invariant** statement (size was positive or null).

3.4.2 Bounded Arrays: Proof Obligations

We now show the verification conditions which are produced by the bounded arrays example to give an intuition of what the user has to prove and what the memory model looks like.

Regions, Locations and Objects We use the approach of Section 2.4.4 to model region. Regions are encoded as maps from locations to *objects*. Their type is *region* (α) where α is a type parameter corresponding to the type of objects, which we will hide in this section to ease reading. Locations are values of a type called *location*. An object is a tuple where each component corresponds to a field of the data structure.

Function *get* is similar to *select*, and function *set* is similar to *store*. Term $get(r, p)$ returns the object at location p in region r; and $set(r, p, o)$ returns a copy of region r where the object at location p is now o.

Chapter 4 formalizes the memory model, i.e. the notion of region and objects. Chapter 5 formalizes how verification conditions are computed.

Bounded Arrays: Creation The inferred **pack** statement of function *arrayCreate* generates the invariant of *res* as a proof obligation, which is ensured by the pre-condition. Here is the (slightly simplified) logic formula:

$$\begin{gathered}\forall \rho\text{: } region, \forall size\text{: } int, \forall res\text{: } location,\\ size \geq 0 \Rightarrow\\ length(get(\rho,\ res)) = size \Rightarrow\\ length(get(\rho,\ res)) \geq 0\end{gathered}$$

This formula is valid and is simple enough to be discharged automatically by most SMT solvers. We obtain the object at location *res* in region ρ using $get(\rho,\ res)$. We read the *length* field of this object using a function which we call *length*.

The second proof obligation of function *arrayCreate* corresponds to the post-condition:

$$\begin{gathered}\forall \rho\text{: } region, \forall size\text{: } int, \forall res\text{: } location,\\ size \geq 0 \Rightarrow\\ length(get(\rho,\ res)) = size \Rightarrow\\ length(get(\rho,\ res)) = size\end{gathered}$$

This formula is valid as well.

Bounded Arrays: Reading There is only one proof obligation for function *arrayGet*, which corresponds to the post-condition:

$$\begin{gathered}\forall \rho\text{: } region, \forall a\text{: } location, \forall i\text{: } int,\\ 0 \leq i < length(get(\rho,\ a)) \Rightarrow\\ select(contents(get(\rho,\ a)),\ i) = select(contents(get(\rho,\ a)),\ i)\end{gathered}$$

The *contents* function returns the value of the *contents* field, i.e. a logic array. This formula is valid.

Bounded Arrays: Writing Here is the proof obligation due to the **pack** statement of function *arraySet*:

$$\begin{gathered}\forall \rho\text{: } region, \forall a\text{: } location, \forall v\text{: } \alpha, \forall i\text{: } int,\\ 0 \leq i < length(get(\rho,\ a)) \Rightarrow\\ length(get(\rho,\ a)) \geq 0 \Rightarrow\\ \forall \rho'\text{: } region,\\ \rho' = set(\rho,\ a,\ setcontents(get(\rho,\ a),\ store(contents(get(\rho,\ a)),\ i,\ v))) \Rightarrow\\ length(get(\rho',\ a)) \geq 0\end{gathered}$$

Hypothesis $length(get(\rho,\ a)) \geq 0$ was introduced by the **use invariant** statement. The new value ρ' of region ρ is due to the assignment. Function *setcontents* takes an object and modifies its *contents* field, leaving its *length* field unchanged. This is part of the memory model, and axioms are provided to state this separation property. This formula is, once again, valid and can be proved automatically by most SMT solvers.

The second proof obligation of function *arraySet* corresponds to the post-condition. It has similar hypotheses, but instead of having to prove that the length is positive or null, we need to prove the post-condition:

$$\begin{array}{c}\forall \rho\text{: } region, \forall a\text{: } location, \forall v\text{: } \alpha, \forall i\text{: } int,\\ 0 \leq i < length(get(\rho,\ a)) \Rightarrow\\ length(get(\rho,\ a)) \geq 0 \Rightarrow\\ \forall \rho'\text{: } region,\\ \rho' = set(\rho,\ a,\ setcontents(get(\rho,\ a),\ store(contents(get(\rho,\ a)),\ i,\ v))) \Rightarrow\\ contents(get(\rho',\ a)) = store(contents(get(\rho,\ a)),\ i,\ v)\\ \wedge\ length(get(\rho',\ a)) = length(get(\rho,\ a))\end{array}$$

This formula is also valid and can be proved automatically by theorem provers.

Component-as-Array Another version of the proof obligation due to the **pack** statement of function *arraySet* can be obtained by using further separation techniques inspired by the component-as-array model [Bornat00]:

$$\begin{array}{c}\forall \rho_{length}\text{: } region, \forall \rho_{contents}\text{: } region, \forall a\text{: } location, \forall v\text{: } \alpha, \forall i\text{: } int,\\ 0 \leq i < get(\rho_{length},\ a) \Rightarrow\\ get(\rho_{length},\ a) \geq 0 \Rightarrow\\ \forall \rho_{contents}{}'\text{: } region,\\ \rho_{contents}{}' = set(\rho_{contents},\ a,\ store(get(\rho_{contents},\ a),\ i,\ v)) \Rightarrow\\ get(\rho_{length},\ a) \geq 0\end{array}$$

The idea is that ρ is decomposed into ρ_{length} and $\rho_{contents}$, where ρ_{length} is a map from pointers to the value of the *length* field of their object, and $\rho_{contents}$ is a map from pointers to the value of the *contents* field of their object. This version does not require axioms stating that *length* and *contents* are separated, and is thus much easier to prove. It is valid and can be proved automatically.

3.4.3 Sparse Arrays

In this section we illustrate the Capucine approach on the *constant-time sparse arrays* challenge of the VACID benchmarks [Leino10]. These benchmarks are a collection of short programs proposed by K. Rustan M. Leino and Michał Moskal which illustrate some of the challenges in modular deductive verification. They involve data structures with aliases as well as data invariants.

The Sparse Arrays Challenge The sparse arrays challenge is posed on the code of Figure 3.1. It is written using a syntax close to Java, but the use of classes is not important and, as indicated in the article [Leino10], we assume arrays allocated using **new** are *not initialized*. Array are of constant length. We have altered the original code slightly, by replacing the implicit array allocation of Java by explicit allocations in the constructor. This allows the creation of arrays of any size `sz` given as parameter, instead of a constant size `MAXLEN`.

Class `SparseArray` implements a *sparse array* data structure. It provides the usual array functions `create`, `get` and `set`. What is interesting about *sparse* arrays is that, as read and write functions `get` and `set` do, allocation `create` runs in constant time. Indeed, `create` does not initialize internal arrays, yet reading an uninitialized index with `get` returns `DEFAULT`.

Sparse arrays are implemented using three arrays `val`, `idx` and `back` and two integers `size` and `n`. Array `val` contains actual values of the sparse array. Value `n` is the number of

```
class SparseArray {
  static final int DEFAULT = 0;
  int val[];
  uint idx[], back [];
  uint n, uint size;

  static SparseArray create(uint sz) {
    SparseArray t = new SparseArray();
    val = new int[sz];
    idx = new uint[sz];
    back = new uint[sz];
    n = 0;
    size = sz;
    return t;
  }

  int get(uint i) {
    if (idx[i] < n && back[idx[i]] == i)
      return val[i];
    else
      return DEFAULT;
  }

  void set(uint i, int v) {
    val[i] = v;
    if (!(idx[i] < n && back[idx[i]] == i)) {
      assert(n < size);
      idx[i] = n;
      back[n] = i;
      n = n + 1;
    }
  }

  static void sparseArrayTestHarness() {
    SparseArray a = create(10), b = create(20);
    assert(a.get(5) == DEFAULT && b.get(7) == DEFAULT);
    a.set(5, 1); b.set(7, 2);
    assert(a.get(0) == DEFAULT && b.get(0) == DEFAULT);
    assert(a.get(5) == 1 && b.get(7) == 2);
    assert(a.get(7) == DEFAULT && b.get(5) == DEFAULT);
  }
}
```

Figure 3.1: Pseudo-Java code for the Sparse Arrays challenge

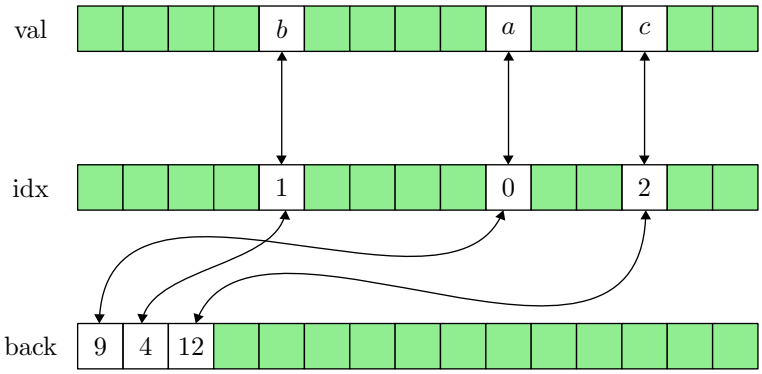

Figure 3.2: Example of State of a Sparse Array Structure

different cells which have been assigned using `set`. Array `back` contains, in cells 0 to $n-1$, the indexes of these assigned cells. Array `idx` allows to know where in `back` these indexes appear. Thus, cells 0 to $n-1$ of `back` are in bijection with the corresponding cells of `idx`.

Figure 3.2 illustrates the state of the structure after three items a, b and c have been added, in this order, in indexes 9, 4 and 12 respectively. Darker cells are cells which have not been assigned yet. First item a, of index 9, was associated with index 0 in array `idx`, and its actual index 9 is stored at index 0 of array `back`.

Method `get` reads index d using the following method. First, read the internal index i stored in `idx[d]`. Then, if not $0 \leq i < n$ then we are sure that cell `i` has not been initialized and thus that no item was inserted at index d, and the default value should be returned. Else if $0 \leq i < n$ then read index d' stored in `back[i]`. If $d' = d$ then an item has indeed been inserted at index d, so `val[d]` should be returned. Else the default value should be returned.

The **`assert`** statements are verifications to be made. A first one appears in method `set`. The others appear in method `sparseArrayTestHarness` which tests the structure on simple data. This could feel easy at first sight — indeed it would be sufficient to run the program and check that these assertions hold — but the goal is to verify the program modularly and statically. We should thus be capable of specifying pre- and post-conditions for methods `get` and `set`, sufficient to establish the assertions.

Please note that we have added an additional assertion to the original code [Leino10]: the last one, which is the only one which tests that `a` and `b` are actually separated. Indeed, what makes this program even harder to verify is that method `create` must be specified such that variables `a` and `b` of method `sparseArrayTestHarness` do not share data. For instance, how can we ensure that `a.idx` and `b.idx` are not the same arrays? Capucine and its type system using regions and permissions provides a methodology to annotate the program using such separation information.

Sparse Arrays: Class Definition We use the bounded arrays that we introduced in Section 3.4.1 to implement sparse arrays:

class *sparse* (α)
{

single *Rvalue*: *array* (α), *Ridx*: *array* (*int*), *Rback*: *array* (*int*);

value: [*Rvalue*];
idx: [*Ridx*];
back: [*Rback*];
n: *int*;
default: α;
size: *int*;

invariant
$0 \leq n \leq size \wedge$
$value.length = size \wedge$
$idx.length = size \wedge$
$back.length = size \wedge$
$\forall i$: *int*.
$0 \leq i < n \Rightarrow$
$0 \leq select(back.contents, i) < size \wedge$
$select(idx.contents, select(back.contents, i)) = i$
}

Class *sparse* owns three singleton regions: *Rvalue*, *Ridx* and *Rback*. It contains six fields. The first three are the three internal arrays of the sparse array data structure: *value*, *idx* and *back*. They are implemented using our bounded arrays, and are thus pointers. Their respective regions are *Rvalue*, *Ridx* and *Rback*. Field *n* is the number of inserted values in the sparse array. Field *default* replaces constant `DEFAULT` of the original code. It is necessary as our sparse arrays are polymorph (type variable α). The default value will be given to the constructor. Finally, field *size* contains the size of the sparse array.

The invariant states that *n* is between 0 and *size* (included). If $n = size$, the array is full. The invariant also states that the length of internal arrays is *size*. This will be used to prove the pre-condition when calling *arrayGet*. Finally, the invariant states the relation between *idx* and *back*, which is that for all already-inserted index *i*, we have $idx[back[i]\,] = i$.

Sparse Arrays: Model We express the specification of our sparse array functions using an abstract model defined as follows. First we introduce the *isElt* predicate, which takes an array *a* and an integer *i*, and states that *i* was inserted in *a* before:

logic *isElt* [ρ: *sparse* (α)] (*a*: [ρ], *i*: *int*) =
$0 \leq select(a.idx.contents, i) < a.n \wedge$
$select(a.back.contents, select(a.idx.contents, i)) = i$

As *isElt* takes pointer *a* as an argument, it also takes its region ρ. Indeed, a pointer can only be read from a given region. Here, *a.idx* actually means that we read the *idx* field of pointer *a* in region ρ. And *a.idx.contents* actually means that we read the *contents* field of pointer *a.idx* in region *a.Ridx*, which is itself read from the *Ridx* region of pointer *a* in region ρ.

Then we axiomatize the *model* logic function, which takes a sparse array *a* and an integer *i*, and returns the value stored at index *i* in *a*. It is thus the equivalent of *select* for sparse arrays.

logic *model* [ρ: *sparse* (α)] (*a*: [ρ], *i*: *int*): α

Again, logic function *model* also takes a region ρ as parameter, as it reads fields from pointer argument a. This logic function is defined using two axioms:

axiom *modelIn*:
 $\forall$**region** ρ: *sparse* (α).
 $\forall a$: $[\rho]$.
 $\forall i$: *int*.
 isElt $[\rho]$ $(a, i) \Rightarrow$
 model $[\rho]$ (a, i) = *select*$(a.value.contents, i)$

axiom *modelOut*:
 $\forall$**region** ρ: *sparse* (α).
 $\forall a$: $[\rho]$.
 $\forall i$: *int*.
 $\neg$*isElt* $[\rho]$ $(a, i) \Rightarrow$
 model $[\rho]$ $(a, i) = a.default$

These axioms state that if i was inserted in a, then its value is stored in the *value* array, otherwise its value is the default value. Note how region ρ is quantified to allow the axiom to be applied to whatever region array a is in.

Sparse Arrays: Creation The Capucine code for the sparse array constructor *create* is:

fun *create* $[\rho$: *sparse* $(\alpha)]$ $(sz$: *int*, *def*: $\alpha)$: $[\rho]$
 consumes $\rho^{\emptyset}$
 produces $\rho^{\times}$
 pre $0 \leq sz$
 post
  result.*size* = $sz \wedge$
  $\forall i$: *int*. *model* $[\rho]$ (**result**, i) = *def*
{
 let a = **new** *sparse* (α) $[\rho]$;
 let *value* = *arrayCreate* $[a.Rvalue]$ (sz);
 $a.value \leftarrow value$;
 let *idx* = *arrayCreate* $[a.Ridx]$ (sz);
 $a.idx \leftarrow idx$;
 let *back* = *arrayCreate* $[a.Rback]$ (sz);
 $a.back \leftarrow back$;
 $a.n \leftarrow 0$;
 $a.default \leftarrow def$;
 $a.size \leftarrow sz$;
 assert $\forall i$: *int*. $\neg$(*isElt* $[\rho]$ (a, i));
 return a
}

It expects an empty region ρ and allocates a new sparse array in it. A **pack** statements is inferred before the **return** statement. The **assert** statement helps automatic provers to establish the proof obligations. There are two: one for the invariant of the returned sparse array, and one for the post-condition, which states that the returned array is of size sz and no value has been inserted yet.

Sparse Arrays: Reading The code for reading is:

```
class Ref (α)
{
  refValue: α;
}

fun sget [ρ: sparse (α)] (a: [ρ], i: int): α
  consumes ρ^×
  produces ρ^×
  pre 0 ≤ i < a.size
  post result = model [ρ] (a, i)
{
  use invariant a;
  let index = arrayGet(a.idx, i);
  let region Rres: Ref (α);
  let res = new sparse (α) [Rres];
  if 0 ≤ index ∧ index < a.n then
  {
    let backIndex = arrayGet(a.back, index);
    if backIndex = i then
    {
      let value_i = arrayGet(a.value, i);
      res.refValue ← value_i;
    }
    else
    {
      res.refValue ← a.default;
    };
  }
  else
  {
    assert ¬(isElt [ρ] (a, i));
    res.refValue ← a.default;
  };
  return res.refValue
}
```

The *Ref* class is introduced because in Capucine, the **return** statement can only be put once, at the very end of functions. So we need a temporary variable to store the result. And in Capucine, **let**-bound variables are immutable. So we define a simple reference class and use it to store the result. The *sget* function requires permission $\rho^\times$ even though it does not modify the array, because reading also requires some permission, and because it requires the invariant of a. Again, the **assert** statement is here to help automatic provers. This function simply follows the definition of logic function *model*, and thus the post-condition holds.

Sparse Arrays: Writing The code for writing is:

```
fun sset [ρ: sparse (α)] (a: [ρ], i: int, v: α): unit
  consumes ρ^×
  produces ρ^×
  pre 0 ≤ i < a.size
  post
    a.size = a.size@pre ∧
    (∀j: int. j ≠ i ⇒ model [ρ] (a, j) = model [ρ] (a, j)@pre) ∧
    model [ρ] (a, i) = v
{
  use invariant a;
  let x = arraySet [a.Rvalue] (a.value, i, v);
  let index = arrayGet [a.Ridx] (a.idx, i);
  if 0 ≤ index ∧ index < a.n then
  {
    let backIndex = arrayGet [a.Rback] (a.back, index);
    if ¬(backIndex = i) then
    {
      assert a.n < a.size;
      let x = arraySet [a.Ridx] (a.idx, i, a.n);
      let x = arraySet [a.Rback] (a.back, a.n, i);
      a.n ← a.n + 1;
    };
  }
  else
  {
    assert a.n < a.size;
    let x = arraySet [a.Ridx] (a.idx, i, a.n);
    let x = arraySet [a.Rback] (a.back, a.n, i);
    a.n ← a.n + 1;
  };
  assert isElt [ρ] (a, i);
}
```

The post-condition not only states that the value of index i in a is now v, it also states that all other values have not been changed. Proof obligations are: the invariant, and the post-condition. Again, the three **assert** statements are here to help the provers prove these proof obligations. However, the two $a.n < a.size$ assertions are not proved automatically. It requires to prove the so-called *pigeon-hole lemma*. We proved this using the Coq proof-assistant. A lot of code is duplicated in this implementation because our prototype implementation of Capucine does not support side-effects, and thus calls, in expressions. With this features, we could use a lazy conjunction to avoid duplication.

Sparse Arrays: Test Harness The test harness code is implemented as follows:

```
fun test(x: unit): unit
{
  let region Ra: sparse (int);
  let region Rb: sparse (int);
  let default = 0;
```

```
  let a = create [Ra] (10, default);
  let b = create [Rb] (10, default);
  let x = sget [Ra] (a, 5);
  let y = sget [Rb] (b, 7);
  assert x = default ∧ y = default;
  let ignore = sset [Ra] (a, 5, 1);
  let ignore = sset [Rb] (b, 7, 2);
  let x = sget [Ra] (a, 5);
  let y = sget [Rb] (b, 7);
  assert x = 1 ∧ y = 2;
  let x = sget [Ra] (a, 7);
  let y = sget [Rb] (b, 5);
  assert x = default ∧ y = default;
  let x = sget [Ra] (a, 0);
  let y = sget [Rb] (b, 0);
  assert x = default ∧ y = default;
  let x = sget [Ra] (a, 9);
  let y = sget [Rb] (b, 9);
}
```

The *ignore* variables are only there because the call statement in Capucine is syntactically required to be **let**-bound. All assertions are proved automatically.

There is one proof obligation per **assert** statement, and one for each call to *sget* and *sset* to check array bounds. In all proof obligations, sparse arrays a and b are *separated*. Indeed, they belong to two different regions, which are encoding using two different variables in the proof obligations. This means that when one of the array is modified, the other is left unchanged, and this fact can immediately be used very easily in proof obligations.

Conclusion All proof obligations except the $a.n < a.size$ assertions of the *sset* function are proved automatically. We proved the more difficult assertions using the Coq proof assistant. To sum up, the specification is proved entirely, including safety of array bounds. The region separation provided by Capucine helps with proof obligations, which are either pre-conditions at call sites, user assertions or invariants when packing objects. All of these are intuitive cases where a proof obligation makes sense.

3.4.4 Function Memoization Using Hash Tables

The goal of our next example is to encode and prove in Capucine the Java program of Figure 3.3, which is a data structure which computes the Fibonacci function. Memoization is used to speed up computation: results are stored in a table to avoid computing them again later on. This is obviously not the most efficient way to encode the Fibonacci function, but this memoization technique applies to any function, and illustrates some features of Capucine.

The main problem is that once keys are entered in the table, they should not be modified or the table data structure will be broken. Another issue is that the table manipulates references to the values which are associated to keys, but the table should not own these values.

Long We need the class *Long* that we have already introduced before:

```
class FibMemo {
  private HashMap<Long,Long> memo;

  FibMemo () {
    memo = new HashMap<Long,Long>();
  }

  long fib(long n) {
    if (n <= 1) { return 1; }
    Long x = memo.get(n);
    if (x != null) { return x.longValue(); }
    Long y = fib(n-1) + fib(n-2);
    memo.put(n,y);
    return y.longValue();
  }
}
```

Figure 3.3: Java code for the memoized Fibonacci function

class *Long*
{
value: *int*;
}

Options We now introduce the *option* type. An option is either *none* or *some*(x), where x is any term. We may retrieve the value x using logic function *getSome*, which is unspecified on value *none*.

type *option* (α)

logic *none*: *option* (α)
logic *some* (x: α): *option* (α)

logic *getSome* (x: *option* (α)): α

axiom *getSomeSome*:
$\forall x$: α.
getSome (*some* (x)) = x

axiom *noneOrSome*:
$\forall x$: *option* (α).
x = *none* $\vee$ $\exists y$: α. x = *some* (y)

axiom *noneXorSome*:
$\forall x$: α. *none*() $\neq$ *some* (x)

Maps We also introduce the logic type of *finite maps* and its axiomatization, which is similar to the theory of arrays we introduced in Section 3.4.1 except that the keys to the array are also polymorphic, that we have the empty map at our disposal and that function *select* returns an option. It returns *none* if the required key is not in the map, and *some*(x) where x is the associated value to the key if the key is in the map.

type *map* (α, β)

logic *emptyMap*: *map* (α, β)
logic *store* (x: *map* (α, β), y: α, z: β): *map* (α, β)
logic *select* (x: *map* (α, β), y: α): *option* (β)

We shall not write the axioms again here.

Table We now introduce the *Table* class. We implement it using maps. Its goal is to simulate a more complex implementation using hash tables, for instance.

class *Table* [*Rval*: *Long*]
{
  group *Rkeys*: *Long*;
  contents: *map* ([*Rkeys*], [*Rval*]);
}

The *Table* class is implemented using logic maps from *Long* pointers of region *Rkeys* to *Long* pointers of region *Rval*. Region *Rval* is a parameter of the class: it is not owned. This reflects the fact that the invariant of a more precise implementation of *Table* would not need to depend on the user values in *Rval*. However, the keys to the map are in region *Rkeys* which is owned by the class. Once a key is in the map, one cannot modify the key without unpacking the map first. In other words, all modifications to the keys are controlled by the table itself.

We introduce logic function *model* which allows to read a table without knowledge of the actual implementation. Function *model* assumes that the keys are *Long* values, and accepts an integer i instead of a pointer.

logic *model* [*Rv*: *Long*, *Rt*: *Table* [*Rv*]] (t: [*Rt*], i: *int*): *option* ([*Rv*])

The idea is that class *Table* [*Rval*] and logic function *model* represent a Java-like signature of our tables. The implementation could be hidden: it is not needed to reason about the table. The implementation of *model* is also kept hidden, but we write an axiom to specify that *model* does not depend on the values of pointers in the *Rv* region argument:

axiom *modelFootprint* :
  ∀**region** *Rv1*: *Long*.
  ∀**region** *Rv2*: *Long*.
  ∀**region** *Rh1*: *Table* [*Rv1*].
  ∀**region** *Rh2*: *Table* [*Rv2*].
  ∀h: [_]. ∀i:*int*.
  (*get*(*Rh1*, h, *contents*) = *get*(*Rh2*, h, *contents*)) ⇒
  model [*Rv1*, *Rh1*] (h, i) = *model* [*Rv2*, *Rh2*] (h, i)

Operation *get*(ρ, h, *contents*) reads the field *contents* of pointer h in region ρ. Indeed, the type of h is [_]: its region is unspecified. In the logic a pointer can be read from any region

using *get*. If h is not actually in ρ, the value of $get(\rho, h, contents)$ is unspecified.

Table Creation Function *createTable* creates a table.

```
fun createTable [Rv: Long, Rt: Table [Rv] ] (): [ρ]
  consumes ρ^∅
  produces ρ^×
  post ∀i: int. model[Rv, Rt](result, i) = none
{
  let t = new [Rt];
  t.contents ← emptyMap ();
  return t
}
```

The post-condition states that the table is empty: for each key, the associated model is *none*.

Table Insertion Function *insert* inserts a new key k and its value v into a table t.

```
fun insert [Rv: Long, Rt: Table [Rv], Rk: Long] (t: [Rt], k: [Rk], v: [Rv]): unit
  consumes Rt^× Rk^×
  produces Rt^×
  post
    model [Rv, Rt] (h, k.value) = some (v) ∧
    ∀i: int.
    i ≠ k.value ⇒ model [Rv, Rt] (t, i) = model [Rv, Rt] (t, i)@pre
{
  adopt Rk as t.Rkeys;
  t.contents ← store (t.contents, k, v);
}
```

The adoption inserts the key into the *Rkeys* region owned by the table. As a consequence, permission $Rk^\times$ is consumed, and is no longer available. Hence the **produces** clause, which mentions no permission on *Rk*. *Ownership of the key has been transferred to the table structure.*

The post-condition states that the only key whose associated value has changed is k. It is expressed using *model* to ensure that tables can be used without knowledge of their implementation.

Table Access Function *find* returns the associated value to a key k in a table t.

```
fun find [Rv: Long, Rt: Table [Rv], Rk: Long] (t: [Rt], k: [Rk]): option ([Rv])
  consumes Rt^× Rk^×
  produces Rt^× Rk^×
  post result = model [Rv, Rt] (t, k.value)
```

We do not provide an implementation, as it is not the interesting point of the example. It finds a key of $t.contents$ whose value is the same as $k.value$, and returns the associated data.

Fibonacci Data Structure We now define class *Fibo* which contains a table to store the values of the Fibonacci function which have already been computed. The class is implemented

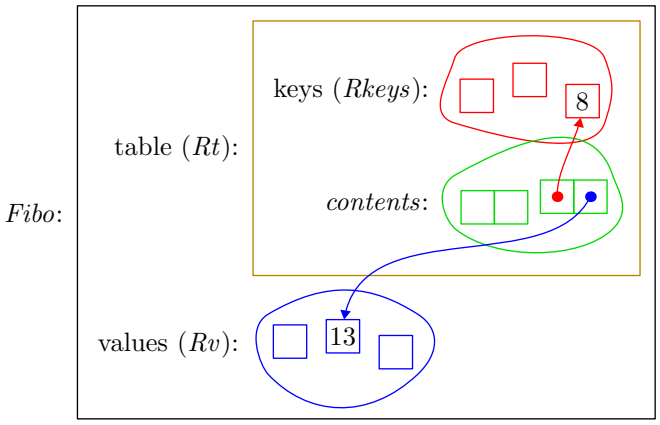

Figure 3.4: The *Fibo* data structure

entirely without ever accessing the *contents* field of the table directly. In other words, class *Fibo* is written without knowledge of the implementation of the *Table* class, thanks to logic function *model*.

First we define the mathematical Fibonacci function as a logic function.

logic *fibo* (x: *int*): *int*

axiom *fibo0*: $fibo(0) = 0$
axiom *fibo1*: $fibo(1) = 1$
axiom *fiboN*: $\forall n$: *int*. $n > 1 \Rightarrow fibo(n) = fibo(n-1) + fibo(n-2)$

Now we can define class *Fibo*.

```
class Fibo
{
  group Rv: Long;
  single Rt: Table [Rv];
  table: [Rt];
  invariant
    ∀x: int.
    ∀y: [Rv].
    model [Rv, Rt] (table, x) = some y ⇒ y.value = fibo(x)
}
```

The invariant states that for all bindings (x, y) in the table, y is the Fibonacci function applied to x. Note that the invariant mentions the values stored in region Rv, which is owned by the *Fibo* class, but there is no field of type Rv: all locations of region Rv are actually stored in the table. Figure 3.4 illustrates the data structure. Note how keys are owned by the table. Also note how values are owned by the *Fibo* data structure, even though the contents of the table may reference them.

We now introduce our main function. The *fibonacci* function takes a *Fibo* pointer as an argument, an integer i and returns $fibo(i)$. First it looks into the table to see if $fibonacci(i)$ has already been computed. If so, it returns the associated value in the table. Else, it

computes the value, inserts it in the table, and returns it.

```
fun fibonacci [ρ: Fibo] (f: [ρ], i: int): int
  consumes ρ^×
  produces ρ^×
  pre i ≥ 0
  post result = fibo(i)
{
  use invariant f;
  let region Rr: Long;
  let r = new [Rr];

  if i ≤ 1 then
  {
    r.value ← i;
  } else {
    let region Rk: Long;
    let k = new Long [Rk];
    k.value ← i;

    let ro = find [f.Rv, f.Rt, Rk] (f.table, k);
    if ro = none then
    {
      let f2 = fibonacci [ρ] (f, i − 2);
      let f1 = fibonacci [ρ] (f, i − 1);
      r.value ← f1 + f2;

      let region Rv: Long;
      let v = new [Rv];
      use v ∉ f.Rv;
      v.value ← r.value;

      use invariant f;

      use ∀x: int. ∀y: [_].
        model [f.Rv, f.Rh] (f.hash, x) = some (y) ⇒ y ∈ f.Rv;

      adopt Rv as f.Rv;
      let ignore = insert [f.Rv, f.Rt, Rk] (f.table, k, v);
    } else {
      let i = getSome (ro);
      r.value ← i.value;
    };
  };

  return r.value
}
```

Variable r stores the result of the computation. It is required because in Capucine, the **return**

statement must syntactically be at the end of the function. To prove this function, we need the invariant of f, which we obtain using the **use invariant** statement. We also need to know that v is fresh in $f.Rv$, which we obtain from typing using another **use** statement. Later, v is adopted in $f.Rv$ thanks to the inference mechanism. We also need to know that all pointers y in the table are in region $f.Rv$. We obtain this, again, from typing using a **use** statement. We use this to prove that all pointers in the table are different than v.

We solved the two problems that we mentionned at the beginning. The region of the keys is owned by the table. This forbids the client of the table from modifying them, and solves our first problem. We used parametric ownership to parameterize the table by the region of values, to ensure the table could reference these values without owning them. This solves our second issue.

3.4.5 Courses and Students

We now introduce the *courses and students* example. It consists in two classes: *Student* and *Course*. A course maintains an array of students. Each student has a mark, and the course maintains the sum of the marks of its students, as well as the student count, to be able to compute the mean easily. Each course owns its students, as modifying the mark of a student would break the course invariant, which is that the maintained sum is indeed the mark sum.

This program illustrates the use, in the specification, of a logic function which is parameterized by a region. This region represents the *footprint* of the logic function as defined in literature [Reynolds02, Kassios06, Banerjee08].

Preliminaries Class *Student* is introduced as follows:

class *Student*
{
  mark: *int*;
}

In a more practical instance of this example, one could add the name of the students and other information as other fields.

We reuse the *larray* logic type of the sparse array example. We rename them *array*, and we keep the *store* and *select* logic functions as well as the axioms which define them. We will use this array theory to model the internal student array of courses.

Mark Sum We introduce logic function *MarkSum*:

logic *MarkSum* [ρ: *Student*] (*a*: *array* ([ρ]), *i*: *int*, *j*: *int*): *int*

Intuitively, $MarkSum[\rho](a, i, j)$ is the sum of the marks of all students in array a from index i (included) to index j (excluded). Region ρ is the region in which information about students, i.e. their mark, is read. Indeed, pointer a is only a pointer: it makes no sense to read its value unless some heap in which to read the value is available. And in Capucine, heaps are modeled as regions.

Logic function *MarkSum* is defined using axioms. The first one states that the sum from i to j, when $i = j$, is 0 as j is excluded:

axiom *MSempty*:
  ∀**region** ρ: *Student*.

$\forall a$: *array* ([ρ]).
$\forall i$: *int*.
MarkSum [ρ] (a, i, i) = 0

Another axiom states that the sum from index i (included) to i+1 (excluded) is exactly the mark at index i:

axiom *MSsingleton*:
$\forall$**region** ρ: *Student*.
$\forall a$: *array* ([ρ]).
$\forall i$: *int*.
MarkSum [ρ] (a, i, i+1) = *select*(a, i).*mark*

The last axiom defining how to compute *MarkSum* is the following:

axiom *MSsplit*:
$\forall$**region** ρ: *Student*.
$\forall a$: *array* ([ρ]).
$\forall i$: *int*.
$\forall j$: *int*.
$\forall k$: *int*.
$i \leq k \wedge k \leq j \Rightarrow$
MarkSum [ρ] (a, i, j) = *MarkSum* [ρ] (a, i, k) + *MarkSum* [ρ] (a, k, j)

It states that the sum from i (included) to j (excluded) is the sum from i (included) to k (excluded) plus the sum from k (included) to j (excluded), if $i \leq k \leq j$.

Mark Sum Footprint The above axioms fully defined how to compute *MarkSum*. However, in the rest of the program we will need another axiom stating that the sum of marks from i to j only depends on the marks from i to j. The set of locations on which a function depends on is called its *footprint*.

axiom *MSfootprint*:
$\forall$**region** *Rold*: *Student*.
$\forall$**region** *Rnew*: *Student*.
$\forall$*Aold*: *array* ([*Rold*]).
$\forall$*Anew*: *array* ([*Rnew*]).
$\forall$*left*: *int*.
$\forall$*right*: *int*.
($\forall i$: *int*. *left* $\leq i \wedge i <$ *right* $\Rightarrow$
select(*Aold*, i).*mark* =
select(*Anew*, i).*mark*) $\Rightarrow$
MarkSum [*Rold*] (*Aold*, *left*, *right*) = *MarkSum* [*Rnew*] (*Anew*, *left*, *right*)

This axiom could be deduced from the above axioms *MSempty*, *MSsingleton* and *MSsplit*. But it requires reasoning which cannot be achieved by automatic theorem provers, namely induction.

Courses Class *Course* is defined as follows:

class *Course*
{

```
  group Rstudents: Student;
  students: array ([Rstudents]);
  sum: int;
  count: int;
  invariant
    sum = MarkSum [Rstudents] (students, 0, count) ∧
    count ≥ 0 ∧
    (∀i: int. ∀j: int.
     0 ≤ i ∧ i < count ⇒
     0 ≤ j ∧ j < count ⇒
     i ≠ j ⇒
     select(students, i) ≠ select(students, j)) ∧
    (∀p: [Rstudents]. p in Rstudents ⇒
     ∃i: int. 0 ≤ i ∧ i < count ∧ p = select(students, i))
}
```

The course owns its students. Indeed, the *students* field is an array of pointers of region *Rstudents* which is owned by class *Course*. This region is a group region: it may contain several students.

The *sum* field is the sum of the marks of all students in the *students* array. The *count* field is the number of students in the course, i.e. the students are stored from index 0 (included) to index *count* (excluded) of array *students*. Those properties about *sum* and *count* are specified in the invariant of class *Course*, in the first two parts.

The third part of the invariant states that all pointers of the *students* array are different. This is the kind of property which is easy to forget, as it is so obviously assumed by the programmer. It will be fundamental to the correctness of our functions.

The fourth and last part of the invariant states that all pointers of region *Rstudents* actually appear in the *students* array. We introduce a new predicate: p **in** ρ, which states that pointer p is in region ρ. We will later see that this predicate can be derived from the p: $[\rho]$ typing information in programs. Note that in the logic, a pointer may have type $[\rho]$ without the pointer actually being in ρ. More discussion about this and the actual meaning of type $[\rho]$ in the logic is made in Chapter 4.

Course Creation We introduce logic constant *someArray* of type *array* (α):

```
logic someArray (): array (α)
```

It has no definition, it is only *some array*. If logic arrays were bounded, it could be the empty array, for instance; but our logic arrays are unbounded. This is not important: *someArray* will only be used as the initial value for the *students* field of *Course* objects. We only read students from index 0 (included) to *count* (excluded), and *count* is initially 0.

Here is the implementation of the *Course* constructor:

```
fun createCourse [ρ: Course] (): [ρ]
  consumes ρ^∅
  produces ρ^×
  post result.count = 0
{
  let c = new Course [ρ];
  c.students ← someArray ();
```

```
  c.sum ← 0;
  c.count ← 0;
  return c
}
```

As usual with constructors, there are two proof obligations: the post-condition and the invariant. They both are proved automatically by theorem provers.

Adding a Student to a Course Function *addStudent* allocates a new student, given its initial mark m, and adds it to a course c. It returns the new student, which is in the owned region *Rstudents* of course c.

```
fun addStudent [Rc: Course] (c: [Rc], m: int): [c.Rstudents]
  consumes Rc^×
  produces Rc^×
  post
    result.mark = m ∧
    c.count = c.count@pre + 1 ∧
    c.sum = c.sum@pre + m ∧

    ∀s: [c.Rstudents]. s in c.Rstudents@pre ⇒
    s.mark = s.mark@pre
{
  use invariant c;
  let region Rstud: Student;
  let s = new Student [Rstud];
  s.mark ← m;

  use
    ∀i: int. i ≥ 0 ∧ i < c.count ⇒
    select(c.students, i) ≠ s;

  adopt s: Rstud as c.Rstudents;
  c.students ← store (c.students, c.count, s);
  c.sum ← c.sum + m;
  c.count ← c.count + 1;

  return s
}
```

The post-condition states that the mark of the new student is m and that the *count* and *sum* fields of c have been updated accordingly. It also states that the marks of all students which were in region *c.Rstudents* before are unchanged. It does not have to state that the new student is in region *c.Rstudents* using the **in** predicate, as this is already known thanks to typing.

We allocate the new student in a new region *Rstud* and update its mark. We then adopt the student in region *c.Rstudents*, i.e. we move the student from its current region *Rstud* to region *c.Rstudents*. After this statement, variable s has type [*c.Rstudents*]. We then update the *students* array as well as the *sum* and *count* fields.

The invariant of c holds at the beginning thanks to permission $Rc^{\times}$, and it will be used to prove proof obligations, so we introduce it in the hypotheses using statement **use invariant** c. To prove the invariant is maintained, we will also need to know that pointer s is different than all pointers which already were in array $c.students$. This can be deduced from typing, as s is allocated in region $Rstud$, which is disjoint from region $c.Rstudents$ before the adoption. We know it is disjoint because at that point, permissions $Rstud^{\circ}$ and $Rc^{\times}$ are available. If $Rc^{\times}$ is available, then $c.Rstudents^{G}$ is also morally available. It is a property of the type system that this implies disjointness of $Rstud$ and $c.Rstudents$. So we introduce the hypothesis we need using the **use** statement. The **use** statement takes a predicate, checks that it can be deduced from typing, and introduces it in the hypotheses.

There are two proof obligations: the invariant must be maintained, and the post-condition must hold. Both are proved automatically using theorem provers, thanks to the hypotheses we introduced using the **use invariant** and **use** statements.

Changing the Mark of a Student Function *changeMark* takes a course c, a student s of this course, a new mark m and changes the mark of s to m. Of course, it also updates the *sum* field of c.

fun *changeMark* [Rc: *Course*] (c: [Rc], s: [$c.Rstudents$], m: *int*): *unit*
consumes $Rc^{\times}$
produces $Rc^{\times}$
post
$s.mark = m \wedge$
$c.sum = c.sum@\textbf{pre} + m - s.mark@\textbf{pre} \wedge$
$c.count = c.count@\textbf{pre} \wedge$

$\forall i$: *int*.
$0 \leq i \wedge i < c.count \Rightarrow$
$s \neq select(c.students, i) \Rightarrow$
$select(c.students, i).mark = select(c.students, i).mark@\textbf{pre}$
{
use invariant c;
use $s \in c.Rstudents$;

label *before*;
let region F: *Student*;
focus s **as** F;
$c.sum \leftarrow c.sum - s.mark + m$;
$s.mark \leftarrow m$;
unfocus s: F **as** $c.Rstudents$;

assert
$\exists i$: *int*.
$0 \leq i \wedge i < c.count \wedge$
$select(c.students, i) = s \wedge$
$MarkSum\ [c.Rstudents]\ (c.students, 0, i) =$
$MarkSum\ [c.Rstudents]\ (c.students, 0, i)@before \wedge$
$MarkSum\ [c.Rstudents]\ (c.students, i{+}1, c.count) =$

MarkSum [*c.Rstudents*] (*c.students*, *i*+1, *c.count*)@*before*;
}

The post-condition states that the mark of student s is now m, the *sum* field of course c has been updated, and the *count* field is unchanged. It also states that the marks of all students except s are unchanged.

The code is mainly composed of the two update statements to *c.sum* and *s.mark* respectively. All other statements deal with regions or the program specification. Statements dealing with regions are **let region**, **focus** and **unfocus**, as the student s being in a group region, it must be moved temporarily into a singleton region so we can modify its mark. Statement **use invariant** c introduces the invariant of c in proof obligation hypotheses for free, as permission $Rc^{\times}$ is available. Statement **use** s **in** *c.Rstudents* introduces the s **in** *c.Rstudents* hypothesis for free, as it is known by typing.

The assertion helps automatic provers to prove proof obligations. It essentially shows them how to split the *students* array in three parts: the students before s, s, and the students after s. Using this splitting, it states that all students but s are unchanged from the *before* label, which was introduced before the assign statements. Thanks to this assertion, both proof obligations, i.e. the invariant and the post-condition, are proved automatically.

Conclusion This example shows the limits of the Capucine approach: when pointers have to belong to group regions, we have to resort to specifying separation in the logic. However, Capucine still helps by separating the two courses in the main function, by providing a methodology to handle the invariant of the *Course* class and by separating fields in proof obligations. This example also illustrates the **in** predicate and how it can be introduced in the proof obligation hypotheses if it can be shown through typing. It also shows the **use** statement which can introduce hypotheses which can be derived from typing, in particular pointer differences when pointers belong two disjoint regions.

Chapter 4

Capucine Language Syntax and Semantics

In this chapter we formalize the Capucine language. In Section 4.1 we give the grammar of Capucine programs. In Section 4.2 we give typing rules for Capucine programs. In Section 4.3 we give the operational semantics of Capucine. We call it the *intuitive* model, as it is close to usual operational semantics of other imperative languages with pointers. In Section 4.4 we introduce the notion of *coherence* of a heap. Our first main result (Theorem 1) is that coherence is preserved through execution. In Section 4.5 we give another operation semantics for Capucine. We call it the *separated* model, as it emphasizes separating the heap into regions. Our second main result (Theorem 5) is that under the coherence hypothesis, execution in the separated model gives the same result as execution in the intuitive model.

4.1 Syntax

In this section we formally introduce the Capucine language. We define its syntax and remind the informal meaning of each operation. Capucine programs are sequence of declarations: logic declarations (logic types, logic functions, predicates and axioms) and program declarations (classes and functions). These notions are mutually dependent. For instance, classes appear in predicate declarations, but predicates appear in class declarations. Thus, this section contains unavoidable forward references.

We denote by $\overline{x}$ the list $x_1, ..., x_n$ for some n. In a similar fashion, $\overline{x : y}$ stands for $x_1 : y_1, ..., x_n : y_n$ for some n, and so on.

4.1.1 Classes

We introduce some namespaces for identifiers: r and s are *region names*, where s will be preferably used for *owned* regions; and f are *field names*. A class declaration is of the form:

class C $(\overline{\alpha})$ $[\overline{r\colon \mathcal{C}}]$
{
$\overline{\mathit{size}\ s\colon \mathcal{C};}$
$\overline{f\colon \tau;}$

invariant P;
}

where *size* may be either **single** or **group**, and $\mathcal{C}$ are *class expressions*:

$\mathcal{C} ::= C\ [\overline{\rho}]\ (\overline{\tau})$

If class C expects no region ρ, we may omit the brackets, and if class C expects no type τ, we may omit the parentheses. For instance, C stands for C [] (). Regions ρ, types τ and predicates P will be defined later in this section.

Type variables α are *type parameters*, and region variables r are *region parameters*. They are bound in the remaining of the class definition. In particular, they can be used as parameters of later class expressions. For instance, this is a valid class definition:

class $C\ (\alpha)\ [r_1\colon C_1\ (\alpha),\ r_2\colon C_2\ [r_1]\ ]\ \{\ \cdots\ \}$

Type parameters are ML-like type parameters. Region parameters are similar, but they are typed using class expressions.

Classes may also *own regions*. Regions s in the above declaration are owned regions. Each region can be either group (**group**) or singleton (**single**). If a region is group, it may contain any number of pointers, including zero or one; if a region is singleton, it may only contain exactly one pointer. Owned regions, like region parameters, are typed using class expressions. They are bound in the remainder of the class declaration, including in the class expressions of following owned regions.

Classes contain *fields*. Each field f is given a type τ. This is similar to structures and records of other languages.

Finally, each class is given an *invariant*. The invariant may mention each field f, as well as owned regions s. There is an important restriction though, which is the usual restriction for invariants in ownership methodologies: the invariant may only dereference owned pointers, i.e. if a term $x.f_1.\ \cdots\ .f_i$ appears in the invariant, then $x.f_1$ is a pointer of an owned region of x, $x.f_1.f_2$ is a pointer of an owned region of $x.f_1$, and so on.

4.1.2 Logic Types

A logic type definition is of the form:

type $t\ (\alpha_1, \cdots, \alpha_n)$

It declares type t, with polymorphic type parameters $\alpha_1, \cdots, \alpha_n$ where n may be zero. It is only a *declaration*: there is no way to build a value of type t without defining its *constructors* as logic functions.

We assume all types to be *inhabited*: for each user-declared type t, there is at least one value of type t. This implies that, for instance, logic formula:

$$\forall x\colon t.\ \mathsf{F}$$

is equivalent to F, where F is the false predicate. If there was no value of type t, it would instead be equivalent to T, the true predicate. In a sense, a type declaration is an axiom stating the existence of a value. This approach is similar to Why [Filliâtre07], in which Capucine programs are interpreted. If the user wishes to, he may build a model of the type in another, richer language such as the one of the Coq proof assistant [Coq].

4.1.3 Regions

A region is of the form:

$\rho ::= r \mid x.s$

It is either a region name r, or an *owned* region. Owned regions are of the form $x.s$, where x is a variable of a class which contains owned region name s in its declaration. Region $x.s$ refers to the owned region s of the object which variable x points to.

4.1.4 Types

An *object* in Capucine is a value of a class. Objects do not appear in programs though. The programmer may only manipulate *pointers* to objects. A variable of a pointer type is a variable whose contents is an address in the heap, at which address an object can be found. Capucine splits the heap into several smaller heaps: one for each region. Regions are not a partition of the heap though, they may overlap. Each pointer belongs to at least one region.

The region of each *program pointer* is denoted by its type. The type of a pointer p of region ρ is written $[\rho]$; pronounced "at ρ". Each region binder comes with a class expression $\mathcal{C}$, so the class of ρ, i.e. the class of pointer p, is known from the typing environment.

In the logic though, pointers do not necessarily belong to a region. To read a pointer, i.e. to select a field of its object, it is necessary to know in which region the pointer is read. But in the logic it sometimes makes sense to read the same pointer in different regions. For instance, two different logic regions may denote different versions of the same region throughout the execution of the program. It then makes sense that a pointer is in both regions at the same time. The type given to pointers in the logic is [_], i.e. the type of pointers of *any* region. Field selection is then annotated with the region in which the reading access is made, using term *get*.

Additionnally, to avoid having to write $get(\rho, x, f)$, we actually allow logic pointers such as x to carry a region ρ in their type. Their type is thus $[\rho]$ instead of [_], but this is only used to desugar $x.f$ as $get(\rho, x, f)$.

A type is of the form:

$\tau ::= t\ (\tau, \cdots, \tau) \mid int \mid bool \mid unit \mid [\rho] \mid [\_]$

It is either a user-defined type t with instanciated parameters $\tau_1, \cdots, \tau_n$, a base type, or a pointer type. Base types are integers *int*, booleans *bool* and *unit*.

4.1.5 Logic Function and Predicate Declarations

A predicate definition is of the form:

logic $p\ \overline{param} = P$

where:

$param ::= [r\colon \mathcal{C}] \mid (x\colon \tau)$

We define predicates P later in this section. This defines predicate p, with arguments $\overline{arg}$. An argument is either a region argument $[r\colon \mathcal{C}]$ where r is a region name and $\mathcal{C}$ is its class, or a regular argument $(x\colon \tau)$ where x is a variable name and τ is its type. Note that regions r may appear in types τ of variables, and that variables x may appear in classes $\mathcal{C}$ of regions. Here is an example of such an interleaving:

logic p $[r\colon C_1]$ $(x\colon [r])$ $[r'\colon C_2\ [x.r_1]\ ]$ $(y\colon [r'])$

assuming r_1 is an owned region of class C_1, and class C_2 expects one region parameter.

A logic function definition is of the form:

logic $f\ \overline{param}\colon \tau = LT$

We define terms LT later in this section. As usual, the only difference between a predicate and a logic function is their return type, which are of different kinds. Logic functions may return values of any user-defined type. The body of a logic function is not a predicate but a term.

The body of predicates and logic functions, respectively P and LT above, is optional. Often, the predicate and term language cannot express them. In this case, the user provides axioms instead. If the user defined a model for the logic types in a richer logic, he may also define logic functions and predicates in this richer logic and prove the axioms to ensure consistency.

4.1.6 Terms

A term is of the form:

$$
\begin{array}{rcl}
LT & ::= & c \\
& | & x \\
& | & LT\ termop\ LT \\
& | & f\ \overline{larg} \\
& | & get(RT,\ LT,\ f)
\end{array}
$$

$termop ::= + \mid - \mid \times \mid \&\& \mid \mid\mid$
$larg ::= [RT] \mid (LT)$

where c is a constant of a base type, i.e. either an integer literal or a boolean *true* or *false*. A term may also be a variable x, the application of a base term operator *termop* or the application of a logic function f to some logic arguments $\overline{larg}$. A logic argument *larg* is either $[RT]$ where RT is a region term (see below) or (LT) where LT is a term.

We use a set of *label names* L. Labels denote the state of the heap at some program point. A special label named **here** denotes the current program point.

Finally, a term can be the selection $get(RT,\ LT,\ f)$ of a field f of a pointer term LT in a region RT, where RT is a *region term*:

$$RT ::= r@L \mid get(RT,\ LT,\ s)$$

It is either a region variable r at some label L or the selection $get(RT,\ LT,\ s)$ of an owned region s of a pointer term LT in a region RT. Selection of owned regions and fields are quite similar.

In practice, we may also write r with no label at all. This is desugared as $r@L$ where L is chosen depending on the context. We may write $RT@L$, $LT@L$ or $P@L$ for any region term RT, term LT or predicate P. Region term r is then expanded as $r@L$, where L is the innermost label. For instance, $((f\ [r])@L)@L'$ is desugared as $f\ [r@L]$ and not $f\ [r@L']$. If no label is provided at all, r is replaced by $r@$**here**.

Abbreviation In practice, $get(\rho, p, f)$ can often be abbreviated as $p.f$. Indeed, if the type of p is $[r]$, then we expand $p.f$ to $get(r, p, f)$. If the type of p is $[x.r]$, then we expand $p.f$ to $get(t, p, f)$ where t is the expansion of $x.r$, computed in a similar fashion. If the type of p is $[\_]$ however, $p.f$ has no meaning and cannot be expanded. It is rejected by Capucine. It is thus mandatory to use the *get* predicate for pointers of type $[\_]$. In examples of Section 3.4 we used this abbreviation when possible. Function *addStudent* of the *courses and students* example shows a use of type $[\_]$.

4.1.7 Predicates

A predicate is of the form:

$$\begin{array}{rcl} P & ::= & p\ \overline{larg} \\ & | & P \wedge P \mid P \vee P \mid P \Rightarrow P \mid P \iff P \mid \neg P \mid \mathsf{T} \mid \mathsf{F} \\ & | & LT = LT \mid LT \neq LT \mid LT > LT \mid LT \geq LT \\ & | & \forall x\colon \tau.\ P \mid \exists x\colon \tau.\ P \\ & | & \forall \textbf{region}\ r\colon \mathcal{C}.\ P \mid \exists \textbf{region}\ r\colon \mathcal{C}.\ P \\ & | & LT \in RT \end{array}$$

It can be the application of a user-defined predicate p or the quantification of a variable x. Usual operators are also provided, including comparison, as well as constant predicates T and F. A predicate can also be a quantification over a region r of class $\mathcal{C}$. Region r can then be used in $get(r, LT, f)$ terms, and in types. Finally, predicate $LT \in RT$ states that the location denoted by term LT is in the region denoted by region term RT.

4.1.8 Axioms and Lemmas

Axiom declarations are of the form:

axiom a: P

It is essentially a pair of a name a and a predicate P. The axiom does not take any parameters; variables and regions must be quantified in the body P. Types appearing in P may use free type variables, which are implicitely universally quantified at the beginning of the axiom, i.e. the axiom is polymorphic in these variables. Axioms are typically used to axiomatize predicate and logic functions which cannot be defined otherwise in the first-order logic of Capucine.

Lemmas are similar to axioms, and are declared as:

lemma l: P

They are available in the context of proof obligations, but contrary to axioms, they must be proved.

4.1.9 Permissions

Permissions are of the form:

$\Sigma ::= \rho^{\emptyset} \mid \rho^{\circ}\{\overline{f}\} \mid \rho^{\times} \mid \rho^{G} \mid \rho \multimap \rho$

Permission $\rho^{\emptyset}$ states that region ρ is empty, i.e. contains no pointer. Permission $\rho^{\circ}\{\overline{f}\}$ states that region ρ is singleton, i.e. contains exactly one pointer p, and that ρ is open, i.e. the invariant p may be broken. It also states that fields $\overline{f}$ are not initialized. Permission $\rho^{\times}$

states that region ρ is singleton and closed, i.e. the invariant of the pointer holds. It also encapsulates permissions on owned regions. Permission ρ^G states that region ρ is group, i.e. it may contain any number of closed pointers. Permission $\sigma \multimap \rho$ states that region ρ is being focused, and is thus disabled as long as $\sigma^\times$ is not given back.

Notation We may use ρ° to denote permission $\rho^\circ\{\}$, i.e. ρ is open and all its fields are initialized.

4.1.10 Expressions

All expressions are pure, i.e. do not cause side-effects. They are of the form:

e	::=	$0, 1, \cdots$	Integers
	|	*true*, *false*	Booleans
	|	$()$	Unique value of the *unit* type
	|	$e\ op\ e$	Operations ($op \in \{+, -, \times, =, \neq, >, <, \geq, \leq\}$)
	|	$f\ \overline{arg}$	Logic function application
	|	x	Variable
	|	$x.f$	Field selection

where:

$arg ::= [\rho] \mid (e)$

We restrict field selection to variables. This is to ensure a type can be given to $x.f$ when f is a field of pointer type of an owned region r. Indeed, the type of $x.f$ is then $[x.r]$: name x is needed. More complex expressions such as $x.f_1.f_2$ can be encoded by giving a name to $x.f_1$ such as y using a let-binding statement.

Logic functions may be used as expressions. They take a list of arguments $\overline{arg}$ where arg is either a region argument $[\rho]$ or a regular expression argument (e).

4.1.11 Statements

Statements do cause side-effects and are of the form:

s	::=	**let** $x = e$	Variable declaration
	|	**if** e **then** S **else** S	Test
	|	**let** $x = f\ \overline{arg}$	Call
	|	**let region** r: $\mathcal{C}$	New region
	|	**let** $x =$ **new** $\mathcal{C}\ [\rho]$	Allocation
	|	**focus** x: ρ **as** ρ	Extract a pointer
	|	$x.f \leftarrow e$	Assign field
	|	**adopt** x: ρ **as** ρ	Move a pointer
	|	**unfocus** x: ρ **as** ρ	Opposite of **focus**
	|	**pack** x: $\mathcal{C}$	Close pointer
	|	**unpack** x	Open pointer
	|	**weaken empty** ρ	Weaken empty region
	|	**weaken single** ρ	Weaken singleton region
	|	**assert** P	Assertion
	|	**label** L	Label

where S are *blocks*, i.e. sequences of statements:

$S ::= \{s; \cdots; s\}$

The scope of all binders is restricted to the current block.

Function calls may be recursive. Loops are encoded using recursive calls. We chose not to have loop statements in order to simplify the presentation of the language. It allows to avoid duplicating the ideas behind typing rules of recursive calls. In particular, loops would require loop invariant annotations, including *invariant permissions*: available permissions should be the same at the beginning and at the end of the body of loops.

The **let region** r: $\mathcal{C}$ operation binds a new region r, of class expression $\mathcal{C}$. The region is initially empty. It can be used to allocate or focus pointers.

Allocation **new** takes a region ρ as argument and allocates a new object of the class of ρ inside this region. A pointer to this object is stored in the let-bound variable x.

The **focus** x: ρ **as** σ operation extracts pointer x from group region ρ to empty region σ, which becomes singleton. The type of x changes from $[\rho]$ to $[\sigma]$. The **unfocus** operation does the opposite. While the pointer is focused, region ρ is temporarily disabled: permission ρ^G is replaced by $\sigma \multimap \rho$. Adoption is similar to unfocus in that it moves a variable from a singleton region to a group region, except that the variable was not being focused.

Packing and unpacking control the state of pointers (open or closed). Packing generates a proof obligation: the invariant of the pointer being packed must hold.

Weakening is strictly a typing operation. It takes an empty or closed singleton region, consumes the permission $\rho^\emptyset$ or $\rho^\times$ and produces the group permission ρ^G.

Assertion **assert** P checks that predicate P holds at the current program point. It generates P as a proof obligation.

Label declaration **label** L binds label name L in the following statements of the current block. The label L then denotes the state of the program at its declaration point.

4.1.12 Function Declarations

A function declaration is of the form:

fun f $\overline{param}$: τ
  consumes $\overline{\Sigma}$
  produces $\overline{\Sigma}$
  pre P
  post P
{
  s;
  $\cdots$
  s;
  return e
}

The meaning of return type τ, pre- and post-conditions P, body $\overline{s}$ and returned expression e is straightforward.

Arguments *param* have been defined in Section 4.1.5. Region arguments are region binders. Regions bound this way are available in the remainder of the function declaration. Typically, they are regions of pointer arguments, or of the return value if it is a pointer. As usual with region binders, each region parameter is given a class expression $\mathcal{C}$.

The **consumes** and **produces** clauses state which permissions $\overline{\Sigma}$ are consumed and produced by the function. They give the state of region parameters before and after the function

is called.

4.2 Typing

Now that we laid down the syntax of Capucine programs and logic, we introduce typing rules. There are three main mechanisms involved: typing of expressions, typing of logic terms, and permissions. Typing of expressions and logic terms use rules similar to type systems of the ML family. In particular, they involve type variables. Capucine also uses region variables, which are similar to type variables in their behavior. For instance, classes may have type and region parameters. Typing of permissions reflects how permissions are consumed and produced by statements. Permissions are also used in typing rules of expressions to check that reading access are allowed. They are not used in typing rules of terms, because our model for logic is different than the one used for programs and allows reading even with no permission.

We assume all identifiers are α-renamed when needed. For instance, we assume programs such as:

let $x = 1$;
let $x = \mathit{true}$;

do not occur. The implementation of Capucine actually transforms this program into:

let $x_1 = 1$;
let $x_2 = \mathit{true}$;

4.2.1 Expressions

Owned Regions and Field First we introduce typing judgements for owned regions and fields:

$$s\colon \mathcal{C}' \in_x \mathcal{C}$$

$$f\colon \tau \in_x \mathcal{C}$$

The first judgement states that owned region name s, defined in class $\mathcal{C}$, has class $\mathcal{C}'$ when owned by pointer variable x. The second judgement is the equivalent for fields: it states that $x.f$ has type τ, where f is defined in class $\mathcal{C}$.

These judgements involve substitution of type and region variables. Consider for instance the following class declaration:

class C (α) $[r\colon \mathit{Long}]$
{
 single $s\colon \mathit{List}$ (α) $[r]$;
 $f\colon \mathit{Array}$ $([s])$;
}

Consider the following environment which gives the type of x:

$$\sigma\colon \mathit{Long},\ \rho\colon C\ (\mathit{int})\ [\sigma],\ x\colon [\rho]$$

The following judgement holds:

$$\frac{}{\Gamma,\ r:\ \mathcal{C} \vdash r:\ \mathcal{C}}\ \textsc{RRoot} \qquad \frac{\Gamma,\ x:\ [\rho] \vdash \rho:\ \mathcal{C} \qquad s:\ \mathcal{C}' \in_x \mathcal{C}}{\Gamma,\ x:\ [\rho] \vdash x.s:\ \mathcal{C}'}\ \textsc{ROwn}$$

Figure 4.1: Typing rules for regions

$$f\text{: } \mathit{Array}([x.s]) \in_x C\ (\mathit{int})\ (\sigma)$$

Note how *Array* ([s]) was substituted to *Array* ([$x.s$]). This illustrates the necessity of x in the $\in_x$ judgement.

This substitution mechanism is the usual ML mechanism for type parameters, on top of which we add substitution of owned regions. Substitution for type and region parameters use the ML mechanism, which is well-known and we do not detail it. Substitution for owned regions is quite simple: all region variables s which are owned regions of class $\mathcal{C}$ are substituted to $x.s$, where x is the variable by which $\in_x$ is indexed.

Environments Typing rules for expressions involve an environment Γ. It is a list of:

- x: τ pairs, where x is a variable and τ is its type;
- r: $\mathcal{C}$ pairs, where r is a region variable and $\mathcal{C}$ is its class expression;
- L: R pairs, where L is a label and R is a set of region names.

The order of the list does not matter, and we will use Γ, a: b to denote an environment containing pair a: b as well as some other pairs Γ. For clarity's sake, you may assume Γ to be a function, although it is not necessary. For instance, environment x: *int*, x: *bool* will never occur naturally.

Regions Regions are typed using the following judgement:

$$\Gamma \vdash \rho\text{: } \mathcal{C}$$

This judgement states that under environment Γ, region ρ contains pointers of class expression $\mathcal{C}$. Figure 4.1 shows typing rules for regions.

Rule RRoot states that if r: $\mathcal{C}$ is in the environment, then r contains pointers of class $\mathcal{C}$.

Rule ROwn states how to type owned region $x.s$. First we look for the type of x in the environment. It should be a pointer type $[\rho]$, where ρ is the region of x. We then type ρ in the same environment, and obtain its class $\mathcal{C}$. We finally look for the type of region s in class expression $\mathcal{C}$ when it is owned by pointer x, and we obtain the class $\mathcal{C}'$ of $x.s$.

Accessible Regions We introduce the following judgements aimed at defining whether there are enough permissions for a region ρ to be *accessible*. Accessibility of a region denotes whether we are able to translate an access to this region to compute proof obligations. To this end we require permissions on the region and its owners. In other words we need to know the state of the region, in particular whether it is being focused.

- $\Gamma, \overline{\Sigma} \vdash_\circ \rho$

  This judgement states that region ρ is accessible and is in the *open* part of the region ownership tree, i.e. that ρ and all its transitive owners are open.

$$\frac{r^{\circ}\{f_1,\ \cdots,\ f_k\} \in \overline{\Sigma}\ (k\ \geq\ 0)}{\Gamma,\ \overline{\Sigma} \vdash_{\circ} r}\ \textsc{Acc}1$$

$$\frac{x.s^{\circ}\{f_1,\ \cdots,\ f_k\} \in \overline{\Sigma}\ (k\ \geq\ 0) \qquad x:\ [\rho] \in \Gamma \qquad \Gamma,\ \overline{\Sigma} \vdash_{\circ} \rho}{\Gamma,\ \overline{\Sigma} \vdash_{\circ} x.s}\ \textsc{Acc}2$$

$$\frac{\overline{\Sigma} \cap \{r^{\emptyset},\ r^{\times},\ r^{G}\}\ \neq\ \emptyset}{\Gamma,\ \overline{\Sigma} \vdash_{\times} r}\ \textsc{Acc}3 \qquad\qquad \frac{\sigma \multimap r \in \overline{\Sigma} \qquad \Gamma,\ \overline{\Sigma} \vdash \sigma}{\Gamma,\ \overline{\Sigma} \vdash_{\times} r}\ \textsc{Acc}4$$

$$\frac{\overline{\Sigma} \cap \{x.s^{\emptyset},\ x.s^{\times},\ x.s^{G}\}\ \neq\ \emptyset \qquad x:\ [\rho] \in \Gamma \qquad \Gamma,\ \overline{\Sigma} \vdash_{\circ} \rho}{\Gamma,\ \overline{\Sigma} \vdash_{\times} x.s}\ \textsc{Acc}5$$

$$\frac{\sigma \multimap x.s \in \overline{\Sigma} \qquad \Gamma,\ \overline{\Sigma} \vdash \sigma \qquad x:\ [\rho] \in \Gamma \qquad \Gamma,\ \overline{\Sigma} \vdash_{\circ} \rho}{\Gamma,\ \overline{\Sigma} \vdash_{\times} x.s}\ \textsc{Acc}6$$

$$\frac{x:\ [\rho] \in \Gamma \qquad \Gamma,\ \overline{\Sigma} \vdash_{\times} \rho}{\Gamma,\ \overline{\Sigma} \vdash_{\times} x.s}\ \textsc{Acc}7 \qquad \frac{\Gamma,\ \overline{\Sigma} \vdash_{\times} \rho}{\Gamma,\ \overline{\Sigma} \vdash \rho}\ \textsc{Acc}8 \qquad \frac{\Gamma,\ \overline{\Sigma} \vdash_{\circ} \rho}{\Gamma,\ \overline{\Sigma} \vdash \rho}\ \textsc{Acc}9$$

Figure 4.2: Typing rules for region accessibility

- $\Gamma, \overline{\Sigma} \vdash_{\times} \rho$

  This judgement states that region ρ is accessible and is in the *closed* part of the region tree. It can be at the *border* between the closed part and the open part, i.e. permission $\rho^{\emptyset}$, $\rho^{\times}$, ρ^{G} or $\sigma \multimap \rho$ is available. If the permission is $\sigma \multimap \rho$, then σ must be accessible. Or ρ can be below the border, i.e. transitively owned by a region of the border.

- $\Gamma, \overline{\Sigma} \vdash \rho$

  This judgement states that region ρ is accessible, whether in the open part or the closed part of the region tree.

Figure 4.2 defines these judgements.

Rules Acc1 and Acc2 define judgement $\vdash_{\circ}$. A region is accessible in the open part of the region ownership tree if the region is open and either the region is at the top of the tree, i.e. is a variable r, or the region is owned by a region which is itself accessible in the open part of the tree.

Rules Acc3 to Acc7 define judgement $\vdash_{\times}$. A region ρ on which we have a closed permission ($\rho^{\emptyset}$, $\rho^{\times}$, ρ^{G} or $\sigma \multimap \rho$ for some region σ) is at the border of the tree (rules Acc3 to Acc6). If the region is owned, its owner must be accessible in the open part of the tree (rules Acc5 and Acc6). If the region is being focused, the target region σ must be accessible (rules Acc4 and Acc6). Finally, a region which is transitively owned by a closed region is also accessible in the closed part of the tree (rule Acc7).

We cannot read from or write into a region which is not accessible. Thus most typing rules use the region accessibility judgement $\vdash$. More details on how we access regions and why this typing judgements are required are given in Section 4.3, Section 4.5, and Section 5.2.

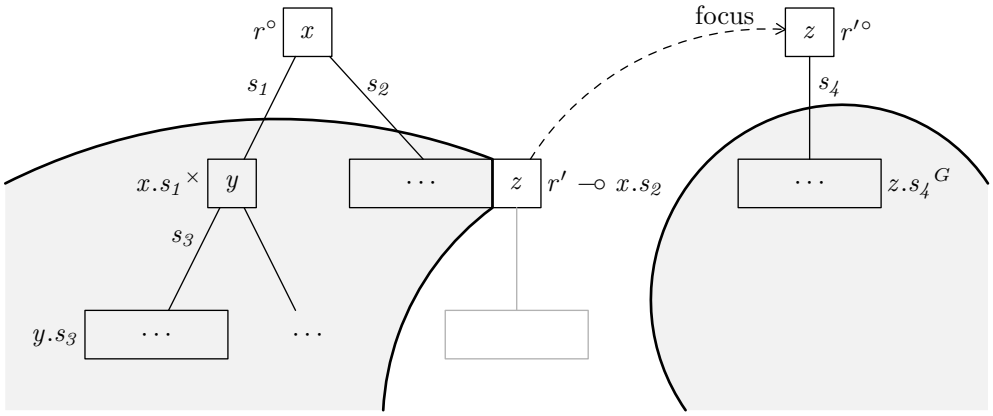

Figure 4.3: Illustration of a region ownership tree and its open/close border

$$\frac{\overline{\Sigma} \cap \{\rho^\emptyset,\ \rho^\times,\ \rho^G\} \neq \emptyset}{\Gamma,\ \overline{\Sigma} \vdash_{\multimap} \rho}$$

$$\frac{\begin{array}{ll} \rho^\circ \in \overline{\Sigma} & \Gamma \vdash \rho : \mathcal{C} \\ \text{owned regions of } \mathcal{C} \text{ are } r_1, \cdots, r_n & x : [\rho] \in \Gamma \qquad \text{for all i, } \Gamma,\ \overline{\Sigma} \vdash_{\multimap} x.r_i \end{array}}{\Gamma,\ \overline{\Sigma} \vdash_{\multimap} \rho}$$

Figure 4.4: Typing rules for focus-freeness

In fact, this typing requirement is only used when translating Capucine programs to Why programs, in Section 5.2.

Figure 4.3 illustrates an example of a region ownership tree. The gray part of the figure is the closed part of the tree. Regions r and r' form the open part of the tree. Region r is open and singleton; it contains one location x. This location owns a closed singleton region s_1. The location y of s_1 owns, among others, a group region s_3. Location x also owns a group region s_2, which contains a location z which is being focused in region r'. Location z owns a closed group region s_4. Thus the subtree rooted at z in $x.s_2$ is inactive: it has been moved into r'.

Focus-Free Regions We introduce the following judgement aimed at checking whether a region is *focus-free*:

$$\Gamma,\ \overline{\Sigma} \vdash_{\multimap} \rho$$

This states that ρ is not being focused, and that none of its (transitively) owned regions is being focused either. Figure 4.4 defines this judgement.

A simple way to prove that region ρ is focus-free is if $\rho^\emptyset$, $\rho^\times$ or ρ^G is available. Else, then ρ° is available and we have to find a variable x of type $[\rho]$ in the environment such that all owned permissions $x.r_i$ are focus-free.

$$\frac{}{\Gamma,\ \overline{\Sigma} \vdash c :\ typeof(c)}\ \textsc{EConst} \qquad \frac{}{(\Gamma,\ x :\ \tau),\ \overline{\Sigma} \vdash x :\ \tau}\ \textsc{EVar}$$

$$\frac{\Gamma,\ \overline{\Sigma} \vdash e_1 :\ int \qquad \Gamma,\ \overline{\Sigma} \vdash e_2 :\ int \qquad op \in \{+,\ -,\ \times\}}{\Gamma,\ \overline{\Sigma} \vdash e_1\ op\ e_2 :\ int}\ \textsc{EIntOp}$$

$$\frac{\Gamma,\ \overline{\Sigma} \vdash e_1 :\ bool \qquad \Gamma,\ \overline{\Sigma} \vdash e_2 :\ bool \qquad op \in \{\&\&,\ ||\}}{\Gamma,\ \overline{\Sigma} \vdash e_1\ op\ e_2 :\ bool}\ \textsc{EBoolOp}$$

$$\frac{\Gamma,\ \overline{\Sigma} \vdash e_1 :\ int \qquad \Gamma,\ \overline{\Sigma} \vdash e_2 :\ int \qquad op \in \{=,\ \neq,\ >,\ <,\ \geq,\ \leq\}}{\Gamma,\ \overline{\Sigma} \vdash e_1\ op\ e_2 :\ bool}\ \textsc{EIntComp}$$

$$\frac{\Gamma,\ \overline{\Sigma} \vdash e_1 :\ \tau \qquad \Gamma,\ \overline{\Sigma} \vdash e_2 :\ \tau \qquad op \in \{=,\ \neq\}}{\Gamma,\ \overline{\Sigma} \vdash e_1\ op\ e_2 :\ bool}\ \textsc{EComp}$$

$$\frac{\begin{array}{c}\rho^{\circ}\{\overline{f}\} \in \overline{\Sigma} \\ f\ \notin\ \overline{f} \qquad \Gamma,\ x :\ [\rho] \vdash \rho :\ \mathcal{C} \qquad (\Gamma,\ x :\ [\rho]),\ \overline{\Sigma} \vdash \rho \qquad f :\ \tau \in_x \mathcal{C}\end{array}}{(\Gamma,\ x :\ [\rho]),\ \overline{\Sigma} \vdash x.f :\ \tau}\ \textsc{ESelectOpen}$$

$$\frac{\begin{array}{c}\overline{\Sigma} \cap \{\rho^{\times},\ \rho^{G},\ \sigma \multimap \rho\}\ \neq\ \emptyset \\ \Gamma,\ x :\ [\rho] \vdash \rho :\ \mathcal{C} \qquad (\Gamma,\ x :\ [\rho]),\ \overline{\Sigma} \vdash \rho \qquad f :\ \tau \in_x \mathcal{C}\end{array}}{(\Gamma,\ x :\ [\rho]),\ \overline{\Sigma} \vdash x.f :\ \tau}\ \textsc{ESelect}$$

$$\frac{f :\ param_1\ \cdots\ param_n\ \rightarrow\ \tau \qquad \text{for all } i :\ \Gamma,\ \overline{\Sigma},\ \sigma \vdash arg_i :\ param_i}{\Gamma,\ \overline{\Sigma} \vdash f\ arg_1\ \cdots\ arg_n :\ \sigma(\tau)}\ \textsc{EApp}$$

Figure 4.5: Typing rules for expressions

This judgement is used when a region is passed from programs to logic. Indeed, we cannot read focused regions in the logic, nor regions which transitively own a focused region.

Expressions Expressions are typed using the following judgement:

$$\Gamma, \overline{\Sigma} \vdash e\colon \tau$$

This judgement states that under environment Γ and if permissions $\overline{\Sigma}$ are available, where $\overline{\Sigma}$ is a multiset, expression e has type τ. Note that although $\overline{\Sigma}$ is a multiset, we show in Section 4.4 that if we type a program which starts with no permission, it is not possible for the same permission to appear twice later on. Figure 4.5 shows typing rules for expressions.

Rule EConst states that the type of a constant c is $typeof(c)$. If c is an integer, $typeof(c)$ is int; if c is a boolean, $typeof(c)$ is $bool$; if c is the unit constant (), then $typeof(c)$ is $unit$.

Rules EIntOp, EBoolOp EIntComp and EComp give the type of operators. Note that only integers may be compared using $>$, $<$, $\geq$ or $\leq$.

Rule EVar states that if $x\colon \tau$ is in the environment, then variable x has type τ.

$$\frac{\Gamma \vdash \rho : \sigma(\mathcal{C}) \qquad \Gamma,\ \overline{\Sigma} \vdash \rho}{\Gamma,\ \overline{\Sigma},\ \sigma \vdash [\rho] :\ [r :\ \mathcal{C}]}\ \textsc{ARegion} \qquad \frac{\sigma(x) \text{ is undefined} \qquad \Gamma,\ \overline{\Sigma} \vdash e :\ \sigma(\tau)}{\Gamma,\ \overline{\Sigma},\ \sigma \vdash (e) :\ (x :\ \tau)}\ \textsc{ARegular}$$

$$\frac{\sigma(x) \ =\ y \qquad \Gamma,\ \overline{\Sigma} \vdash y :\ \sigma(\tau)}{\Gamma,\ \overline{\Sigma},\ \sigma \vdash (y) :\ (x :\ \tau),\ \sigma}\ \textsc{AVariable}$$

Figure 4.6: Typing rules for arguments

Rules ESelectOpen and ESelect states how to type field selection $x.f$. First we look for the type of x in the environment. It should be a pointer type $[\rho]$, where ρ is the region of x. We then check that there is a permission available for region ρ, be it $\rho^{\circ}\{\overline{f}\}$ where f is initialized, i.e. is not in $\overline{f}$ (rule ESelectOpen) or $\rho^{\times}$, ρ^{G} or $\sigma \multimap \rho$ (for any region σ — rule ESelect). We then type ρ in the same environment, and obtain its class $\mathcal{C}$. We finally look for the type of field f in class expression $\mathcal{C}$, and we obtain the type τ of $x.f$. As you can see, this rule is similar to rule ROwn, with an additional permission check.

Rule EApp states how to type logic function application. It uses the following new judgement:

$$f\colon \mathit{param}_1 \cdots \mathit{param}_n \to \tau$$

which states that f has been declared as a function taking n region or regular arguments $\mathit{param}_1 \cdots \mathit{param}_n$ and returning a value of type τ. It also uses the following new judgement:

$$\Gamma,\ \overline{\Sigma},\ \sigma \vdash \mathit{arg}\colon \mathit{param}$$

which states that argument *arg* has the type of argument declaration *param*, under environment Γ, permissions $\overline{\Sigma}$, and substitution σ. This judgement is defined in Figure 4.6. To sum up how application is typed, first we find a substitution σ for type variables α, region variables r, and variables x which are defined as parameters of function f. Region variables are substituted to the regions given to the application. Variables are substituted to the variables given to the application (rule AVariable) if needed. Type variables are substituted to ensure arguments e_i are of the correct type. The type of the application is then substitution σ applied to the return type τ.

If a regular argument is a variable y, rule AVariable may be applied instead of ARegular. Rule AVariable allows to substitute y in regions. For instance, say f is defined as:

logic f: [r: C] (x: [r]) → [$x.s$]

then f [r] (y) has type [$y.s$]. In fact, if the variable name x of a parameter $\mathit{param}_i = (x\colon \tau)$ is used in the type of another formal parameter or in the return type τ', then arg_i must be a variable, as otherwise $\sigma(\tau')$ would not be defined. Indeed, if arg_i is not a variable then rule ARegular must be applied instead of AVariable for param_i, thus forcing $\sigma(x)$ to be undefined.

4.2.2 Terms and Predicates

Terms and Region Terms Terms are typed using the following judgement:

$$\frac{}{\Gamma \vdash c : \mathit{typeof}(c)}\ \textsc{TConst} \qquad \frac{}{\Gamma,\ x : \tau \vdash x : \tau}\ \textsc{TVar}$$

$$\frac{\Gamma \vdash LT_1 : \mathit{int} \qquad \Gamma \vdash LT_2 : \mathit{int} \qquad \mathit{op} \in \{+,\ -,\ \times\}}{\Gamma \vdash LT_1\ \mathit{op}\ LT_2 : \mathit{int}}\ \textsc{TIntOp}$$

$$\frac{\Gamma \vdash LT_1 : \mathit{bool} \qquad \Gamma \vdash LT_2 : \mathit{bool} \qquad \mathit{op} \in \{\&\&,\ ||\}}{\Gamma \vdash LT_1\ \mathit{op}\ LT_2 : \mathit{bool}}\ \textsc{TBoolOp}$$

$$\frac{\Gamma \vdash RT : \mathcal{C} \qquad \Gamma \vdash LT : [\_] \qquad f : \tau \in \mathcal{C}}{\Gamma \vdash \mathit{get}(RT,\ LT,\ f) : \tau}\ \textsc{TSelect}$$

$$\frac{f : \mathit{param}_1\ \cdots\ \mathit{param}_n\ \rightarrow\ \tau \qquad \text{for all } i : \Gamma,\ \sigma \vdash \mathit{larg}_i : \mathit{param}_i}{\Gamma \vdash f\ \mathit{larg}_1\ \cdots\ \mathit{larg}_n : \sigma(\tau)}\ \textsc{TApp}$$

$$\frac{L : R \in \Gamma \qquad r \in R}{\Gamma,\ r : \mathcal{C} \vdash r@L : \mathcal{C}}\ \textsc{RTVar} \qquad \frac{\Gamma \vdash RT : \mathcal{C}' \qquad \Gamma \vdash LT : [\_] \qquad s : \mathcal{C} \in \mathcal{C}'}{\Gamma \vdash \mathit{get}(RT,\ LT,\ s) : \mathcal{C}}\ \textsc{RTSelect}$$

Figure 4.7: Typing rules for terms

$$\frac{\Gamma \vdash RT : \sigma(\mathcal{C})}{\Gamma,\ \sigma \vdash [RT] : [r : \mathcal{C}]}\ \textsc{LARegion} \qquad \frac{\sigma(x) \text{ is undefined} \qquad \Gamma \vdash LT : \sigma(\tau)}{\Gamma,\ \sigma \vdash (LT) : (x : \tau)}\ \textsc{LARegular}$$

$$\frac{\sigma(x) = y \qquad \Gamma \vdash y : \sigma(\tau)}{\Gamma,\ \sigma \vdash (y) : (x : \tau)}\ \textsc{LAVariable}$$

Figure 4.8: Typing rules for logic arguments

$$\Gamma \vdash LT\text{: } \tau$$

This judgement states that under environment Γ, term LT has type τ. Region terms are typed using the following judgement:

$$\Gamma \vdash RT\text{: } \tau$$

This judgement states that under environment Γ, region term RT has class $\mathcal{C}$. Figure 4.7 shows typing rules for terms and region terms.

Rule TApp involves typing logic arguments with the following judgement:

$$\Gamma,\ \sigma \vdash \mathit{larg}\text{: } \mathit{param}$$

It is similar to the judgement for typing arguments in expressions and is defined in Figure 4.8.

Rules are similar to expression and region typing rules. The main differences are for the application rule TApp and for the selection rules TSelect and RTSelect, as region terms

$$\frac{}{\Gamma \vdash \mathsf{T}}\ \textsc{PTrue} \qquad \frac{}{\Gamma \vdash \mathsf{F}}\ \textsc{PFalse} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash \neg P}\ \textsc{PNot}$$

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2 \qquad op \in \{\wedge,\ \vee,\ \Rightarrow,\ \Longleftrightarrow\}}{\Gamma \vdash P_1\ op\ P_2}\ \textsc{POp}$$

$$\frac{\Gamma \vdash LT_1 : int \qquad \Gamma \vdash LT_2 : int \qquad op \in \{>,\ \geq\}}{\Gamma \vdash LT_1\ op\ LT_2}\ \textsc{PIntComp}$$

$$\frac{\Gamma \vdash LT_1 : \tau \qquad \Gamma \vdash LT_2 : \tau \qquad op \in \{=,\ \neq\}}{\Gamma \vdash LT_1\ op\ LT_2}\ \textsc{PComp}$$

$$\frac{\Gamma,\ x : \tau \vdash P \qquad quant \in \{\forall,\ \exists\}}{\Gamma \vdash quant\ x : \tau.\ P}\ \textsc{PQuant}$$

$$\frac{\Gamma,\ r : \mathcal{C},\ \textbf{here} : (R \cup \{r\}) \vdash P \qquad quant \in \{\forall,\ \exists\}}{\Gamma,\ \textbf{here} : R \vdash quant\ \textbf{region}\ r : \mathcal{C}.\ P}\ \textsc{PRegionQuant}$$

$$\frac{p : param_1\ \cdots\ param_n \qquad \text{for all } i : \Gamma,\ \sigma \vdash larg_i : param_i}{\Gamma \vdash p\ larg_1\ \cdots\ larg_n}\ \textsc{PApp}$$

$$\frac{\Gamma \vdash LT : [\_] \qquad \Gamma \vdash RT : \mathcal{C}}{\Gamma \vdash LT \in RT}\ \textsc{PIn}$$

Figure 4.9: Typing rules for predicates

RT are involved and *get* accept any pointer term LT, instead of just variables. We also check in RTVar that label L is in the environment.

Rule TSelect involves the following new judgement:

$$f\colon \tau \in \mathcal{C}$$

This judgement is similar to $f\colon \tau \in_x \mathcal{C}$, except that all pointer types are substituted to $[\_]$. Indeed, in the logic, pointer may belong to any region. Thus owner x is not needed. Similarly, rule RTSelect involve the following new judgement:

$$r\colon \mathcal{C} \in \mathcal{C}'$$

which also does not need any owner x as pointer types are replaced by $[\_]$.

Predicates Predicates are typed using the following judgement:

$$\Gamma \vdash P$$

This judgement states that under environment Γ, predicate P is well-typed. Figure 4.9 shows typing rules for predicates.

Rules PTRUE, PFALSE, PNOT and POP are straightforward. Typing rules PINTCOMP and PCOMP deal with comparisons. Typing rules PQUANT and PREGIONQUANT deal with quantifiers. They introduce new variables or region variables, respectively, in the environment to type the body. Note that regions are also added to special label **here**. Rule PAPP is similar to TAPP, except that there is no return type. It involves the following new judgement:

$$p\colon \mathit{param}_1 \cdots \mathit{param}_n$$

which is similar to the one used for terms. Rule PIN states how to type $LT \in RT$: we simply check that LT is a pointer (i.e. has type [_]) and that RT is well-typed.

Region Names Used by a Predicate Predicates appearing inside programs, i.e. in pre-conditions, post-conditions, assertions and invariants, *may only mention focus-free regions.* This is ensured by the typing rules for functions, classes and statements, which we introduce later.

Here we define function *freeregions*, which takes a predicate, a term or a region term as an argument and returns all free region names used by the argument. The definition is straightforward: it collects all region names r with label **here** appearing in the predicate which are not bound by ∀**region** r or ∃**region** r. In particular:

- $\mathit{freeregions}(r@\textbf{here}) = \{r\}$;
- $\mathit{freeregions}(r@L) = \emptyset$ if $L \neq \textbf{here}$;
- $\mathit{freeregions}(\mathit{get}(RT,\ LT,\ s)) = \mathit{freeregions}(RT) \cup \mathit{freeregions}(LT)$;
- $\mathit{freeregions}(\forall\textbf{region}\ r\colon \mathcal{C}.\ P) = \mathit{freeregions}(P) - r$;
- $\mathit{freeregions}(\exists\textbf{region}\ r\colon \mathcal{C}.\ P) = \mathit{freeregions}(P) - r$;
- $\mathit{freeregions}(p\ \mathit{larg}_1 \cdots \mathit{larg}_n) = \mathit{freeregions}(\mathit{larg}_1) \cup \cdots \cup \mathit{freeregions}(\mathit{larg}_n)$;
- $\mathit{freeregions}(f\ \mathit{larg}_1 \cdots \mathit{larg}_n) = \mathit{freeregions}(\mathit{larg}_1) \cup \cdots \cup \mathit{freeregions}(\mathit{larg}_n)$;

Note that typing rules enforce regions bound using quantifiers to only be used with label **here** anyway. Function *freeregions* is used to compute the set of regions which should be focus-free.

4.2.3 Statements

Sequences Statements are typed using the following judgement:

$$\Gamma \vdash \{\overline{\Sigma}\}\ s\ \{\overline{\Sigma}'\},\ \Gamma'$$

This judgement states that given environment Γ and permissions $\overline{\Sigma}$, statement s returns permissions $\overline{\Sigma}'$ and environment Γ', where $\overline{\Sigma}$ and $\overline{\Sigma}'$ are multisets. Indeed, statements may change permissions: they consume some and produce others. They also may introduce new variables or region variables in the environment.

Sequences of statements are typed using the following similar judgement:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S\ \{\overline{\Sigma}'\},\ \Gamma'$$

This judgement states that given environment Γ and permissions $\overline{\Sigma}$, sequence S returns permissions $\overline{\Sigma}'$ and environment Γ'. Here is the typing rule for sequences:

$$\frac{\textit{for all } i,\ \Gamma_i \vdash \{\overline{\Sigma}_i\}\ s_i\ \{\overline{\Sigma}_{i+1}\},\ \Gamma_{i+1}}{\Gamma_1 \vdash \{\overline{\Sigma}_1\}\ s_1;\ \cdots;\ s_n\ \{\overline{\Sigma}_{n+1}\},\ \Gamma_{n+1}}\ \textsc{SSeq}$$

We simply chain the statement judgements.

To type *blocks*, i.e. sequences of statements which limit the scope of variables, we introduce the following judgement:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S\ \{\overline{\Sigma}'\}$$

The environment is unchanged: the scope of variables and regions which are bound in the block is the block itself. This also implies that all permissions which mention such variable or region must be removed. This is shown by the rule for blocks:

$$\frac{\Gamma \vdash \{\overline{\Sigma}\}\ S\ \{\overline{\Sigma}'\},\ \Gamma'}{\Gamma \vdash \{\overline{\Sigma}\}\ S\ \{\mathit{scope}(\Gamma,\ \overline{\Sigma}')\}}\ \textsc{SBlock}$$

where $\mathit{scope}(\Gamma, \overline{\Sigma})$ is $\overline{\Sigma}$ without all permissions mentioning a variable x or a region r which is not in the left-hand side of a pair in environment Γ.

A consequence of the fact that the environment is unchanged is that if the type of a variable changes during execution because of an adoption, focus or unfocus, then its original type is restored. Consider for instance the following program:

```
let region r: Long;
let x = new Long [r];
if true then
{
  let region s: Long;
  weaken empty s;
  adopt x: r as s;
}
```

The type of x after the allocation is $[r]$. After the adoption it is $[s]$. But after the **if** statement, the type is restored to $[r]$. It is *safe* for x to have type $[r]$, because typing rules prevent accessing x with type $[r]$ if no permission on r is available, and the adoption consumes $r^{\times}$. Note that we could consider having both x: $[r]$ and x: $[s]$ in the environment after the adoption.

Statements Figures 4.10 and 4.11 shows typing rules for statements. These typing rules are the most important rules of Capucine, as they are the ones which deal with permissions. Rule SCall is especially important, as it requires region arguments to be *separated*.

Rule SLet states how to type let-bindings. The expression is typed, and the variable is given the type of the expression in the new environment. No permission is consumed or produced.

Rule SIf states how to type **if** statements. The condition must have type *bool*. The **then** body and the **else** body must consume and produce the same permissions. In other words, from permissions $\overline{\Sigma}$ available before the **if**, both bodies must return the same set

$$\frac{\Gamma,\ \overline{\Sigma} \vdash e :\ \tau}{\Gamma \vdash \{\overline{\Sigma}\}\ \textsf{let}\ x\ =\ e\ \{\overline{\Sigma}\},\ (\Gamma,\ x :\ \tau)}\ \textsc{SLet}$$

$$\frac{\Gamma,\ \overline{\Sigma} \vdash e :\ \mathit{bool} \qquad \Gamma \vdash \{\overline{\Sigma}\}\ s_1;\ \cdots;\ s_n\ \{\overline{\Sigma}'\} \qquad \Gamma \vdash \{\overline{\Sigma}\}\ s'_1;\ \cdots;\ s'_m\ \{\overline{\Sigma}'\}}{\Gamma \vdash \{\overline{\Sigma}\}\ \textsf{if}\ e\ \textsf{then}\ \{s_1;\ \cdots;\ s_n\}\ \textsf{else}\ \{s'_1;\ \cdots;\ s'_m\}\ \{\overline{\Sigma}'\},\ \Gamma}\ \textsc{SIf}$$

$$\frac{\begin{array}{c} f\ \textsf{consumes}\ \overline{\Sigma}_1 \qquad f\ \textsf{produces}\ \overline{\Sigma}_2 \qquad f :\ \mathit{param}_1\ \cdots\ \mathit{param}_n\ \rightarrow\ \tau \\ \text{for all } i :\ \Gamma,\ \overline{\Sigma},\ \sigma \vdash \mathit{arg}_i :\ \mathit{param}_i \qquad \mathit{separated}(\mathit{arg}_1,\ \cdots,\ \mathit{arg}_n) \end{array}}{\Gamma \vdash \{\overline{\Sigma},\ \sigma(\overline{\Sigma}_1)\}\ \textsf{let}\ x\ =\ f\ \mathit{arg}_1\ \cdots\ \mathit{arg}_n\ \{\overline{\Sigma},\ \sigma(\overline{\Sigma}_2)\},\ (\Gamma,\ x :\ \sigma(\tau))}\ \textsc{SCall}$$

$$\frac{\Gamma \vdash \mathcal{C}}{\Gamma \vdash \{\overline{\Sigma}\}\ \textsf{let region}\ r :\ \mathcal{C}\ \{\overline{\Sigma},\ r^{\emptyset}\},\ (\Gamma,\ r :\ \mathcal{C})}\ \textsc{SLetRegion}$$

$$\frac{\Gamma \vdash \mathcal{C} \qquad \Gamma \vdash \rho :\ \mathcal{C} \qquad \Gamma,\ \{\overline{\Sigma},\ \rho^{\emptyset}\} \vdash \rho \qquad \overline{f} \text{ are the fields declared in } \mathcal{C}}{\Gamma \vdash \{\overline{\Sigma},\ \rho^{\emptyset}\}\ \textsf{let}\ x\ =\ \textsf{new}\ \mathcal{C}\ [\rho]\ \{\overline{\Sigma},\ \rho^{\circ}\{\overline{f}\}\},\ \textsf{own}(x,\ \mathcal{C})^{\emptyset}\},\ (\Gamma,\ x :\ [\rho])}\ \textsc{SNew}$$

$$\frac{\Gamma,\ \{\overline{\Sigma},\ \sigma^{\emptyset},\ \rho^{G}\} \vdash \rho \qquad \Gamma,\ \{\overline{\Sigma},\ \sigma^{\emptyset},\ \rho^{G}\} \vdash \sigma}{\Gamma,\ x :\ [\rho] \vdash \{\overline{\Sigma},\ \sigma^{\emptyset},\ \rho^{G}\}\ \textsf{focus}\ x : \rho\ \textsf{as}\ \sigma\ \{\overline{\Sigma},\ \sigma^{\times},\ \sigma\ \multimap\ \rho\},\ (\Gamma,\ x :\ [\sigma])}\ \textsc{SFocus}$$

$$\frac{\begin{array}{c} \Gamma \vdash \rho :\ \mathcal{C} \\ \Gamma,\ \{\overline{\Sigma},\ \rho^{\circ}\{\overline{f}\}\ \} \vdash \rho \qquad f :\ \tau \in_x \mathcal{C} \qquad (\Gamma,\ x :\ [\rho]),\ \{\overline{\Sigma},\ \rho^{\circ}\{\overline{f}\}\ \} \vdash e :\ \tau \end{array}}{\Gamma,\ x :\ [\rho] \vdash \{\overline{\Sigma},\ \rho^{\circ}\{\overline{f}\}\ \}\ x.f\ \leftarrow\ e\ \{\overline{\Sigma},\ \rho^{\circ}\{\overline{f} - f\}\ \},\ \Gamma}\ \textsc{SAssign}$$

Figure 4.10: Typing rules for statements (1 of 2)

$$\frac{\Gamma,\ \{\overline{\Sigma},\ \sigma^{\times},\ \rho^{G}\} \vdash \rho \qquad \Gamma,\ \{\overline{\Sigma},\ \sigma^{\times},\ \rho^{G}\} \vdash \sigma}{\Gamma,\ x:\ [\sigma] \vdash \{\overline{\Sigma},\ \sigma^{\times},\ \rho^{G}\}\ \textsf{adopt}\ x : \sigma\ \textsf{as}\ \rho\ \{\overline{\Sigma},\ \rho^{G}\},\ (\Gamma,\ x:\ [\rho])}\ \textsc{SAdopt}$$

$$\frac{\Gamma,\ \{\overline{\Sigma},\ \sigma^{\times},\ \sigma \multimap \rho\} \vdash \rho \qquad \Gamma,\ \{\overline{\Sigma},\ \sigma^{\times},\ \sigma \multimap \rho\} \vdash \sigma}{\Gamma,\ x:\ [\sigma] \vdash \{\overline{\Sigma},\ \sigma^{\times},\ \sigma \multimap \rho\}\ \textsf{unfocus}\ x : \sigma\ \textsf{as}\ \rho\ \{\overline{\Sigma},\ \rho^{G}\},\ (\Gamma,\ x:\ [\rho])}\ \textsc{SUnfocus}$$

$$\frac{\Gamma \vdash \rho:\ \mathcal{C}}{\Gamma,\ x:\ [\rho] \vdash \{\overline{\Sigma},\ \rho^{\times}\}\ \textsf{unpack}\ x\ \{\overline{\Sigma},\ \rho^{\circ},\ \textsf{single}(x,\ \mathcal{C})^{\times},\ \textsf{group}(x,\ \mathcal{C})^{G}\},\ \Gamma}\ \textsc{SUnpack}$$

$$\frac{\begin{array}{c}\Gamma \vdash \rho:\ \mathcal{C} \qquad \Gamma,\ \{\overline{\Sigma},\ \rho^{\circ},\ \textsf{single}(x,\ \mathcal{C})^{\times},\ \textsf{group}(x,\ \mathcal{C})^{G}\} \vdash \rho \\ \Gamma,\ \{\overline{\Sigma},\ \rho^{\circ},\ \textsf{single}(x,\ \mathcal{C})^{\times},\ \textsf{group}(x,\ \mathcal{C})^{G}\} \vdash_{\multimap} \mathit{root}(\rho)\end{array}}{\Gamma,\ x:\ [\rho] \vdash \{\overline{\Sigma},\ \rho^{\circ},\ \textsf{single}(x,\ \mathcal{C})^{\times},\ \textsf{group}(x,\ \mathcal{C})^{G}\}\ \textsf{pack}\ x:\ \mathcal{C}\ \{\overline{\Sigma},\ \rho^{\times}\},\ \Gamma}\ \textsc{SPack}$$

$$\frac{\Gamma,\ \{\overline{\Sigma},\ \rho^{\emptyset}\} \vdash \rho}{\Gamma \vdash \{\overline{\Sigma},\ \rho^{\emptyset}\}\ \textsf{weaken empty}\ \rho\ \{\overline{\Sigma},\ \rho^{G}\},\ \Gamma}\ \textsc{SWeakenEmpty}$$

$$\frac{\Gamma,\ \{\overline{\Sigma},\ \rho^{\times}\} \vdash \rho}{\Gamma \vdash \{\overline{\Sigma},\ \rho^{\times}\}\ \textsf{weaken single}\ \rho\ \{\overline{\Sigma},\ \rho^{G}\},\ \Gamma}\ \textsc{SWeakenSingle}$$

$$\frac{\mathit{logicenv}(\Gamma),\ \textsf{here} \vdash P \qquad \text{for all } r \in \mathit{freeregions}(P)\text{: } \Gamma,\ \overline{\Sigma} \vdash_{\multimap} r}{\Gamma \vdash \{\overline{\Sigma}\}\ \textsf{assert}\ P\ \{\overline{\Sigma}\},\ \Gamma}\ \textsc{SAssert}$$

$$\frac{L \notin \Gamma \qquad \text{for all } r \in R\text{, there is } \mathcal{C} \text{ such that } r\text{: } \mathcal{C} \in \Gamma}{\Gamma \vdash \{\overline{\Sigma}\}\ \textsf{label}\ L\ \{\overline{\Sigma}\},\ (\Gamma,\ L:\ R)}\ \textsc{SLabel}$$

Figure 4.11: Typing rules for statements (2 of 2)

of permissions $\overline{\Sigma}'$, which is also the set of permissions returned by the **if** statement. The environment is unchanged, as the scope of variables and regions which are bound in the bodies are their respective **then** or **else** block.

Rule SCall states how to type function calls. First we type region arguments and normal arguments. We obtain a substitution σ. We use σ to compute consumed permissions and produced permissions. Finally, we add the new bound variable x to the environment, with the return type of the function, substituted using σ. This rule uses three new judgements, used to retrieve typing information associated to a function:

$$f \textbf{ consumes } \overline{\Sigma}$$

which states that the declaration of f states that f consumes permissions $\overline{\Sigma}$,

$$f \textbf{ produces } \overline{\Sigma}$$

which states that the declaration of f states that f produces permissions $\overline{\Sigma}$, and

$$f\colon \overline{param}\colon \tau$$

which gives the region names and their classes in the declaration of f, the argument names and their types, and the return type.

The SCall rule also requires $separated(arg_1, \cdots, arg_n)$, which we now define as follows: for all i, j, if $arg_i = [\rho]$ and $arg_j = [\sigma]$, then $root(\rho) \neq root(\sigma)$. Function $root(\sigma)$ is defined as follows: $root(r) = r$, and $root(x.s) = root(\rho_x)$ where $x\colon [\rho_x]$ is in Γ. Informally, *separated* states that region arguments are disjoint. We know that two regions are disjoint if their roots are different. For instance, a function f expecting two region arguments cannot be called as $f\ [\rho]\ [x.s]$ if x has type $[\rho]$. The *separated* condition of SCall ensures that we know how to translate the call in Why, and thus that we can compute proof obligations.

Rule SLetRegion states how to type region binding. First we check that the given class $\mathcal{C}$ is well-formed using the following new judgement:

$$\Gamma \vdash \mathcal{C}$$

This judgement simply checks that all variables, all regions and all type variables are bound in $\mathcal{C}$. Then we enter the new region r in the environment, with class $\mathcal{C}$. This operation produces permission $r^{\emptyset}$: region r is initially empty.

Rule SNew states how to type allocation in region ρ. We first type ρ and its class $\mathcal{C}$. Region ρ must be empty before the allocation. We check this by consuming permission $\rho^{\emptyset}$. Permission $\rho^{\circ}\{\overline{f}\}$ where $\overline{f}$ are the fields of $\mathcal{C}$ is produced, stating that after the allocation the region is no longer empty but singleton and that no field is initialized. Permissions $\textbf{own}(x, \mathcal{C})^{\emptyset}$ are also produced. Notation $\textbf{own}(x, \mathcal{C})$ denotes the set $\{x.r_1, \cdots, x.r_n\}$ where r_i are the owned regions in the declaration of class $\mathcal{C}$. Notation $\{\rho_1, \cdots, \rho_n\}^{\emptyset}$ denotes ${\rho_1}^{\emptyset}, \cdots, {\rho_n}^{\emptyset}$. In other words, the owned regions of a newly-allocated pointer are initially empty.

Rule SFocus states how to type the **focus** $x{:}\rho$ **as** σ operation. Variable x must have type $[\rho]$ before, and its type is changed to $[\sigma]$. The operation consumes permissions $\sigma^{\emptyset}$ and ρ^{G}: source region ρ must be group, and target region σ must be empty. Target region becomes singleton: permission $\sigma^{\times}$ is produced. Region ρ is temporarily disabled: permission $\sigma \multimap \rho$ is produced.

Rule SAssign states how to type assignment to a field $x.f$. First we check that x is a pointer and we look for its region ρ in the environment. We then look for the class $\mathcal{C}$ of ρ and use this information to obtain the type τ of $x.f$. We check the type of expression e is

τ. We then check that x is open, i.e. that permission $\rho^\circ\{\overline{f}\}$ is available. This permission is consumed and replaced by $\rho^\circ\{\overline{f}-f\}$, where $\overline{f}-f$ is $\overline{f}$ without field f if it was in $\overline{f}$, and $\overline{f}$ otherwise. In other words, if f was not initialized, it is now.

Rule SAdopt states how to type the **adopt** $x{:}\sigma$ **as** ρ operation. It changes the type of x, which must be $[\sigma]$, to $[\rho]$. It also consumes permissions $\sigma^\times$ (region σ must be singleton and closed) and ρ^G (region ρ must be group). No permission on σ is produced: the region is no longer usable. Permission ρ^G is reproduced.

Rule SUnfocus states how to type the **unfocus** operation. It is essentially the same as the **adopt** operation: a pointer is moved from a singleton region to a group region. However, **unfocus** deals with the case where the pointer already was in the group region once. Hence the only difference is that instead of consuming permission ρ^G, the operation consumes $\sigma \multimap \rho$.

Rules SUnpack and SPack state how to type unpacking and packing, respectively. Unpacking a pointer x requires finding its region ρ in the environment. We then look for its class $\mathcal{C}$. We can then compute owned permissions. Notations **single**$(x, \mathcal{C})$ and **group**$(x, \mathcal{C})$ are similar to **own**$(x, \mathcal{C})$ except that instead of denoting all owned regions, they are restricted to regions which were declared as singleton and group, respectively. Unpacking consumes $\rho^\times$ (region ρ is closed) and produces ρ° (region ρ is open), as well as permissions on owned regions. Packing does the opposite. Note that packing x requires all fields of x to be initialized. Also note that packing requires the permissions on ρ to be complete and its root must be focus-free, as the invariant is read. Unpacking does not.

Rules SWeakenEmpty and SWeakenSingle state how to type the weakening operations. They simply consume the expected permission $\rho^\emptyset$ or $\rho^\times$, respectively, and produce the group permission ρ^G.

Rule SAssert states that to type the **assert** P operation, we type the P predicate under environment $\mathit{logicenv}(\Gamma)$. The $\mathit{logicenv}$ function substitutes all pointer types $[\rho]$, for any region ρ, into type [_] which is the type for pointers in the logic. For instance, environment:

$$\Gamma = x\text{: } [r],\ r\text{: } \mathit{List}\ ([y.s]),\ y\text{: } [r'],\ r'\text{: } \mathit{Long}$$

becomes:

$$\mathit{logicenv}(\Gamma) = x\text{: } [\_],\ r\text{: } \mathit{List}\ ([\_]),\ y\text{: } [\_],\ r'\text{: } \mathit{Long}$$

Label **here** is available when typing P. Note that all regions used by the predicate should be focus-free.

Finally, rule SLabel states how to type the **label** L statement. We check that L is not already in the environment, and then we add it.

4.2.4 Declarations

A program is a sequence of *declarations*. The user may declare or define logic functions, predicates, axioms, lemmas, classes and functions.

Logic Declarations A logic type declaration:

type $t\ (\alpha_1, \cdots, \alpha_n)$

is quite easy to type: we simply check that for all $i \neq j$, $\alpha_i \neq \alpha_j$. The scope of a type declaration is the remainder of the program, i.e. type t may be used in all declarations which follow its declaration.

A predicate declaration:

logic $p\ param_1\ \cdots\ param_n = P$

or a logic function declaration:

logic $f\ param_1\ \cdots\ param_n\colon \tau = LT$

requires checking that each argument $param_i$ is well-formed. A region argument $[r\colon \mathcal{C}]$ or a regular argument $(x\colon \tau)$ is well-formed if class expression $\mathcal{C}$ or type τ only mentions region variables r and variables x bound in previous parameters $param_1\ \cdots\ param_{i-1}$. We also check that each parameter name is unique. We then check that pointer types, being logic pointers and not program pointers, are of the form $[\_]$ (and not $[\rho]$). The same restriction applies for the return type τ for logic functions. All class expressions $\mathcal{C}$ and types τ may mention any number of free type variable, which are then considered implicitely universally quantified, i.e. the predicate p or the function f is polymorphic.

The body of a predicate declaration such as the one above is typed as follows:

$$param_1\ \cdots\ param_n,\ \textbf{here}\colon \emptyset \vdash P$$

Similarly, the body of a logic function is typed as follows:

$$param_1\ \cdots\ param_n,\ \textbf{here}\colon \emptyset \vdash LT\colon \tau$$

If the body is not given by the user, this type-checking of course does not happen. Label **here** is available when typing P and LT. It is the only label available.

The scope of a logic function or predicate is the remainder of the program. It cannot be used inside its own body. To define a recursive logic function or predicate, the user needs to give an axiomatization.

Axioms and lemmas defined as:

axiom $a\colon P$
lemma $l\colon P$

are type-checked as follows:

$$\textbf{here}\colon \emptyset \vdash P$$

In other words, the body P is typed in the empty environment. Note that we have not explicitely shown the environment for polymorphic type variables. In practice, free type variables in the body P are added to this environment. Label **here** is available when typing P. Axioms and lemmas are available in the context of all proof obligations which are generated by declarations which follow the declaration of the axiom or lemma.

Class Declarations Class declarations:

class $C\ (\overline{\alpha})\ [\overline{r\colon \mathcal{C}}]$
{
$\overline{size\ s\colon \mathcal{C}};$
$\overline{f\colon \tau};$
invariant $P;$
}

are typed as follows. First, each class expression $\mathcal{C}$ is typed in an environment where previously-declared region variables and regular variables are available. We allow recursive types: class C is known when typing all class expressions in its definition. Thus the

scope of C is the remainder of the program, plus its own definition. Field types τ are also type-checked: all regions and type variables must have been declared before in the class definition. Moreover, all pointer types must be of the form $[\rho]$, i.e. not $[\_]$. And of course all class expressions must have the right number of arguments (regions and types).

The invariant P is type-checked as follows:

$$\mathit{logicenv}(r_1\colon \mathcal{C}_1, \cdots, r_n\colon \mathcal{C}_n, f_1\colon \tau_1, \cdots, f_n\colon \tau_n), \textsf{here}\colon \emptyset \vdash P$$

where $r_1, \cdots, r_n$ are all region parameters and owned regions of class C, of respective classes $\mathcal{C}_1, \cdots, \mathcal{C}_n$. Fields are available in P. Label **here** is available when typing P.

Restriction on Class Invariants We add another important restriction to the invariant predicate of classes: it cannot read from a region which is not owned by class C. For each subterm $\mathit{get}(r, LT, f)$, where r is a region variable, we require that r is an owned region of C (i.e. not a region parameter). This is fundamental for the invariant methodology. It is similar to the restriction of Spec# [Barnett04b].

Function Declarations Function declarations:

fun f $\mathit{param}_1 \cdots \mathit{param}_n\colon \tau$
consumes $\overline{\Sigma}$
produces $\overline{\Sigma}'$
pre P
post Q
{
S;
return e
}

are typed as follows. First we check each argument *param* by requiring region variables and regular variables to be bound, i.e. to be region parameters of the function appearing before. Free type variables are polymorphic type variables. Let Γ be the following environment:

$$\Gamma = \mathit{param}_1 \cdots \mathit{param}_n$$

We type the pre- and post-conditions P and Q in this environment:

$$\Gamma, \textsf{here}\colon R \vdash P$$

$$\Gamma, \textsf{here}\colon R, \textsf{pre}\colon R \vdash Q$$

where R is the set of regions defined in arguments *param*. Label **here** is available both in the pre-condition and the post-condition. In the post-condition, label **pre** is also available; it denotes the state of the program at the beginning of the function execution. We also require, for all r in $\mathit{freeregions}(P)$:

$$\Gamma, \overline{\Sigma} \vdash_{\multimap} r$$

and for all r in $\mathit{freeregions}(Q)$:

$$\Gamma, \overline{\Sigma}' \vdash_{\multimap} r$$

We then type the body:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S\ \{\overline{\Sigma}_0\}, \Gamma'$$

We then check that $\overline{\Sigma}' = \mathit{scope}(\Gamma, \overline{\Sigma}_0)$. Finally, we type the returned expression e in the final environment:

$$\Gamma', \overline{\Sigma}_0 \vdash e\colon \tau$$

Its type must be equal to the declared return type.

The scope of a function name f is the remainder of the program, plus its own body. Indeed, all functions may be recursive.

Note that just as we do not prevent the user from writing F as a pre-condition, we do not require consumed permissions to be consistent. For instance, the user may require twice the same permission $\rho^{\times}$. However, the function will not be callable, as typing rule SCall requires all regions to be *separated*.

4.3 Intuitive Memory Model

Partial Functions We use *partial functions* to model Capucine heaps. A partial function f has a *domain*, denoted by $\mathit{Dom}(f)$. It is defined as follows: $f(x)$ is defined if, and only if $x \in \mathit{Dom}(f)$.

We denote by $f[x \mapsto v]$ the partial function g such that:

- $\mathit{Dom}(g) = \mathit{Dom}(f) \cup \{x\}$;
- $g(x) = v$;
- for all $y \in \mathit{Dom}(f) - x$, we have $g(y) = f(y)$.

In other words, if $x \in \mathit{Dom}(f)$, its value is *replaced* by v. If $x \notin \mathit{Dom}(f)$, then x is *added* to the domain of the function and its value is now v.

Model: Values, Objects, Active Regions and Locations A *value* is either a location p, an integer, a boolean or unit.

The *active region* of a location is the region ρ in which its object should be read. Usually, a pointer belongs to only one region, which is then its active region. But a focused pointer belongs to several regions at the same time, and it must be read from the region in which it has been temporarily extracted. Active regions are not important for the intuitive model, they are only used to relate the intuitive model with the separated model.

An *object* is a function o from field names f to values, and from region names r to sets of locations. These are the regions which are owned by the object. Set $o(r)$ contains all locations which are in the region r owned by o. This set does not contain transitively owned pointers, only owned pointers. Note that $o(r)$ may contain locations whose active region is not r.

We assume locations to be infinite, countable and ordered. Given a location l, we will denote the location following l using $l{+}1$.

Model: Logic Regions Functions A *logic region* is a function h from locations to objects, and from a special symbol *contents* to a set of pointers. The contents $h(contents)$ contains locations which are part of the region. In practice, it is used to interpret predicate *inRegion*.

The domain $Dom(h)$ of a logic region h contains not only the locations of the region, but also all (transitively) owned locations. We thus have $h(contents) \subseteq Dom(h)$.

To illustrate the difference between $Dom(h)$ and $h(contents)$, consider the following Capucine program:

```
class Long
{
  value: int;
}

class Pair
{
  single Rleft: Long;
  single Rright: Long;
  left: [Rleft];
  right: [Rright];
}

logic PairSum [r: Pair] (p: [r]) =
  p.left.value + p.right.value
```

Assume logic region h models the region r of logic function *PairSum*. Then the domain of h is the following set of locations:

$$Dom(h) = \{p,\ p.left,\ p.right\}$$

whereas the contents of h is the following set of locations:

$$h(contents) = \{p\}$$

That is, assuming region r is singleton and contains only p.

Note that $Dom(h)$ corresponds to the notion of *footprint* sometimes used in the literature. In the above example, it is exactly the footprint of logic function *PairSum*. In general, the footprint of a logic function is a subset of the union of the domain of all logic regions which are passed as parameters to the logic function.

Model: Heap We model the heap using a function $\mathcal{H}$:

- $\mathcal{H}(p)$ returns the object at location p;
- $\mathcal{H}(x)$ returns the value of variable x;
- $\mathcal{H}(r)$ returns the set of pointers of region name r;
- $\mathcal{H}(p?)$ returns the active region ρ (of the form r or $x.s$) of pointer p;
- $\mathcal{H}(next)$ returns the next free location to be allocated;
- $\mathcal{H}(L)$ returns a function from region names r to the logic region function corresponding to r at label L.

Note that locations p, variables x, region variables r and labels L do not share the same namespace and *next* is a unique symbol, so there is no ambiguity.

Logic Model Logic values are the same as program values, and we simply call them *values*. Logic objects are also the same as objects: functions from field names f to values v and from owned region names s to sets of pointers $\mathcal{P}$.

A *logic environment* $\mathcal{L}$ is a function:

- $\mathcal{L}(L)$ returns a function from region names r to the logic region function corresponding to r at label L;
- $\mathcal{L}(x)$ returns the value of variable x.

To give semantics to function applications, we assume a model of user-declared types, functions and predicates. This model, given a function or a predicate f declared as:

logic f $param_1$ $\cdots$ $param_n$ $\cdots$

defines $[\![f]\!]$ as a function taking n arguments. This function $[\![f]\!]$ is assumed to model user axioms. In the case of predicates p, $[\![p]\!](\overline{x})$ either holds or does not hold.

From Programs to Logic We define the *flattening* of a set of locations $\mathcal{P}$ with respect to a heap $\mathcal{H}$ as all locations in a region transitively owned by $\mathcal{P}$. Formally, the flattening of $\mathcal{P}$ is the set $\mathcal{P}'$ such that p is in $\mathcal{P}'$ if, and only if there is $q_1, \cdots, q_n$ such that $q_1 \in \mathcal{P}$, $p = q_n$ and for all $i > 1$, there is an owned region name r such that $q_i \in \mathcal{H}(q_{i-1})(r)$.

We define function *lor* (logic of region): if $\mathcal{H}$ is a heap and $\mathcal{P}$ is a set of locations, $lor(\mathcal{H}, \mathcal{P})$ is the logic region corresponding to $\mathcal{P}$ in $\mathcal{H}$. Assume $\mathcal{P}'$ is the flattening of $\mathcal{P}$ with respect to $\mathcal{H}$. Domain $Dom(lor(\mathcal{H}, \mathcal{P}))$ is $\mathcal{P}' \cup \{contents\}$. For all p in $\mathcal{P}'$, we define:

$$lor(\mathcal{H}, \mathcal{P})(p) = \mathcal{H}(p)$$

We also define:

$$lor(\mathcal{H}, \mathcal{P})(contents) = \mathcal{P}$$

We define *loh* (logic environment of heap) as a function from heaps $\mathcal{H}$ to logic environments $\mathcal{L}$. The domain of *loh* is the domain of $\mathcal{H}$ restricted to variables x and region names r. For each region name r in $Dom(\mathcal{H})$, we define the region function associated to r at label **here** as follows:

$$loh(\mathcal{H})(\textbf{here})(r) = lor(\mathcal{H}, \mathcal{H}(r))$$

For each label L in $Dom(\mathcal{H})$, we define:

$$loh(\mathcal{H})(L) = \mathcal{H}(L)$$

For each variable name x in $Dom(\mathcal{H})$, we define:

$$loh(\mathcal{H})(x) = \mathcal{H}(x)$$

Note that the definition of the flattening of $\mathcal{P}$ was given for a heap $\mathcal{H}$, but it also applies for a logic region h, as $h(q)(r)$ makes sense as well. We thus define $lor(h, \mathcal{P})$ as a logic region of domain *contents* plus the flattening of $\mathcal{P}$ with respect to h. For all p in this flattening, we define:

$$lor(h, \mathcal{P})(p) = h(p)$$

We also define:

$$lor(h, \mathcal{P})(contents) = \mathcal{P}$$

From Programs to Logic: Example We illustrate function *loh* on an example. Consider class *Pair* which was introduced above. Consider the following Capucine program:

let region r: $Pair$;
let $pair$ = **new** $Pair$ $[r]$;
let $left$ = **new** $Long$ $[pair.Rleft]$;
$left.value \leftarrow 42$;
let $right$ = **new** $Long$ $[pair.Rright]$;
$right.value \leftarrow 69$;

If we start with an empty heap and execute the program, we obtain heap $\mathcal{H}$, where $\mathcal{H}$ is defined as the smallest partial function such that:

$\mathcal{H}(r) = \{0\}$
$\mathcal{H}(pair) = 0$
$\mathcal{H}(left) = 1$
$\mathcal{H}(right) = 2$
$\mathcal{H}(0)(Rleft) = \{1\}$
$\mathcal{H}(0)(Rright) = \{2\}$
$\mathcal{H}(0)(left) = 1$
$\mathcal{H}(0)(right) = 2$
$\mathcal{H}(1)(value) = 42$
$\mathcal{H}(2)(value) = 69$
$\mathcal{H}(0?) = r$
$\mathcal{H}(1?) = pair.Rleft$
$\mathcal{H}(2?) = pair.Rright$
$\mathcal{H}(next) = 3$

Logic environment $loh(\mathcal{H})$ is the smallest partial function such that:

$loh(\mathcal{H})(pair) = 0$
$loh(\mathcal{H})(left) = 1$
$loh(\mathcal{H})(right) = 2$
$loh(\mathcal{H})(\mathbf{here})(r)(contents) = \{0\}$
$loh(\mathcal{H})(\mathbf{here})(r)(0)(Rleft) = \{1\}$
$loh(\mathcal{H})(\mathbf{here})(r)(0)(Rright) = \{2\}$
$loh(\mathcal{H})(\mathbf{here})(r)(0)(left) = 1$
$loh(\mathcal{H})(\mathbf{here})(r)(0)(right) = 2$
$loh(\mathcal{H})(\mathbf{here})(r)(1)(value) = 42$
$loh(\mathcal{H})(\mathbf{here})(r)(2)(value) = 69$

We have $Dom(loh(\mathcal{H})(\mathbf{here})(r)) = \{contents, 0, 1, 2\}$.

Term Evaluation We denote by:

$$[\![LT]\!]_{\mathcal{L}}$$

$[\![c]\!]_{\mathcal{L}} = c$
$[\![LT_1 \ termop \ LT_2]\!]_{\mathcal{L}} = [\![LT_1]\!]_{\mathcal{L}} \ termop \ [\![LT_2]\!]_{\mathcal{L}}$
$[\![x]\!]_{\mathcal{L}} = \mathcal{L}(x)$
$[\![get(RT,\ LT,\ f)]\!]_{\mathcal{L}} = [\![RT]\!]_{\mathcal{L}}([\![LT]\!]_{\mathcal{L}})(f)$
$[\![f \ larg_1 \ \cdots \ larg_n]\!]_{\mathcal{L}} = [\![f]\!] \ ([\![larg_1]\!]_{\mathcal{L}},\ \cdots,\ [\![larg_n]\!]_{\mathcal{L}})$

Semantics of logic arguments *larg*:
$[\![\ [RT]\ ]\!]_{\mathcal{L}} = [\![RT]\!]_{\mathcal{L}}$
$[\![\ (LT)\ ]\!]_{\mathcal{L}} = [\![LT]\!]_{\mathcal{L}}$

Figure 4.12: Term Semantics (Intuitive Model)

$[\![r@L]\!]_{\mathcal{L}} = \mathcal{L}(L)(r)$
$[\![get(RT,\ LT,\ s)]\!]_{\mathcal{L}} = lor([\![RT]\!]_{\mathcal{L}},\ [\![RT]\!]_{\mathcal{L}}([\![LT]\!]_{\mathcal{L}})(s))$

Figure 4.13: Region Term Semantics (Intuitive Model)

the evaluation of term LT in logic environment $\mathcal{L}$. It returns a value v. Figure 4.12 gives the definition of $[\![LT]\!]_{\mathcal{L}}$.

We denote by:

$$[\![RT]\!]_{\mathcal{L}}$$

the evaluation of region term RT in $\mathcal{L}$. It returns a logic region. Figure 4.13 gives the definition of $[\![RT]\!]_{\mathcal{L}}$.

Semantics for constants, operators and variables are standard. Application uses the assumed user-defined model $[\![f]\!]$. Field selection $get(RT,\ LT,\ f)$ reads location $[\![LT]\!]_{\mathcal{L}}$ in logic region $[\![RT]\!]_{\mathcal{L}}$, and then reads field f in the obtained logic object. The semantics of a region name r at label L is $\mathcal{L}(L)(r)$. The semantics of a region term $get(RT,\ LT,\ s)$ uses *roh* to restrict the logic region obtained from RT, to the flattening of the set of locations in region s owned by LT.

Predicate Evaluation We denote by:

$$[\![P]\!]_{\mathcal{L}}$$

the evaluation of predicate P in logic environment $\mathcal{L}$. It either holds or does not hold. Figure 4.14 gives the definition of $[\![P]\!]_{\mathcal{L}}$.

Definition of $[\![P]\!]_{\mathcal{L}}$ is standard for most operators. The first novelty is $[\![p\ \overline{larg}]\!]_{\mathcal{L}}$ where p is a predicate name, which is similar to $[\![f\ \overline{larg}]\!]_{\mathcal{L}}$ where f is a logic function name. The other novelties are $[\![\forall$**region** r: $\mathcal{C}$. $P]\!]_{\mathcal{L}}$ and $[\![\exists$**region** r: $\mathcal{C}$. $P]\!]_{\mathcal{L}}$, which quantify over logic regions. Quantified regions are available at label **here**, the default label.

Expression Evaluation We now give semantics to expression. We denote by:

$$[\![e]\!]_{\mathcal{H}}$$

the evaluation of expression e in heap $\mathcal{H}$. It returns a value v. Figure 4.15 gives the definition of $[\![e]\!]_{\mathcal{H}}$.

$[\![p\ larg_1\ \cdots\ larg_n]\!]_{\mathcal{L}} = [\![p]\!]\ ([\![arg_1]\!]_{\mathcal{L}}, \cdots, [\![arg_n]\!]_{\mathcal{L}})$
$[\![P_1 \wedge P_2]\!]_{\mathcal{L}}$ holds iff $[\![P_1]\!]_{\mathcal{L}}$ holds and $[\![P_2]\!]_{\mathcal{L}}$ holds
$[\![P_1 \vee P_2]\!]_{\mathcal{L}}$ holds iff $[\![P_1]\!]_{\mathcal{L}}$ holds or $[\![P_2]\!]_{\mathcal{L}}$ holds
$[\![P_1 \Rightarrow P_2]\!]_{\mathcal{L}}$ holds iff $[\![P_2]\!]_{\mathcal{L}}$ holds or $[\![P_1]\!]_{\mathcal{L}}$ does not hold
$[\![P_1 \iff P_2]\!]_{\mathcal{L}}$ holds iff $[\![P_1]\!]_{\mathcal{L}}$ holds if and only if $[\![P_2]\!]_{\mathcal{L}}$ holds
$[\![\neg P]\!]_{\mathcal{L}}$ holds iff $[\![P]\!]_{\mathcal{L}}$ does not hold
$[\![\mathsf{T}]\!]_{\mathcal{L}}$ holds
$[\![\mathsf{F}]\!]_{\mathcal{L}}$ does not hold
$[\![LT_1 = LT_2]\!]_{\mathcal{L}}$ holds iff $[\![LT_1]\!]_{\mathcal{L}} = [\![LT_2]\!]_{\mathcal{L}}$
$[\![LT_1 \neq LT_2]\!]_{\mathcal{L}}$ holds iff $[\![LT_1]\!]_{\mathcal{L}} \neq [\![LT_2]\!]_{\mathcal{L}}$
$[\![LT_1 > LT_2]\!]_{\mathcal{L}}$ holds iff $[\![LT_1]\!]_{\mathcal{L}} > [\![LT_2]\!]_{\mathcal{L}}$
$[\![LT_1 \geq LT_2]\!]_{\mathcal{L}}$ holds iff $[\![LT_1]\!]_{\mathcal{L}} \geq [\![LT_2]\!]_{\mathcal{L}}$
$[\![\forall x\colon \tau.\ P]\!]_{\mathcal{L}}$ holds iff for all value v of type τ, $[\![P]\!]_{\mathcal{L}[x \mapsto v]}$ holds
$[\![\exists x\colon \tau.\ P]\!]_{\mathcal{L}}$ holds iff there is a value v of type τ such that $[\![P]\!]_{\mathcal{L}[x \mapsto v]}$ holds
$[\![\forall \textbf{region}\ r\colon \mathcal{C}.\ P]\!]_{\mathcal{L}}$ holds iff for all logic region h, $[\![P]\!]_{\mathcal{L}[\textsf{here} \mapsto \mathcal{L}(\textsf{here})[r \mapsto h]\ ]}$ holds
$[\![\exists \textbf{region}\ r\colon \mathcal{C}.\ P]\!]_{\mathcal{L}}$ holds iff there is h such that $[\![P]\!]_{\mathcal{L}[\textsf{here} \mapsto \mathcal{L}(\textsf{here})[r \mapsto h]\ ]}$ holds
$[\![LT \in RT]\!]_{\mathcal{L}}$ holds iff $[\![LT]\!]_{\mathcal{L}}$ is in $[\![RT]\!]_{\mathcal{L}}(contents)$

Figure 4.14: Predicate Semantics (Intuitive Model)

$[\![c]\!]_{\mathcal{H}} = c$
$[\![e_1\ op\ e_2]\!]_{\mathcal{H}} = [\![e_1]\!]_{\mathcal{H}}\ op\ [\![e_2]\!]_{\mathcal{H}}$
$[\![x]\!]_{\mathcal{H}} = \mathcal{H}(x)$
$[\![x.f]\!]_{\mathcal{H}} = \mathcal{H}(\mathcal{H}(x))(f)$
$[\![f\ arg_1\ \cdots\ arg_n]\!]_{\mathcal{H}} = [\![f]\!]\ ([\![arg_1]\!]_{\mathcal{H}}, \cdots, [\![arg_n]\!]_{\mathcal{H}})$

Semantics of arguments *arg*:
$[\![\ [\rho]\ ]\!]_{\mathcal{H}} = lor(\mathcal{H}, \mathcal{H}(\rho))$
$[\![\ (e)\ ]\!]_{\mathcal{H}} = [\![e]\!]_{\mathcal{H}}$

Figure 4.15: Expression Semantics (Intuitive Model)

Constants and operations are evaluated as themselves. The evaluation of logic functions in expressions is the same as the one for terms, except that the model for regions is constructed from program regions ρ. To evaluate a variable is just to read its value in the heap. To evaluate $x.f$, first we read variable x to obtain a location p, then we read at location p to obtain an object, and finally we read the f field of the object.

Statement Execution We now give semantics to statements. We denote by:

$$\mathcal{H},\ s \Longrightarrow \mathcal{H}'$$

the execution of statement s which transforms heap $\mathcal{H}$ into heap $\mathcal{H}'$. We denote by:

$$\mathcal{H},\ s \Longrightarrow \infty$$

the fact that the execution of statement s diverges.

We write:

$$\mathcal{H},\ s_1;\cdots;\ s_n \Longrightarrow \mathcal{H}'$$

if, and only if for all i:

$$\mathcal{H}_i,\ s_i \Longrightarrow \mathcal{H}_{i+1}$$

and $\mathcal{H} = \mathcal{H}_1$ and $\mathcal{H}' = \mathcal{H}_{n+1}$. We write:

$$\mathcal{H},\ s_1;\cdots;\ s_n \Longrightarrow \infty$$

if, and only if there is some i such that:

$$\mathcal{H},\ s_1;\cdots;\ s_i \Longrightarrow \mathcal{H}'$$

and:

$$\mathcal{H}',\ s_{i+1} \Longrightarrow \infty$$

If f is a partial function (a heap $\mathcal{H}$, for instance) we denote by $f[\rho \mapsto \mathcal{P}]$ the function g defined as follows. If ρ is a region variable r, then $g(r) = \mathcal{P}$ and for all $x \neq r$, $g(x) = f(x)$. This corresponds to the already-defined $f[r \mapsto \mathcal{P}]$. Else ρ is an owned region $x.r$, and then $g(x) = g(x)[r \mapsto \mathcal{P}]$ and for all $y \neq x$, $g(y) = f(y)$.

We denote $\mathcal{H}(x.r) = \mathcal{H}(\mathcal{H}(x))(r)$. In particular, this allows us to write $\mathcal{H}(\rho)$ whether ρ is a region variable or an owned region.

Figures 4.16 and 4.17 give semantics for each statement of the Capucine language. Figure 4.16 defines the semantics of statements which terminate. The definition is inductive. Figure 4.17 defines the semantics of statements which diverge, which may happen due to the presence of recursive functions. The definition is coinductive. Programs which neither terminate nor diverge are programs which block because of an error.

The let-binding statement inserts a new variable into the heap. It uses the semantics of expressions defined in Figure 4.15 to compute the value given to the variable.

The let-region statement simply adds the new region in the heap, initially empty.

The unpack and weakening statements do nothing and always reduce. They are only used as typing annotations.

The focus statement first computes the location p stored in variable x. Then it sets region σ to a singleton region containing only p. The active region of p is changed to σ.

$\mathcal{H}$, **let** $x = e \Longrightarrow \mathcal{H}[x \mapsto [\![e]\!]_{\mathcal{H}}]$
$\mathcal{H}$, **let region** r: $\mathcal{C} \Longrightarrow \mathcal{H}[r \mapsto \emptyset]$
$\mathcal{H}$, **unpack** $x \Longrightarrow \mathcal{H}$
$\mathcal{H}$, **weaken empty** $\rho \Longrightarrow \mathcal{H}$
$\mathcal{H}$, **weaken single** $\rho \Longrightarrow \mathcal{H}$
$\mathcal{H}$, **focus** x: ρ **as** $\sigma \Longrightarrow \mathcal{H}[\sigma \mapsto \{\mathcal{H}(x)\}][\mathcal{H}(x)? \mapsto \sigma]$
$\mathcal{H}$, **unfocus** x: σ **as** $\rho \Longrightarrow \mathcal{H}[\mathcal{H}(x)? \mapsto \rho]$
$\mathcal{H}$, $x.f \leftarrow e \Longrightarrow \mathcal{H}[\mathcal{H}(x) \mapsto \mathcal{H}(x)[f \mapsto [\![e]\!]_{\mathcal{H}}]\ ]$
$\mathcal{H}$, **adopt** x: σ **as** $\rho \Longrightarrow \mathcal{H}[\rho \mapsto \mathcal{H}(\rho) \cup \{\mathcal{H}(x)\}][\mathcal{H}(x)? \mapsto \rho]$

$\mathcal{H}$, **if** e **then** S **else** $S' \Longrightarrow \mathcal{H}'$
with $\mathcal{H}$, $S \Longrightarrow \mathcal{H}'$ if $[\![e]\!]_{\mathcal{H}} = \mathit{true}$ and $\mathcal{H}$, $S' \Longrightarrow \mathcal{H}'$ if $[\![e]\!]_{\mathcal{H}} = \mathit{false}$.

$\mathcal{H}$, **let** $x = f\ \mathit{arg}_1 \cdots \mathit{arg}_l \Longrightarrow \mathcal{H}_4[x \mapsto [\![e]\!]_{\mathcal{H}_3}]$
where:
$\mathit{arg}_1 \cdots \mathit{arg}_l$ restricted to region arguments is $[\rho_1] \cdots [\rho_n]$,
$\mathit{arg}_1 \cdots \mathit{arg}_l$ restricted to regular arguments is $(e_1) \cdots (e_m)$,
f was declared with region arguments $r_1, \cdots, r_n$ in this order,
f was declared with regular arguments $x_1, \cdots, x_m$ in this order,
f was declared with body $s_1; \cdots; s_n$; **return** e,
f was declared with pre-condition P and post-condition Q,
$\mathcal{H}_1 = \mathcal{H}[x_1 \mapsto [\![e_1]\!]_{\mathcal{H}}] \cdots [x_m \mapsto [\![e_m]\!]_{\mathcal{H}}]$,
$\mathcal{H}_2 = \mathcal{H}_1[r_1 \mapsto \mathcal{H}(\rho_1)] \cdots [r_n \mapsto \mathcal{H}(\rho_n)][\rho_1? \mapsto r_1] \cdots [\rho_n? \mapsto r_n]$,
$\mathcal{H}_2$, $s_1; \cdots; s_l \Longrightarrow \mathcal{H}_3$,
$\mathcal{H}_4 = \mathcal{H}_3[\rho_1 \mapsto \mathcal{H}_3(r_1)] \cdots [\rho_n \mapsto \mathcal{H}_3(r_n)][r_1? \mapsto \rho_1] \cdots [r_n? \mapsto \rho_n]$,
and only if $[\![P]\!]_{loh(\mathcal{H}_2)}$ and $[\![Q]\!]_{loh(\mathcal{H}_3[\textsf{result} \mapsto [\![e]\!]_{\mathcal{H}_3}])}$ hold.

$\mathcal{H}$, **let** $x =$ **new** $\mathcal{C}$ $[\rho] \Longrightarrow \mathcal{H}[p \mapsto o][x \mapsto p][\rho \mapsto \{p\}][p? \mapsto \rho][\mathit{next} \mapsto p{+}1]$
where $p = \mathcal{H}(\mathit{next})$ and o is an object associating $\emptyset$ to every region owned by $\mathcal{C}$.

$\mathcal{H}$, **pack** x: $\mathcal{C} \Longrightarrow \mathcal{H}$
if $[\![\mathit{Inv}_{\mathcal{C}}(\mathcal{H}(x))]\!]_{loh(\mathcal{H})}$ holds.

$\mathcal{H}$, **assert** $P \Longrightarrow \mathcal{H}$
if $[\![P]\!]_{loh(\mathcal{H})}$ holds.

$\mathcal{H}$, **label** $L \Longrightarrow \mathcal{H}[L \mapsto \mathit{cur}]$
where cur is $loh(\mathcal{H})$ restricted to labels.

Figure 4.16: Statement Semantics (Intuitive Model)

$\mathcal{H}$, **if** e **then** S **else** $S' \Longrightarrow \infty$
with $\mathcal{H}$, $S \Longrightarrow \infty$ if $[\![e]\!]_{\mathcal{H}} = \mathit{true}$ and $\mathcal{H}$, $S' \Longrightarrow \infty$ if $[\![e]\!]_{\mathcal{H}} = \mathit{false}$.

$\mathcal{H}$, **let** $x = f\ \mathit{arg}_1 \cdots \mathit{arg}_l \Longrightarrow \infty$
where:
$\mathit{arg}_1 \cdots \mathit{arg}_l$ restricted to region arguments is $[\rho_1] \cdots [\rho_n]$,
$\mathit{arg}_1 \cdots \mathit{arg}_l$ restricted to regular arguments is $(e_1) \cdots (e_m)$,
f was declared with region arguments $r_1, \cdots, r_n$ in this order,
f was declared with regular arguments $x_1, \cdots, x_m$ in this order,
f was declared with body $s_1; \cdots; s_n$; **return** e,
f was declared with pre-condition P and post-condition Q,
$\mathcal{H}_1 = \mathcal{H}[x_1 \mapsto [\![e_1]\!]_{\mathcal{H}}] \cdots [x_m \mapsto [\![e_m]\!]_{\mathcal{H}}]$,
$\mathcal{H}_2 = \mathcal{H}_1[r_1 \mapsto \mathcal{H}(\rho_1)] \cdots [r_n \mapsto \mathcal{H}(\rho_n)][\rho_1? \mapsto r_1] \cdots [\rho_n? \mapsto r_n]$,
$\mathcal{H}_2$, $s_1; \cdots; s_l \Longrightarrow \infty$

Figure 4.17: Statement Semantics: Divergence (Intuitive Model)

The unfocus statement restores the active region of x to ρ.

The assignment statement first reads the location stored in variable x. Then it modifies the object stored at this location to change the f field.

The adoption statement adds location p stored in variable x to the set of pointers denoted by region ρ. The active region of p becomes ρ.

The if statement is only defined if the condition evaluates to *true* or *false*. The block which is executed depends on this value.

The call statement semantics is a little tricky. Because the body of function f is defined using formal parameters $r_1, \cdots, r_n$ and $x_1, \cdots, x_n$, we build a heap $\mathcal{H}_2$ in which these variables make sense. Their value is the value which was given as argument to the call. This is simple for variables $x_1, \cdots, x_n$. For regions $r_1, \cdots, r_n$ we also have to modify the active regions of pointers. To this end we define $\mathcal{H}[\rho? \mapsto r]$ as $f[p_1? \mapsto r] \cdots [p_n? \mapsto r]$, if $\{p \mid \mathcal{H}(p?) = \rho\} = \{p_1, \cdots, p_n\}$. Note that we also require the pre-condition to hold in $\mathcal{H}_2$. After the call we obtain heap $\mathcal{H}_3$, and we have to move pointers back to their original region. We thus obtain $\mathcal{H}_4$, which is almost the final heap. We then return heap $\mathcal{H}_4$ with additional variable x whose value is the evaluation of the return expression e in heap $\mathcal{H}_3$. Note that we also assume that no variable capture occurs.

The allocation statement takes the next free location p in the heap and puts it in newly-bound variable x. Then $\mathcal{H}(\mathit{next})$ becomes $p{+}1$. An object o is allocated in $\mathcal{H}(p)$, with empty sets for all owned regions. The active region of p is ρ, and ρ is set to be the singleton set $\{p\}$.

The pack statement does not change the heap, but it only reduces if the invariant of the pointer being packed holds. To this end we define $[\![\mathit{Inv}_{\mathcal{C}}(p)]\!]_{\mathcal{L}}$, the invariant predicate of class $\mathcal{C}$ applied to location p. It is equivalent to $[\![P]\!]_{\mathcal{L}'}$ where P is the invariant predicate defined in the declaration of $\mathcal{C}$, and $\mathcal{L}'$ is a logic environment such that: for each field f of $\mathcal{C}$, we have $\mathcal{L}'(f) = \mathcal{L}(p)(f)$ (field name f is viewed as a variable name) and for all region r of $\mathcal{C}$, we have $\mathcal{L}'(r) = \mathcal{L}(p)(r)$.

The assert statement does not change the heap, but it only reduces if its predicate holds in the current heap.

4.4 Coherence Preservation

Type of Values We say that type τ is a pointer type of class $\mathcal{C}$ in environment Γ and heap $\mathcal{H}$ if τ is $[r]$ and r: $\mathcal{C}$ is in Γ, or if τ is $[x.r]$, x: τ' is in Γ, τ' is a pointer type of class $\mathcal{C}'$, and r is an owned region of class $\mathcal{C}$ in the class definition of $\mathcal{C}'$.

We say that p is a location of class $\mathcal{C}$ in heap $\mathcal{H}$ if $\mathcal{H}(p)(f)$ is defined for each field f defined in $\mathcal{C}$ and is a value of the type of f (see below), and $\mathcal{H}(p)(r)$ is defined for each region r defined in $\mathcal{C}$.

We say that v is a value of type τ in environment Γ and heap $\mathcal{H}$ if:

- τ is *int* and v is an integer;
- or τ is *bool* and v is *true* or *false*;
- or τ is *unit* and v is the unit constant ();
- or τ is a user type and v is a value of this type;
- or τ is a pointer type of class $\mathcal{C}$ and v is a location of class $\mathcal{C}$.

These definitions coinductive. An inductive definition would restrict ourself to programs which do not manipulate data structures with cycles.

Coherence We say that location p is *closed* for class $\mathcal{C}$ in heap $\mathcal{H}$ if, and only if the invariant of p holds in $\mathcal{H}$, i.e.:

$$[\![Inv_{\mathcal{C}}(p)]\!]_{loh(\mathcal{H})}$$

and all owned locations of p are closed, i.e. for all region r, for all q in $\mathcal{H}(p)(r)$ (if defined), then q is closed in $\mathcal{H}$. Moreover, we require that if r is defined as a singleton region in $\mathcal{C}$, then $\mathcal{H}(p)(r)$ is a singleton set.

Heap $\mathcal{H}$ is *coherent* with respect to permissions $\overline{\Sigma}$ and typing environment Γ if, and only if:

1. **(unicity of permissions)** for all region ρ, there is at most one permission of the form $\rho^{\emptyset}$, ρ°, $\rho^{\times}$, ρ^{G} or $\sigma \multimap \rho$ in $\overline{\Sigma}$;
2. **(empty permissions)** if $\rho^{\emptyset}$ is in $\overline{\Sigma}$, then $\mathcal{H}(\rho) = \emptyset$;
3. **(singleton permissions)** if $\rho^{\circ}\{\overline{f}\}$ or $\rho^{\times}$ is in $\overline{\Sigma}$, then there is p such that $\mathcal{H}(\rho) = \{p\}$;
4. **(closed permissions)** if $\rho^{\times}$ or ρ^{G} is in $\overline{\Sigma}$ and $\Gamma \vdash \rho$: $\mathcal{C}$, then all p in $\mathcal{H}(\rho)$ is closed for $\mathcal{C}$ in $\mathcal{H}$;
5. **(focus permissions)** if $\sigma \multimap \rho$ is in $\overline{\Sigma}$ and $\Gamma \vdash \rho$: $\mathcal{C}$, then all p in $\mathcal{H}(\rho) - \mathcal{H}(\sigma)$ is closed for $\mathcal{C}$ in $\mathcal{H}$;
6. **(type of values)** for all expression e such that Γ, $\overline{\Sigma} \vdash e$: τ, then $[\![e]\!]_{\mathcal{H}}$ is defined and is a value of type τ;
7. **(regions are defined)** for all region ρ such that $\Gamma \vdash \rho$: $\mathcal{C}$, then $\mathcal{H}(\rho)$ is defined;
8. **(active regions)** for all location p, for all region ρ, if $p \in \mathcal{H}(\rho)$ and either ρ°, $\rho^{\times}$, ρ^{G} is in $\overline{\Sigma}$ or $\sigma \multimap \rho$ is in $\overline{\Sigma}$ and $p \notin \mathcal{H}(\sigma)$, then $\mathcal{H}(p?) = \rho$;

9. **(labels are defined)** for all L: R in Γ, for all r in R, $\mathcal{H}(L)(r)$ is defined.

Item 1 is required for consistency; having both $\rho^{\emptyset}$ and $\rho^{\times}$, for instance, would result in ρ being both empty and not empty. Items 2, 3, 4 and 5 state the meaning of each permission. Items 4 and 5 are especially important, as they ensure that invariants hold when they are supposed to. In fact, they are the items which matter the most from an external point of vue; other items strengthen the coherence predicate to be able to prove items 4 and 5. Item 6 mainly states that variables and locations of the heap are of the correct type. It also states that all well-typed expressions can actually be evaluated. In the same fashion, item 7 states that all well-typed regions make sense in the heap. Item 8 gives the meaning of active regions. Item 9 ensures that labels in the environment are in the heap domain.

The following theorem states that executing a program in a coherent heap results in a coherent heap.

Theorem 1 (Coherence Preservation) For all heap $\mathcal{H}$ coherent with respect to an environment Γ and some permissions $\overline{\Sigma}$, for all statement s, if

$$\Gamma \vdash \{\overline{\Sigma}\}\ s\ \{\overline{\Sigma}'\}, \Gamma'$$

and if

$$\mathcal{H}, s \Longrightarrow \mathcal{H}'$$

then $\mathcal{H}'$ is coherent with respect to Γ' and $\overline{\Sigma}'$.

Proof. By induction on the derivation of $\mathcal{H}, s \Longrightarrow \mathcal{H}'$. We first note that all statements preserve item 1 of coherence. We will implicitely use this hypothesis each time a region or a location is modified in the heap in the following fashion: if we show item 2, 3, 4 or 5 for a given region ρ, we do not have to show any other of these items for region ρ, except for $\rho^{\times}$ which is shared by 3 and 4.

- **let** $x = e$

  Item 6 of coherence is preserved because the only new variable in the environment is x of the type τ of expression e, and thus the only new well-typed expressions are x, and other expressions mentionning x. The value given to x in the heap is the value obtained by evaluating e. Item 6 of coherence of $\mathcal{H}$ states that this value has type τ. From there we prove item 6 on all new well-typed expressions, i.e. those who mention x, by induction on these expressions. Other items of coherence are trivial.

- **if** e **then** S_1 **else** S_2

  We use the induction hypothesis on either S_1 or S_2, depending on $[\![e]\!]_{\mathcal{H}}$.

- **let** $x = f\ \overline{arg}$

  Recall typing rule SCall:

$$\frac{\begin{array}{c} f \textbf{ consumes } \overline{\Sigma}_1 \qquad f \textbf{ produces } \overline{\Sigma}_2 \qquad f : param_1 \cdots param_n \to \tau \\ \text{for all } i : \Gamma, \overline{\Sigma}, \sigma \vdash arg_i : param_i \qquad separated(arg_1, \cdots, arg_n) \end{array}}{\Gamma \vdash \{\overline{\Sigma}, \sigma(\overline{\Sigma}_1)\}\ \textbf{let}\ x = f\ arg_1 \cdots arg_n\ \{\overline{\Sigma}, \sigma(\overline{\Sigma}_2)\}, (\Gamma, x : \sigma(\tau))}\ \textsc{SCall}$$

First we show that $\mathcal{H}_2$ is coherent with respect to the internal environment Γ_2 and permissions $\overline{\Sigma}_2$ with which the body of function f was typed. Then we apply the induction hypothesis and obtain that $\mathcal{H}_3$ is coherent. Then we show that $\mathcal{H}'$ is coherent.

Heap $\mathcal{H}_1$ is obtained from $\mathcal{H}$ by adding variables $\overline{x}$ with respective values $\overline{[\![e]\!]_{\mathcal{H}}}$. Typing ensures that the type of these values is the formal type of the regular arguments $\overline{x}$ of f, and thus item 6 of coherence of $\mathcal{H}_1$ with respect to Γ' holds.

Heap $\mathcal{H}_2$ is obtained from $\mathcal{H}_1$ by adding regions $\overline{r}$ with respective sets $\overline{\mathcal{H}(\rho)}$. Typing of permissions ensures that permissions $\overline{\Sigma}_1$ are available. Thus items 2, 3, 4 and 5 of coherence hold too, and $\mathcal{H}_2$ is coherent. Thus $\mathcal{H}_3$ is coherent.

We use a similar reasoning from produced permissions to show that $\mathcal{H}_4$ is coherent after transferring regions back. Heap $\mathcal{H}'$ is obtained from $\mathcal{H}_4$ by adding variable x of the correct type, thus preserving coherence.

- **let region** r: $\mathcal{C}$

  The only items of coherence which are not straightforward are items 2, 6 and 7, as there is new permission $r^{\emptyset}$ and new region r in the environment. Item 2 holds by definition of $\Longrightarrow$. The only new well-typed expressions are logic function applications using r, which are of the correct type. So item 6 of coherence is preserved. By definition of the reduction rule, $\mathcal{H}'(r)$ is defined, so item 7 is preserved.

- **let** x = **new** $\mathcal{C}$ $[\rho]$

  Item 6 holds as x is a pointer of the correct type. Expressions $x.f$ is not well-typed as permission on ρ states that no field is initialized. Permission $\rho^{\emptyset}$ is consumed and replaced by ρ° and empty permissions on owned regions of x. Region ρ is indeed singleton after the allocation, and owned regions are indeed empty. Moreover, $\mathcal{H}(x.r)$ is defined for each owned region r, so item 7 of coherence is preserved. Let $p = \mathcal{H}(next)$. Item 8 holds as we have $p \in \mathcal{H}'(\rho)$ along with $\mathcal{H}'(p?) = \rho$ and ρ°.

- **focus** x: ρ **as** σ

  Permissions ρ^{G} and $\sigma^{\emptyset}$ are consumed and replaced by $\sigma \multimap \rho$ and $\sigma^{\times}$. The location put into σ was in ρ; we use item 4 to show that it was thus closed, and is still closed in σ. Region σ is singleton after the operation by definition of $\Longrightarrow$. Thus items 3 and 4 are preserved. The active region of the focused pointer is changed from σ to ρ, to preserve item 8.

  Variable x has changed type from $[\rho]$ to $[\sigma]$, but it indeed contains a location of σ, so item 6 is preserved.

- $x.f \leftarrow e$

  This operation does not break items 4 and 5 of coherence. Indeed, assume that the invariant of a pointer p is broken by this assignment. Then this invariant depends on $x.f$. The restriction on invariants requires that x is (transitively) owned by p, which contradicts item 4 of coherence.

  Item 6 holds as typing ensures that $[\![e]\!]_{\mathcal{H}}$ has the type of f, and the only new well-typed expression is $x.f$ (permissions state that f is now initialized if it was not).

- **adopt** x: σ **as** ρ

Permission ρ^G is produced. We thus have to show that the new pointer of ρ, i.e. x, is closed. To this end we use the fact that $\sigma^\times$ was available before the operation. Thus item 4 is preserved. The active region of the focused pointer is changed from σ to ρ, to preserves item 8.

Variable x has changed type from $[\sigma]$ to $[\rho]$, but it indeed contains a location which is now in ρ, so item 6 is preserved.

- **unfocus** x: σ **as** ρ

  Permission ρ^G is produced while $\sigma \multimap \rho$ was available before. We thus have to show that the pointer of σ, i.e. x, is closed. To this end we use the fact that $\sigma^\times$ was available before the operation. Thus item 4 is preserved. The active region of the focused pointer is changed from σ to ρ, to preserves item 8.

  Variable x has changed type from $[\sigma]$ to $[\rho]$, but it indeed contains a location which is now in ρ, so item 6 is preserved.

  Note how close this reasoning is to the one we used above for adoption. Indeed, **adopt** and **unfocus** are very similar operations.

- **pack** x: $\mathcal{C}$

  Permission $\rho^\times$ is produced, where ρ is the region of x. We thus have to show that x is closed. By definition of $\Longrightarrow$, the invariant of x holds. Moreover, typing ensures that closed permissions on all owned regions of x are consumed. So we use item 4 of coherence to show that all (transitively) owned locations are closed, and thus that x is closed itself.

- **unpack** x

  Permission ρ° is produced, and $\rho^\times$ is consumed so ρ is indeed singleton. Permissions on owned regions are produced. We use item 4 to show that all owned locations of x are closed and thus that item 4 is preserved. We also use item 4 to show that all singleton owned locations are indeed singletons, and thus that item 3 is preserved.

- **weaken empty** ρ

  An empty region has no pointer, so item 4 trivially holds.

- **weaken single** ρ

  Permission $\rho^\times$ is consumed and replaced by ρ^G, which preserves item 4 of coherence.

- **assert** P

  Heap, environment and permissions are not modified by this operation.

- **label** L

  We add L: R in Γ, so we have to check item 9 of coherence. The typing rule requires all region names r in R to be in Γ, so from item 7 of coherence we obtain that $\mathcal{H}(r)$ is defined. So $loh(\mathcal{H})(r)$ is defined as well, and thus $\mathcal{H}'(L)(r)$ is defined.

□

4.5 Separated Memory Model

4.5.1 Separated Operational Semantics

We now give Capucine a model which is less standard, but which is closer to the Capucine-to-Why translation which will be introduced in Chapter 5. The idea is that locations do not belong to the heap but to regions. In other words, instead of writing $\mathcal{H}(p)$, we write $\mathcal{H}_s(r)(p)$ where r is the region of location p. Moreover, if p owns a region s, we can obtain its object by writing $\mathcal{H}_s(r)(p)(s)$. If location q belongs to s, we can write $\mathcal{H}_s(r)(p)(s)(q)$ to obtain the object of q.

Model We define a *region function* g as a function from locations p to *separated objects* o. A separated object is a function from field names f to values, and from owned region names r to region functions.

We define the *separated heap* $\mathcal{H}_s$ as a function:

- $\mathcal{H}_s(x)$ returns the value of variable x;
- $\mathcal{H}_s(r)$ returns the region function of region name r;
- $\mathcal{H}_s(p?)$ returns the active region of pointer p;
- $\mathcal{H}_s(next)$ returns the next free location to be allocated;
- $\mathcal{H}_s(L)(r)$ returns a region function.

$\mathcal{H}_s(x)$, $\mathcal{H}_s(p?)$ and $\mathcal{H}_s(next)$ are the same as $\mathcal{H}(x)$, $\mathcal{H}(p?)$ and $\mathcal{H}(next)$, respectively. The only change is thus that $\mathcal{H}(p)$ is moved into region functions $\mathcal{H}_s(r)$.

We then define $\mathcal{H}_s(p)$ as follows. If $\mathcal{H}_s(p?) = r$, where r is a region variable name, then we define $\mathcal{H}_s(p) = \mathcal{H}_s(r)(p)$. If $\mathcal{H}_s(p?) = x.r$ then we define $\mathcal{H}_s(p) = \mathcal{H}_s(\mathcal{H}_s(x))(r)(p)$. This is a recursive definition, and thus $\mathcal{H}_s(p)$ is only defined if the path described by $\mathcal{H}_s(p?)$ is defined and finite.

We define $\mathcal{H}_s[p \mapsto o]$ as follows. If $\mathcal{H}_s(p?) = r$ then:

$$\mathcal{H}_s[p \mapsto o] = \mathcal{H}_s[r \mapsto \mathcal{H}_s(r)[p \mapsto o]\ ]$$

If $\mathcal{H}_s(p?) = x.r$ then:

$$\mathcal{H}_s[p \mapsto o] = \mathcal{H}_s[\mathcal{H}_s(x) \mapsto \mathcal{H}_s(\mathcal{H}_s(x))[r \mapsto \mathcal{H}_s(\mathcal{H}_s(x))(r)[p \mapsto o]\ ]\ ]$$

It is once again a recursive definition.

We also define:

$$\mathcal{H}_s(p.r) = \mathcal{H}_s(p)(r)$$

$$\mathcal{H}_s(x.r) = \mathcal{H}_s(\mathcal{H}_s(x))(r)$$

as well as the corresponding replacements:

$$\mathcal{H}_s[p.r \mapsto g] = \mathcal{H}_s[p \mapsto \mathcal{H}_s(p)[r \mapsto g]\ ]$$

$$\mathcal{H}_s[x.r \mapsto g] = \mathcal{H}_s[\mathcal{H}_s(x) \mapsto \mathcal{H}_s(\mathcal{H}_s(x))[r \mapsto g]\ ]$$

Logic Model The separated logic model is very close to the separated program model. A *separated logic environment* $\mathcal{L}_s$ is a function:

- $\mathcal{L}_s(L)(r)$ returns a region function;
- $\mathcal{L}_s(x)$ returns the value of variable x.

$\mathcal{L}_s$ thus looks like a projection of $\mathcal{H}_s$ on regions and variables.

From Programs to Logic We define $lor_s(g)$, where g is a separated region function, as the logic region corresponding to g. By logic region we mean *intuitive* logic region as introduced in Section 4.3, not separated logic region. Formally, $lor_s(g)$ is defined as the smallest function such that:

$$lor_s(g)(contents) = Dom(g)$$

and for all $n \geq 0$, for all $p_1, \cdots, p_n$, for all $r_1, \cdots, r_n$, for all p, for all field name f, for all owned region name s, if $g(p_1)(r_1)\cdots(p_n)(r_n)(p)(f)$ is defined then:

$$lor_s(p)(f) = g(p_1)(r_1)\cdots(p_n)(r_n)(p)(f)$$

and if $g(p_1)(r_1)\cdots(p_n)(r_n)(p)(r)$ is defined then:

$$lor_s(p)(s) = Dom(g(p_1)(r_1)\cdots(p_n)(r_n)(p)(s))$$

If a location appears twice in g, then $lor_s(g)$ is not defined. For instance, if $p \in Dom(g)$ and $p \in Dom(g(p)(s))$, $lor_s(g)$ is not defined.

We define $lor_s{}'(g, \mathcal{H}_s)$ where g is a separated region function, as the logic region corresponding to g in $\mathcal{H}_s$. The difference with lor_s is that we take active regions of pointers in $\mathcal{H}_s$ into account. Thus:

$$Dom(lor_s{}'(g, \mathcal{H}_s)) = Dom(lor_s(g))$$

and for all p in $Dom(lor_s(g))$:

$$lor_s{}'(g, \mathcal{H}_s)(p) = \mathcal{H}_s(p)$$

Remember that $\mathcal{H}_s(p)$ is actually a notation which uses active regions of pointers. Finally we of course have:

$$lor_s{}'(g, \mathcal{H}_s)(contents) = lor_s(g)(contents) = Dom(g)$$

We define loh_s as a function from separated heaps $\mathcal{H}_s$ to separated logic environments $\mathcal{L}_s$. The domain of loh_s is the domain of $\mathcal{H}_s$ restricted to variables x and region names r. For each region name r in $Dom(\mathcal{H}_s)$, we define:

$$loh_s(\mathcal{H}_s)(\textsf{here})(r) = lor_s{}'(\mathcal{H}_s(r), \mathcal{H}_s)$$

For each label L in $Dom(\mathcal{H}_s)$, we define:

$$loh_s(\mathcal{H}_s)(L) = \mathcal{H}_s(L)$$

For each variable name x in $Dom(\mathcal{H}_s)$, we define:

$$loh_s(\mathcal{H}_s)(x) = \mathcal{H}_s(x)$$

$(\!|c|\!)_{\mathcal{L}_s} = c$
$(\!|LT_1 \ termop \ LT_2|\!)_{\mathcal{L}_s} = (\!|LT_1|\!)_{\mathcal{L}_s} \ termop \ (\!|LT_2|\!)_{\mathcal{L}_s}$
$(\!|x|\!)_{\mathcal{L}_s} = \mathcal{L}_s(x)$
$(\!|get(RT,\ LT,\ f)|\!)_{\mathcal{L}_s} = (\!|RT|\!)_{\mathcal{L}_s}((\!|LT|\!)_{\mathcal{L}_s})(f)$
$(\!|f \ larg_1 \cdots larg_n|\!)_{\mathcal{L}_s} = [\![f]\!]\ ((\!|larg_1|\!)_{\mathcal{L}_s},\ \cdots,\ (\!|larg_n|\!)_{\mathcal{L}_s})$

Semantics of logic arguments *larg*:
$(\!|\ [RT]\ |\!)_{\mathcal{L}_s} = (\!|RT|\!)_{\mathcal{L}_s}$
$(\!|\ (LT)\ |\!)_{\mathcal{L}_s} = (\!|LT|\!)_{\mathcal{L}_s}$

Figure 4.18: Term Semantics (Separated Model)

$(\!|r@L|\!)_{\mathcal{L}_s} = \mathcal{L}_s(L)(r)$
$(\!|get(RT,\ LT,\ s)|\!)_{\mathcal{L}_s} = lor_s((\!|RT|\!)_{\mathcal{L}_s}((\!|LT|\!)_{\mathcal{L}_s})(s))$

Figure 4.19: Region Term Semantics (Separated Model)

Term Evaluation We denote by:

$$(\!|LT|\!)_{\mathcal{L}_s}$$

the evaluation of term LT in separated logic environment $\mathcal{L}_s$. It returns a value v. Figure 4.18 gives the definition of $(\!|LT|\!)_{\mathcal{L}_s}$.

We denote by:

$$(\!|RT|\!)_{\mathcal{L}_s}$$

the evaluation of region term RT in $\mathcal{L}_s$. It returns a logic region. Figure 4.19 gives the definition of $(\!|RT|\!)_{\mathcal{L}_s}$.

Term evaluation in the separated model is very similar to term evaluation in the intuitive model. In fact, the only difference is evaluation of region term $get(RT,\ LT,\ s)$. Note that we do not assume another model $(\!|f|\!)$ for logic functions and predicates; instead we reuse the existing one $[\![f]\!]$. This ensures that we obtain the same result when applying f in the intuitive model and in the separated model.

Predicate Evaluation We denote by:

$$(\!|P|\!)_{\mathcal{L}_s}$$

the evaluation of predicate P in separated logic environment $\mathcal{L}_s$. It is similar to predicate evaluation in an intuitive logic environment $\mathcal{L}$, except that we use the separated term and region evaluation defined above. Thus the definition of $(\!|P|\!)_{\mathcal{L}_s}$ is exactly the same as the definition of $[\![P]\!]_{\mathcal{L}}$ in Figure 4.14, except that most $[\![.]\!]$ are replaced by $(\!|.|\!)$ and $\mathcal{L}$ by $\mathcal{L}_s$. We only keep $[\![p]\!]$ instead of $(\!|p|\!)$ in predicate application, just as we did for terms.

Expression Evaluation We denote by:

$$(\!|e|\!)_{\mathcal{H}_s}$$

the evaluation of expression e in separated heap $\mathcal{H}_s$. Figure 4.20 gives the definition of $(\!|e|\!)_{\mathcal{H}_s}$. It is similar to expression evaluation in an intuitive heap $\mathcal{H}$ as defined in 4.15, except that the definition of $\mathcal{H}_s(p)$ is not the same as the definition of $\mathcal{H}(p)$, and that the definition of $(\!|\ [\rho]\ |\!)_{\mathcal{H}}$ is not the same as the definition of $[\![\ [\rho]\ ]\!]_{\mathcal{H}}$.

$(\!|c|\!)_{\mathcal{H}_s} = c$
$(\!|e_1 \ op \ e_2|\!)_{\mathcal{H}_s} = (\!|e_1|\!)_{\mathcal{H}_s} \ op \ (\!|e_2|\!)_{\mathcal{H}_s}$
$(\!|x|\!)_{\mathcal{H}_s} = \mathcal{H}_s(x)$
$(\!|x.f|\!)_{\mathcal{H}_s} = \mathcal{H}_s(\mathcal{H}_s(x))(f)$
$(\!|f \ arg_1 \cdots arg_n|\!)_{\mathcal{H}_s} = [\![f]\!] \ ((\!|arg_1|\!)_{\mathcal{H}_s}, \cdots, (\!|arg_n|\!)_{\mathcal{H}_s})$

Semantics of arguments *arg*:
$(\!| \ [\rho] \ |\!)_{\mathcal{H}_s} = lor_s'(\mathcal{H}_s(\rho), \mathcal{H}_s)$
$(\!| \ (e) \ |\!)_{\mathcal{H}_s} = (\!|e|\!)_{\mathcal{H}_s}$

Figure 4.20: Expression Semantics (Separated Model)

Statement Execution We denote by:

$$\mathcal{H}_s, \ s \Longrightarrow \mathcal{H}_s'$$

the execution of statement s which transforms separated heap $\mathcal{H}_s$ into separated heap $\mathcal{H}_s'$. We denote by:

$$\mathcal{H}_s, \ s \Longrightarrow \infty$$

the fact that the execution of statement s diverges. In a similar fashion as for the intuitive semantics, we write:

$$\mathcal{H}_s, \ s_1; \cdots; \ s_n \Longrightarrow \mathcal{H}_s'$$

and:

$$\mathcal{H}_s, \ s_1; \cdots; \ s_n \Longrightarrow \infty$$

the semantics for sequences.

Figures 4.21 and 4.22 give semantics for each statement of the Capucine language in our separated model. The definitions are respectively inductive and coinductive.

We denote by $\emptyset$ the function of empty domain. It is used by the **let region** statement, which adds an empty region, i.e. a function of empty domain, to the heap.

Most definitions are unchanged from the intuitive model, or at least looks unchanged thanks to notations such as $\mathcal{H}_s[p \mapsto \cdots]$. This is the case in particular for assignment and function calls. Definition for function calls uses notation $f[\rho? \mapsto \cdots]$ which was defined for the intuitive model. The only definitions which actually look different are definitions for focus, unfocus, adoption and allocation.

Adoption and allocation need to be rewritten because regions are no longer sets, but functions from locations to objects. In the allocation rule, notation $(p \mapsto o)$ denotes the function of domain $\{p\}$, which associates o to p. More importantly, we need to rewrite rules for focus, unfocus and adoption. Indeed, this time the object associated to the location must be moved with the location. Note that the rule for adoption is the same as the rule for unfocus. Also note that in those two rules, it happens that $p = \mathcal{H}_s(x)$ and $o = \mathcal{H}_s(p)$.

Allocation Although pointers are separated, they are still allocated using a single counter $\mathcal{H}_s(next)$. There is not one such counter per region. This implies, in particular, that if two locations are in two distinct regions and if some permissions are available on those two regions, then the locations are different. The permission condition is important as some locations may actually be in several regions because of the **focus** and **adopt** operations. We show in Section 5.7 how the programmer can use such kind of information.

$\mathcal{H}_s$, **let** $x = e \Longrightarrow \mathcal{H}_s[x \mapsto (\!|e|\!)_{\mathcal{H}_s}]$
$\mathcal{H}_s$, **let region** r: $\mathcal{C} \Longrightarrow \mathcal{H}_s[r \mapsto \emptyset]$
$\mathcal{H}_s$, **unpack** $x \Longrightarrow \mathcal{H}_s$
$\mathcal{H}_s$, **weaken empty** $\rho \Longrightarrow \mathcal{H}_s$
$\mathcal{H}_s$, **weaken single** $\rho \Longrightarrow \mathcal{H}_s$
$\mathcal{H}_s$, $x.f \leftarrow e \Longrightarrow \mathcal{H}_s[\mathcal{H}_s(x) \mapsto \mathcal{H}_s(x)[f \mapsto (\!|e|\!)_{\mathcal{H}_s}]\ ]$

$\mathcal{H}_s$, **if** e **then** S **else** $S' \Longrightarrow \mathcal{H}_s{}'$
with $\mathcal{H}_s$, $S \Longrightarrow \mathcal{H}_s{}'$ if $(\!|e|\!)_{\mathcal{H}_s} = \mathit{true}$ and $\mathcal{H}_s$, $S' \Longrightarrow \mathcal{H}_s{}'$ if $(\!|e|\!)_{\mathcal{H}_s} = \mathit{false}$.

$\mathcal{H}_s$, **let** $x = f\ \mathit{arg}_1 \cdots \mathit{arg}_l \Longrightarrow \mathcal{H}_{s4}[x \mapsto (\!|e|\!)_{\mathcal{H}_{s3}}]$
where:
$\mathit{arg}_1 \cdots \mathit{arg}_l$ restricted to region arguments is $[\rho_1] \cdots [\rho_n]$,
$\mathit{arg}_1 \cdots \mathit{arg}_l$ restricted to regular arguments is $(e_1) \cdots (e_m)$,
f was declared with region arguments $r_1, \cdots, r_n$ in this order,
f was declared with regular arguments $x_1, \cdots, x_m$ in this order,
f was declared with body $s_1; \cdots; s_n$; **return** e,
f was declared with pre-condition P and post-condition Q,
$\mathcal{H}_{s1} = \mathcal{H}_s[x_1 \mapsto (\!|e_1|\!)_{\mathcal{H}_s}] \cdots [x_m \mapsto (\!|e_m|\!)_{\mathcal{H}_s}]$,
$\mathcal{H}_{s2} = \mathcal{H}_{s1}[r_1 \mapsto \mathcal{H}_s(\rho_1)] \cdots [r_n \mapsto \mathcal{H}_s(\rho_n)][\rho_1? \mapsto r_1] \cdots [\rho_n? \mapsto r_n]$,
$\mathcal{H}_{s2}$, $s_1; \cdots; s_l \Longrightarrow \mathcal{H}_{s3}$,
$\mathcal{H}_{s4} = \mathcal{H}_{s3}[\rho_1 \mapsto \mathcal{H}_{s3}(r_1)] \cdots [\rho_n \mapsto \mathcal{H}_{s3}(r_n)][r_1? \mapsto \rho_1] \cdots [r_n? \mapsto \rho_n]$,
and only if $(\!|P|\!)_{\mathit{loh}_s(\mathcal{H}_{s2})}$ and $(\!|Q|\!)_{\mathit{loh}_s(\mathcal{H}_{s3}[\textsf{result} \mapsto (\!|e|\!)_{\mathcal{H}_{s3}}])}$ hold.

$\mathcal{H}_s$, **let** $x =$ **new** $\mathcal{C}\ [\rho] \Longrightarrow \mathcal{H}_s[\rho \mapsto (p \mapsto o)][x \mapsto p][p? \mapsto \rho][\mathit{next} \mapsto p{+}1]$
where $p = \mathcal{H}_s(\mathit{next})$ and o is an object associating $\emptyset$ to every region owned by $\mathcal{C}$.

$\mathcal{H}_s$, **pack** x: $\mathcal{C} \Longrightarrow \mathcal{H}_s$
if $(\!|\mathit{Inv}_{\mathcal{C}}(\mathcal{H}_s(x))|\!)_{\mathit{loh}_s(\mathcal{H}_s)}$ holds.

$\mathcal{H}_s$, **assert** $P \Longrightarrow \mathcal{H}_s$
if $(\!|P|\!)_{\mathit{loh}_s(\mathcal{H}_s)}$ holds.

$\mathcal{H}_s$, **focus** x: ρ **as** $\sigma \Longrightarrow \mathcal{H}_s[\sigma \mapsto (p \mapsto \mathcal{H}_s(p))][p? \mapsto \sigma]$
where $p = \mathcal{H}_s(x)$.

$\mathcal{H}_s$, **unfocus** x: σ **as** $\rho \Longrightarrow \mathcal{H}_s[\rho \mapsto \mathcal{H}_s(\rho)[p \mapsto o]\ ][p? \mapsto \rho]$
if $\mathcal{H}_s(\sigma) = p \mapsto o$.

$\mathcal{H}_s$, **adopt** x: σ **as** $\rho \Longrightarrow \mathcal{H}_s[\rho \mapsto \mathcal{H}_s(\rho)[p \mapsto o]\ ][p? \mapsto \rho]$
if $\mathcal{H}_s(\sigma) = p \mapsto o$.

$\mathcal{H}_s$, **label** $L \Longrightarrow \mathcal{H}_s[L \mapsto \mathit{cur}]$
where cur is $\mathit{loh}_s(\mathcal{H})$ restricted to labels.

Figure 4.21: Statement Semantics (Separated Model)

$\mathcal{H}_s$, **if** e **then** S **else** $S' \Longrightarrow \infty$
with $\mathcal{H}_s$, $S \Longrightarrow \infty$ if $(\!|e|\!)_{\mathcal{H}_s} = \mathit{true}$ and $\mathcal{H}_s$, $S' \Longrightarrow \infty$ if $(\!|e|\!)_{\mathcal{H}_s} = \mathit{false}$.

$\mathcal{H}_s$, **let** $x = f\ arg_1 \cdots arg_l \Longrightarrow \infty$
where:
$arg_1 \cdots arg_l$ restricted to region arguments is $[\rho_1] \cdots [\rho_n]$,
$arg_1 \cdots arg_l$ restricted to regular arguments is $(e_1) \cdots (e_m)$,
f was declared with region arguments $r_1, \cdots, r_n$ in this order,
f was declared with regular arguments $x_1, \cdots, x_m$ in this order,
f was declared with body $s_1; \cdots; s_n$; **return** e,
f was declared with pre-condition P and post-condition Q,
$\mathcal{H}_{s1} = \mathcal{H}_s[x_1 \mapsto (\!|e_1|\!)_{\mathcal{H}_s}] \cdots [x_m \mapsto (\!|e_m|\!)_{\mathcal{H}_s}]$,
$\mathcal{H}_{s2} = \mathcal{H}_{s1}[r_1 \mapsto \mathcal{H}_s(\rho_1)] \cdots [r_n \mapsto \mathcal{H}_s(\rho_n)][\rho_1? \mapsto r_1] \cdots [\rho_n? \mapsto r_n]$,
$\mathcal{H}_{s2}$, $s_1; \cdots; s_l \Longrightarrow \infty$

Figure 4.22: Statement Semantics: Divergence (Separated Model)

4.5.2 Intuitive and Separated Model Equivalence

We now compare the intuitive model and the separated model. We define a relation between intuitive heaps and separated heaps, and we show that executing a statement preserves this relation.

We define relation $\mathcal{R}_o(o, o_s)$ between an intuitive heap $\mathcal{H}$, an intuitive object o and a separated object o_s as the smallest relation verifying the following properties:

1. for all field name f, $o(f)$ is defined if, and only if $o_s(f)$ is defined and if so, $o_s(f) = o(f)$;
2. for all region name r, $o(r)$ is defined if, and only if $o_s(r)$ is defined and if so, $\mathit{Dom}(o_s(r)) = o(r)$.

We define relation $\mathcal{R}(\mathcal{H}, \mathcal{H}_s)$ between an intuitive heap $\mathcal{H}$ and a separated heap $\mathcal{H}_s$ as the smallest relation verifying the following properties:

1. for all variable name x, $\mathcal{H}(x)$ is defined if, and only if $\mathcal{H}_s(x)$ is defined and if so, $\mathcal{H}_s(x) = \mathcal{H}(x)$;
2. for all location p, $\mathcal{H}(p?)$ is defined if, and only if $\mathcal{H}_s(p?)$ is defined and if so, $\mathcal{H}_s(p?) = \mathcal{H}(p?)$;
3. for all location p, $\mathcal{H}(p)$ is defined if, and only if $\mathcal{H}_s(p)$ is defined and if so, $\mathcal{R}_o(\mathcal{H}(p), \mathcal{H}_s(p))$;
4. for all region name r, $\mathcal{H}(r)$ is defined if, and only if $\mathcal{H}_s(r)$ is defined and if so, $\mathit{Dom}(\mathcal{H}_s(r)) = \mathcal{H}(r)$;
5. $\mathcal{H}(\mathit{next}) = \mathcal{H}_s(\mathit{next})$;
6. for all label L, $\mathcal{H}(L)$ is defined if, and only if $\mathcal{H}_s(L)$ is defined and if so, $\mathcal{H}(L) = \mathcal{H}_s(L)$.

Note that for two heaps to be in relation, it is necessary for paths described by active regions of pointers to be finite. Indeed, $\mathcal{H}_s(p)$ is not defined otherwise.

We first show that terms and region terms evaluate to the same values in both models.

Theorem 2 (Soundness of Separated Term Evaluation) Let $\overline{\Sigma}$ be a set of permissions, Γ an environment, RT a well-typed region term with respect to Γ, LT a well-typed term with respect to Γ, $\mathcal{H}$ an coherent intuitive heap with respect to $\overline{\Sigma}$ and Γ, and $\mathcal{H}_s$ a separated heap such that $\mathcal{R}(\mathcal{H}, \mathcal{H}_s)$. Let $\mathcal{L} = loh(\mathcal{H})$ and $\mathcal{L}_s = loh_s(\mathcal{H}_s)$. Then:

1. $[\![RT]\!]_{\mathcal{L}}$ and $(\!|RT|\!)_{\mathcal{L}_s}$ are defined, and are equal region functions;
2. $[\![LT]\!]_{\mathcal{L}}$ and $(\!|LT|\!)_{\mathcal{L}_s}$ are defined, and are equal values.

Proof. By structural induction on RT and LT. More precisely, our induction hypothesis is the following. Let x be either a a well-typed region term or a well-typed term. Then for all sub-region-term RT of x, $[\![RT]\!]_{\mathcal{L}}$ and $(\!|RT|\!)_{\mathcal{L}_s}$ are defined, and are equal region functions. And for all sub-term LT of x, $[\![LT]\!]_{\mathcal{L}}$ and $(\!|LT|\!)_{\mathcal{L}_s}$ are defined, and are equal values.

First, we consider the cases where x is a term.

- c

  Constants are trivially evaluated as the same values.

- x

  From item 6 of coherence we know that $\mathcal{H}(x)$ is defined. $\mathcal{L}(x)$ and $\mathcal{L}_s(x)$ are, by construction, respectively equal to $\mathcal{H}(x)$ and $\mathcal{H}_s(x)$, which are equal by $\mathcal{R}$ if they are defined.

- LT_1 *termop* LT_2

  We apply the induction hypothesis on LT_1 and LT_2. Operations *termop* then evaluate equally.

- $f\ \overline{larg}$

  For each argument *larg*, if *larg* is of the form $[RT]$ then we apply the induction hypothesis on RT, else *larg* is of the form (LT) and we apply the induction hypothesis on LT. Function $[\![f]\!]$ applied to these arguments is then the same in both models.

- $get(RT, LT, f)$

  We apply the induction hypothesis on RT and LT.

  Then we prove that *lor* and lor_s construct the same logic region. On *contents*, these logic regions are equal by $\mathcal{R}$. On each location p, these logic regions are equal by $\mathcal{R}$. The domain of these logic regions are equal by construction of the flattenings.

  Finally we use $\mathcal{R}$ to obtain the fact that the f fields are equal in both models.

Then we consider the cases where x is a region term.

- $r@L$

  This case amounts to showing that *lor* and lor_s construct the same logic region, in a similar fashion than for the $get(RT, LT, f)$ case above. We also use the fact that $\mathcal{H}(L) = \mathcal{H}_s(L)$.

- $get(RT, LT, s)$

  We apply the induction hypothesis on RT and LT, and then we show that *lor* and lor_s construct the same logic region in a similar fashion than we did for the above $get(RT, LT, f)$ case.

□

We can now show a similar theorem for predicates.

Theorem 3 (Soundness of Separated Predicate Evaluation) Let $\overline{\Sigma}$ be a set of permissions, Γ an environment, P a well-typed predicate with respect to $logicenv(\Gamma)$, $\mathcal{H}$ an coherent intuitive heap with respect to $\overline{\Sigma}$ and Γ, and $\mathcal{H}_s$ a separated heap such that $\mathcal{R}(\mathcal{H}, \mathcal{H}_s)$. Let $\mathcal{L} = loh(\mathcal{H})$ and $\mathcal{L}_s = loh_s(\mathcal{H}_s)$. Then $[\![P]\!]_{\mathcal{L}}$ and $(\!|P|\!)_{\mathcal{L}_s}$ are defined and are equivalent.

Proof. By induction on P. We apply Theorem 2 on terms and region terms, in particular for the case of the $LT \in RT$ predicate. □

We now show a similar theorem for expressions.

Theorem 4 (Soundness of Separated Expression Evaluation) Let $\overline{\Sigma}$ be a set of permissions, Γ an environment, e a well-typed expression with respect to $\overline{\Sigma}$ and Γ, $\mathcal{H}$ an coherent intuitive heap with respect to $\overline{\Sigma}$ and Γ, and $\mathcal{H}_s$ a separated heap such that $\mathcal{R}(\mathcal{H}, \mathcal{H}_s)$. Then $[\![e]\!]_{\mathcal{H}}$ and $(\!|e|\!)_{\mathcal{H}_s}$ are defined and $[\![e]\!]_{\mathcal{H}} = (\!|e|\!)_{\mathcal{H}_s}$.

Proof. By induction on e. Item 6 of coherence ensures that $[\![e]\!]_{\mathcal{H}}$ is defined. We consider each case for e.

- c

  Constants are trivially equally evaluated.

- x

  As $\mathcal{H}(x)$ is defined, so is $\mathcal{H}_s(x)$ by $\mathcal{R}$ and they are equal.

- $e_1\ op\ e_2$

  We apply the induction hypothesis on e_1 and e_2. Operations op then evaluate equally.

- $f\ \overline{arg}$

  For each argument arg, if arg is of the form (e) we apply the induction hypothesis on e. Else arg is of the form $[\rho]$, and we show that $lor_s'(\mathcal{H}_s(\rho), \mathcal{H}_s) = lor(\mathcal{H}, \mathcal{H}(\rho))$ as we did for the r case of Theorem 2. Function $[\![f]\!]$ applied to these arguments is then the same in both models.

- $x.f$

  By definition of $\mathcal{R}$, $\mathcal{H}_s(p)$ is defined and $\mathcal{R}_o(\mathcal{H}(p), \mathcal{H}_s(p))$. By definition of $\mathcal{R}_o$, we conclude that $\mathcal{H}(p)(f) = \mathcal{H}_s(p)(f)$.

Note how similar the proof is to the proof of Theorem 2. □

We finally show our main theorem for statements, relating intuitive and separated heaps.

Theorem 5 (Soundness of Separated Statement Execution) Let s be a statement such that:

$$\overline{\Sigma}, \Gamma \vdash \{\overline{\Sigma}\}\ s\ \{\overline{\Sigma}'\}, \Gamma'$$

Let $\mathcal{H}$ an intuitive heap, coherent with respect to $\overline{\Sigma}$ and Γ. Let $\mathcal{H}_s$ a separated heap, such that $\mathcal{R}(\mathcal{H}, \mathcal{H}_s)$.

There is $\mathcal{H}'$ such that:

$$\mathcal{H},\ s \Longrightarrow \mathcal{H}'$$

if, and only if there is $\mathcal{H}_s{}'$ such that:

$$\mathcal{H}_s,\ s \Longrightarrow \mathcal{H}_s{}'$$

and if so, then $\mathcal{R}(\mathcal{H}',\ \mathcal{H}_s{}')$.

Additionally:

$$\mathcal{H},\ s \Longrightarrow \infty$$

if, and only if:

$$\mathcal{H}_s,\ s \Longrightarrow \infty$$

Proof. We begin by proving the case of programs which terminates. We prove the first implication by induction on the derivation of $\mathcal{H},\ s \Longrightarrow \mathcal{H}'$, and the other by induction on the derivation of $\mathcal{H}_s,\ s \Longrightarrow \mathcal{H}_s{}'$. Here we only tackle the first implication; the other is similar. We examine each possible case for s.

- **let** $x = e$

  Theorem 4 gives $[\![e]\!]_{\mathcal{H}} = (\!|e|\!)_{\mathcal{H}_s} = v$. Instruction s reduces in $\mathcal{H}$ into $\mathcal{H}[x \mapsto v]$ and in $\mathcal{H}_s$ into $\mathcal{H}_s[x \mapsto v]$. So we do have $\mathcal{R}(\mathcal{H}',\ \mathcal{H}_s{}')$ as $\mathcal{H}'(x) = \mathcal{H}_s{}'(x) = v$.

- **if** e **then** σ_1 **else** σ_2

  From Theorem 4, $[\![e]\!]_{\mathcal{H}} = [\![e]\!]_{\mathcal{H}_s}$. If $[\![e]\!]_{\mathcal{H}} = \mathit{true}$, we apply the induction hypothesis on σ_1, else we apply it on σ_2.

- **let** $x = f\ \overline{arg}$

  We first show $\mathcal{R}(\mathcal{H}_1,\ \mathcal{H}_{s\,1})$ where $\mathcal{H}_1$ and $\mathcal{H}_{s\,1}$ are defined as in the reduction rule. To this end we simply apply Theorem 4 on expressions $\overline{e}$.

  We now show $\mathcal{R}(\mathcal{H}_2,\ \mathcal{H}_{s\,2})$. For all i, $\mathcal{H}_1(\rho_i)$ is defined by item 7 of coherence and $\mathcal{H}_{s\,1}(\rho_i)$ is defined by definition of $\mathcal{R}$. From $\mathcal{R}(\mathcal{H}_1,\ \mathcal{H}_{s\,1})$ we obtain $\mathit{Dom}(\mathcal{H}_{s\,1}(\rho_i)) = \mathcal{H}_1(\rho_i)$. So $\mathit{Dom}(\mathcal{H}_{s\,2}(r_i)) = \mathcal{H}_2(r_i)$. Typing rule requires that for all i and j such that $i \neq j$, region arguments ρ_i and ρ_j do not share the same root. In particular, this implies that $\rho_i \neq \rho_j$. So if $\mathcal{H}_1(p?) = \rho_i$, then $\mathcal{H}_2(p?) = r_i$. As we also copied the contents of r_i, we have $\mathcal{R}_o(\mathcal{H}_2(p),\ \mathcal{H}_{s\,2}(p))$. Active region of other pointers did not change.

  We then show that $\mathcal{H}_3$ is defined if, and only if $\mathcal{H}_{s\,3}$ is defined, and that $\mathcal{R}(\mathcal{H}_3,\ \mathcal{H}_{s\,3})$. To this end we apply the induction hypothesis on the body of f. We apply Theorem 3 on the pre-condition in $\mathcal{H}_2$ and $\mathcal{H}_{s\,2}$, and on the post-condition in $\mathcal{H}_3$ and $\mathcal{H}_{s\,3}$.

  Typing rule requires that for all i and j, region arguments ρ_i and ρ_j do not share the same root. This implies that modifying $\mathcal{H}_s(\rho_i)$ cannot also change $\mathcal{H}_s(\rho_j)$ as a side-effect, and vice versa. Indeed, if $\mathit{root}(\rho_i) = r_a$ and $\mathit{root}(\rho_j) = r_b$ then $r_a \neq r_b$. And by definition, modifying $\mathcal{H}_s(\rho_i)$ modifies $\mathcal{H}_s(r_a)$ and only $\mathcal{H}_s(r_a)$, while reading $\mathcal{H}_s(\rho_j)$ implies reading $\mathcal{H}_s(r_b)$ and only $\mathcal{H}_s(r_b)$.

  We finally show that $\mathcal{H}'$ is defined if, and only if $\mathcal{H}_s{}'$ is defined and that $\mathcal{R}(\mathcal{H}',\ \mathcal{H}_s{}')$. As we noted, regions ρ_i all have different roots, so we have $\mathcal{H}_s{}'(\rho_i) = \mathcal{H}_{s\,3}(r_i)$ for all i. We apply Theorem 4 on returned expression e, as well as a similar reasoning to the one we used for $\mathcal{H}_2$ and $\mathcal{H}_{s\,2}$ on the copy from r_i regions to ρ_i regions.

- **let region** r: $\mathcal{C}$

  Statement s always reduce in both model. New region r is empty in both cases.

- **let** x = **new** $\mathcal{C}$ $[\rho]$

  From $\mathcal{R}$ we have:

  $$p = \mathcal{H}(next) = \mathcal{H}_s(next)$$

  We have:

  $$Dom(\mathcal{H}_s'(\rho)) = Dom(p \mapsto o) = \{p\} = \mathcal{H}'(\rho)$$

  $$\mathcal{H}_s'(p?) = \rho = \mathcal{H}'(p?)$$

  $$\mathcal{H}_s'(x) = p = \mathcal{H}'(x)$$

  There is no initial value for fields in both semantics, so both $\mathcal{H}'(p)(f)$ and $\mathcal{H}_s'(p)(f)$ are undefined. For all region r defined in $\mathcal{H}'(p)$, $\mathcal{H}'(p)(r)$ is the empty set and $\mathcal{H}_s'(p)(r)$ has empty domain, so $\mathcal{R}_o(\mathcal{H}'(p), \mathcal{H}_s'(p))$. Finally, we have:

  $$\mathcal{H}'(next) = p{+}1 = \mathcal{H}_s'(next)$$

- **focus** x: ρ **as** σ

  Let $p = [\![x]\!]_{\mathcal{H}}$, which is also equal to $(\!|x|\!)_{\mathcal{H}_s}$ by Theorem 4. We also have by definition of the reduction rule:

  $$Dom(\mathcal{H}_s'(\sigma)) = Dom(p \mapsto o) = \{p\} = \mathcal{H}'(\sigma)$$

  $$\mathcal{H}_s'(p?) = \sigma = \mathcal{H}'(p?)$$

  Finally, by $\mathcal{R}(\mathcal{H}, \mathcal{H}_s)$, for all f we have:

  $$\mathcal{H}'(p)(f) = \mathcal{H}(p)(f) = \mathcal{H}_s(p)(f) = \mathcal{H}_s'(p)(f)$$

  and for all region name r:

  $$\mathcal{H}'(p)(r) = \mathcal{H}(p)(r) = Dom(\mathcal{H}_s(p)(r)) = Dom(\mathcal{H}_s'(p)(r))$$

- $x.f \leftarrow e$

  We apply typing rules and coherence to show that $\mathcal{H}(\mathcal{H}(x))(f)$ is defined, and relation $\mathcal{R}$ to show that $\mathcal{H}_s(\mathcal{H}_s(x))(f)$ is defined as well. Statement thus reduces in both models. We apply Theorem 4 to show that e evaluates into the same value in both heaps which thus stay in relation.

- **adopt** x: σ **as** ρ

  Region σ is singleton by typing and thus contains a unique location p by coherence. Its active region becomes ρ. We have $\mathcal{H}_s'(p) = \mathcal{H}_s(\sigma)(p)$. Item 8 of coherence gives $\mathcal{H}_s(p?) = \sigma$ as $\sigma^\times$ was available before the adoption. Thus $\mathcal{H}_s(p) = \mathcal{H}_s(\sigma)(p) = \mathcal{H}_s'(p)$.

- **unfocus** x: σ **as** ρ

  The proof is similar to the one for adoption, as the execution rules are the same.

- **pack** x

  The statement reduces if, and only if the invariant predicate on x holds. We apply Theorem 3 to obtain that the statement reduces in the intuitive model if, and only if it reduces in the separated model. The heaps are not modified and thus are still in relation.

- **unpack** x, **weaken empty** x, **weaken single** x

  Statement always reduce and heap is not modified.

- **assert** P

  This is the same proof than for packing, except with predicate P instead of the invariant.

- **label** L

  It is sufficient to prove that for each label L, we have $loh(\mathcal{H})(L) = loh_s(\mathcal{H}_s)(L)$. Let r be a region name such that $\mathcal{H}(r)$ is defined. Thus $loh(\mathcal{H})(L)(r)$ is defined. From $\mathcal{R}$, so is $\mathcal{H}_s(r)$ and thus so is $loh_s(\mathcal{H}_s)(L)(r)$. We have already shown for terms how to prove that lor and lor_s construct the same logic region.

The proof for programs which diverge is done by coinduction on the derivation $\mathcal{H}$, $s \Longrightarrow \infty$ or $\mathcal{H}_s$, $s \Longrightarrow \infty$. Each case is dealt with similarly to the terminating case. □

A corollary of this theorem is that paths described by active regions in $\mathcal{H}'$ are finite. Indeed, $\mathcal{H}_s'$ would otherwise not exist.

Finally, let's define the starting intuitive heap $\mathcal{H}_0$ and the starting separated heap $\mathcal{H}_{s0}$, both only defined on *next*, and with $\mathcal{H}_{s0}(next) = \mathcal{H}_0(next)$. We remark that $\mathcal{H}_0$ is coherent with respect to the empty environment and the empty list of permissions, and that $\mathcal{R}(\mathcal{H}_0, \mathcal{H}_{s0})$ holds. We thus do have a starting point for our programs.

Note that we did not show that well-typed statements always reduce. This actually is false, as we also requires proof obligations to be proven to ensure that pre- and post-conditions, invariants and assertions hold. We will discuss this when introducing how proof obligations are computed, in Chapter 5.

4.6 Conclusion

In this section we presented the core language of Capucine. We gave its syntax, typing rules involving permissions, and semantics for two different models: the intuitive model and the separated model. The former is close to models of existing languages, while the latter separates regions more and is closer to the Why interpretation of Capucine, which is discussed in Chapter 5. We have shown two main theorems: coherence of the heap is preserved in the intuitive model, and a relation between an intuitive heap and a separated heap is preserved. From this we conclude that coherence of the heap is preserved in the separated model.

This work will be used in Chapter 5 to prove properties about interpreted programs, as well as a *progress* theorem stating that a well-typed Capucine program whose proof obligations have been proved reduces. We cannot state this progress theorem yet, as we first require to know how proof obligations are produced.

Finding the right definition for coherence, and the right definition for the $\mathcal{R}$ relation between an intuitive heap and a separated heap, is crucial. Their knowledge is not needed to *use* Capucine, but they play a central role in the *proof* that Capucine is a sound language. Following this, the semantics must be defined with great care.

It is important to note that a permission is needed to select fields in *expressions*, but that no permission is needed to select fields in *terms*. The latter is a strong feature of Capucine: no permission is needed to give semantics to the logic part of the language. In programs, a permission on x is needed to read $x.f$ to ensure that the active region of x is defined; but there is no active region in the logic model. It is possible thanks to the construction of logic environments $\mathcal{L}$ and $\mathcal{L}_s$, and more precisely to the construction of logic regions using *lor* and lor_s.

However, typing requires that program regions used in the logic are *focus-free*. This restriction is not necessary to prove that the intuitive model and the separated model are equivalent, but it greatly simplifies the Why interpretation introduced in Chapter 5.

Chapter 5

Generation of Verification Conditions

In this chapter, we discuss how proof obligations of Capucine programs are produced. We also discuss how proving these proof obligations relate to the soundness of Capucine programs.

5.1 The Why Intermediate Language

In order to produce proof obligations, we encode Capucine programs as Why programs [Filliâtre07]. We introduced the Why language informally in Section 2.3.1. In this section we formally introduce the subpart of Why that we will be using[1]. To this end we present the syntax of Why programs. We do not fully formalize the semantics of Why, which is straightforward. However we do state the main theorem we will be using, namely that if the proof obligations generated by the Why tool on a Why program are proven, then the Why program executes safely.

We index Why syntactic classes with letter w. For instance, Why types are τ_w. Compare this to Capucine types τ.

5.1.1 Types

A Why logic type is of the form:

$\tau_w ::= t\ (\tau_w, \cdots, \tau_w) \mid \mathit{int} \mid \mathit{bool} \mid \mathit{unit}$

It is the set of Capucine types without pointer types. User types t may have any number of parameters, and base types are the usual integers, booleans and unit.

A Why program type is either a Why logic type, or a reference on a Why logic type:

$\tau_w{}' ::= \tau_w \mid \mathbf{ref}\ (\tau_w)$

5.1.2 Terms

A Why term is of the form:

[1] The syntax we use is slightly different than the actual syntax of Why.

$$
\begin{array}{rcl}
LT_w & ::= & c \\
& | & x \\
& | & LT_w \ termop \ LT_w \\
& | & f\ (\overline{LT_w}) \\
& | & !x \\
& | & !x@L
\end{array}
$$

$termop ::= + \mid - \mid \times \mid \&\& \mid ||$

where c is a constant of a base type, i.e. either an integer literal or a boolean *true* or *false*. The set of Why terms contains most of Capucine terms, but there is no field selection. Instead, we may dereference a reference variable x using $!x$, or using $!x@L$ where L is a label. Note that reference variables can only be program variables, they cannot be logically quantified variables. Compared to Capucine, the application is simpler, as there is no region term.

5.1.3 Predicates

A Why predicate is of the form:

$$
\begin{array}{rcl}
P_w & ::= & p\ (\overline{LT_w}) \\
& | & P_w \wedge P_w \mid P_w \vee P_w \mid P_w \Rightarrow P_w \mid P_w \iff P_w \mid \neg P_w \mid \mathsf{T} \mid \mathsf{F} \\
& | & LT_w \ relop \ LT_w \\
& | & \forall x\colon \tau_w.\ P_w \mid \exists x\colon \tau_w.\ P_w
\end{array}
$$

$relop ::= = \mid \neq \mid > \mid \geq \mid < \mid \leq$

Once again, this is a subset of Capucine predicates. There is no region, thus there is no region quantification nor any $\in$ predicate, and application is simpler.

5.1.4 Expressions

A Why expression is of the form:

e_w	::=	c	Constant
	|	x	Variable
	|	$e_w \ op \ e_w$	Arithmetic or logic operation
	|	$f\ (\overline{e_w})$	Logic function application
	|	**if** e_w **then** e_w **else** e_w	Test
	|	**let** $x = e_w$ **in** e_w	Let-binding
	|	$(e_w, \cdots, e_w)$	Tuple
	|	$proj_i(e_w)$	Tuple projection
	|	**let** $x =$ **ref** e_w **in** e_w	Reference creation
	|	$!x$	Dereferenced variable
	|	$x := e_w$	Assignment
	|	$e_w; e_w$	Sequence
	|	$[\{P_w\}$ **reads** $\overline{x}$ **writes** $\overline{x}\ \tau_w\ \{P_w\}]$	Black box
	|	**label** L; e_w	Label

Expressions Without Side-Effects Constants are the same as Capucine constants: integers, booleans and unit. Operations *op* are the same as Capucine operations on integers

and booleans. Logic functions can be applied in expressions. The **if** test and let-binding are also expressions. Tuples are available, and the i-th component of a tuple $(x_1, \cdots, x_n)$, i.e. x_i, can be obtained using projection $proj_i$.

Expressions With Side-Effects Why expressions may contain side-effects. References can be created with the **ref** keyword. Remember though that the type system ensures that there is no alias. A variable x on a reference can be dereferenced with $!x$, and assigned to with $:=$. The sequence is also an expression.

Black Boxes Black boxes are special expressions with side-effects:

$$[\{P\}\ \textbf{reads}\ \overline{x}\ \textbf{writes}\ \overline{y}\ \tau_w\ \{Q\}]$$

is an expression of type τ_w, with pre-condition P and post-condition Q, and which reads references $\overline{x}$ and writes references $\overline{y}$. The actual value returned by the black box is non-deterministic; it only has to verify the post-condition, in which variable **result** is bound to the returned value.

We may omit the **reads** clause, in which case $\overline{x}$ is implicitely equal to the set of references read by P and Q. If $\overline{y}$ is the empty list, we may omit the **writes** clause. We denote by **assert** P_w the black box $[\{P_w\}\ unit\ \{\mathsf{T}\}]$, by **assume** P_w the black box $[\{\mathsf{T}\}\ unit\ \{P_w\}]$, and by **any** τ the black box $[\{\mathsf{T}\}\ \tau\ \{\mathsf{T}\}]$. Finally, if $e_{w\,1}$ and $e_{w\,2}$ have type τ_w, we denote by:

$$\textbf{if}\ P_w\ \textbf{then}\ e_{w\,1}\ \textbf{else}\ e_{w\,2}$$

the following black box:

$$[\tau_w\ \{P_w \Rightarrow \textbf{result} = e_{w\,1} \wedge \neg P_w \Rightarrow \textbf{result} = e_{w\,2}\}]$$

Note that black boxes, such as **assume** F for instance, may introduce inconsistent hypotheses in proof obligations. It is up to the user to guarantee that black boxes are realizable.

5.1.5 Logic Declarations

Types User types may be introduced using a declaration of the form:

type $t\ (\overline{\alpha})$

It is simply a name t and a list of type variables as parameters.

Logic Functions and Predicates Logic functions may be introduced using a declaration of the form:

logic $f\ (\overline{x\colon \tau_w})\colon \tau_w = LT_w$

The body LT_w may be omitted. If it is, variable names x may be omitted as well. If it is not, the return type may be omitted: it will be inferred.

Predicates are defined in a similar fashion:

logic $p\ (\overline{x\colon \tau_w}) = P_w$

Once again, the body P_w may be omitted, and if so, variable names x may be omitted as well.

Axioms Axioms may be defined using a declaration of the form:

axiom a: P_w

where a is the name of the axiom.

5.1.6 Program Declarations

Functions Functions may be declared as follows.

fun ($\overline{x: \tau_w}$): τ_w
  pre P_w
  post P_w
{
  e_w
}

They are similar to Capucine function declarations, except that there is no permissions, and thus no **consumes** and **produces** clauses. Moreover, the body is not a list of statements but an expression. Functions generate proof obligations.

Parameters Parameters are functions whose body { e_w } was omitted:

fun ($\overline{x: \tau_w}$): τ_w
  pre P_w
  post P_w
  reads $\overline{x}$
  writes $\overline{x}$

Note that to replace the body, there is **reads** and **writes** clauses. In fact, parameters can be seen as block boxes with a name and with arguments. They will not generate proof obligations as there is nothing to prove. They can, however, be called from other functions.

5.1.7 Model

Values Values are either integers, booleans, unit, tuples, or values of user-defined types.

Heap A Why heap H_w is a function:

- $H_w(x)$ is the value of variable x;
- $H_w(y, L)$ is the value of reference variable y at label L.

Note that $!y$ is evaluated as $H_w(y,$ **here**$)$. In other words, label **here** is the default label.

Execution The semantics of expressions is defined by the following relations:

$$e_w, H_w \overset{\text{Why}}{\Longrightarrow} v, H_w{}'$$

which states that Why expression e_w under heap H_w reduces to value v, with new heap $H_w{}'$. The evaluation of a Why expression e_w might also diverge. This is denoted by:

$$e_w, H_w \overset{\text{Why}}{\Longrightarrow} \infty$$

It may happen because of loops and recursive calls. We will not be using loops though, so we are only concerned with recursive calls. A program which diverges does not raise any run-time error, so it is a correct program. Note that because some Why expressions are non-deterministic, both judgements may hold for the same program.

The following theorem states that the Why language is sound [Filliâtre03].

Theorem 6 (Why Soundness) Assume e_w is a well-typed Why expression, whose pre-condition has been proven to entail its weakest pre-condition. Assume that all functions have been proven to verify their contract. Then for every heap H_w in which the pre-condition holds, either:

$$e_w,\, H_w \stackrel{\text{Why}}{\Longrightarrow} \infty$$

or there is v and ${H_w}'$ such that:

$$e_w,\, H_w \stackrel{\text{Why}}{\Longrightarrow} v,\, {H_w}'$$

5.2 Direct Encoding of the Separated Model

In this section, we show how to translate Capucine programs into Why programs. We base our encoding on the separated model introduced in Section 4.5.

5.2.1 Pointers, Regions, Types and Objects

Understanding how separated heaps $\mathcal{H}_s$ are encoded in Why is key. Indeed, the remaining of the translation follows directly from this encoding.

Pointers Regions are encoded as logic maps. Keys to these maps are locations:

type *location*

This type is abstract. The only way to build a location is by using a black box of type *location*. Indeed, there is no logic function which returns a *location*.

Regions We declare the Why type of regions:

type *region* (α)

It denotes maps from *location* values to α values. Logic function *set* adds a location and its α value into a region:

logic *set* (α *region*, *location*, α): *region* (α)

If the location already has an associated value in the map, it is replaced. Logic function *get* reads the value of a location in a region:

logic *get* (α *region*, *location*): α

Functions *get* and *set* are axiomatized as follows:

axiom *getSetEq*:
$\forall r$: *region* (α). $\forall p$: *location*. $\forall x$: α.

$get(set(r,\ p,\ x),\ p) = x$

axiom *getSetNeq*:
$\forall r$: *region* (α). $\forall p$: *location*. $\forall q$: *location*. $\forall x$: α.
$p \neq q \Rightarrow get(set(r,\ p,\ x),\ q) = get(r,\ q)$

This is the usual theory of infinite arrays.

We add the notion of *domain* for regions. Logic constant *empty* is a region of empty domain:

logic *empty*: *region* (α)

Predicate *inRegion* takes a location and a region as arguments, and holds if, and only if the location is in the domain of the region:

logic *inRegion* (*location*, *region* (α))

This predicate is defined using four axioms. The first one states that the empty region is empty, i.e. has an empty domain:

axiom *notInEmptyRegion*:
$\forall p$: *location*.
$\neg inRegion(p,\ empty)$

Another one states that location p is in regions into which p was added:

axiom *inSetRegionEq*:
$\forall p$: *location*. $\forall r$: *region* (α). $\forall v$: α.
$inRegion(p,\ set(r,\ p,\ v))$

The next one states that *set* only makes the domain bigger:

axiom *inSetRegionNeq*:
$\forall p$: *location*. $\forall q$: *location*. $\forall r$: *region* (α). $\forall v$: α.
$inRegion(p,\ r) \Rightarrow inRegion(p,\ set(r,\ q,\ v))$

The last one states that *set* only adds one location:

axiom *notInSetRegion*:
$\forall p$: *location*. $\forall q$: *location*. $\forall r$: *region* (α). $\forall v$: α.
$p \neq q \Rightarrow \neg inRegion(p,\ r) \Rightarrow \neg inRegion(p,\ set(r,\ q,\ v))$

Note that regions are *finite* maps. However, all Why functions are total, including function *get*. This means that one can write $get(r,\ p)$ for any location p, including ones which are not actually in region r. However, the above axioms do not specify the result, and we cannot prove anything about $get(r,\ p)$ except that it exists. The actual domain of r is given by predicate *inRegion*.

Types We define the Why translation $why(\tau)$ of type τ as follows:

- $why(int) = int$;
- $why(bool) = bool$;
- $why(unit) = unit$;

- $why([\rho]) = location$;
- $why([\_]) = location$;
- $why(t\ (\tau_1, \cdots, \tau_n)) = t\ (why(\tau_1), \cdots, why(\tau_n))$.

This translation is recursive, and is well-founded.

Objects We define the Why translation $why(\mathcal{C})$ of class expression $\mathcal{C}$ as follows. Assume that $r_1: \mathcal{C}_1, \cdots, r_n: \mathcal{C}_n$ is the list of owned regions of $\mathcal{C}$, and that $f_1: \tau_1, \cdots, f_m: \tau_m$ is the list of fields of $\mathcal{C}$. Note that this assumes that type parameters and region parameters in class expression $\mathcal{C}$ have been substituted in $\mathcal{C}_1, \cdots, \mathcal{C}_n$ and $\tau_1, \cdots, \tau_m$. Then $why(\mathcal{C})$ is the following Why tuple type:

$$(region\ (why(\mathcal{C}_1)) \times \cdots \times region\ (why(\mathcal{C}_n)) \times why(\tau_1) \times \cdots \times why(\tau_m))$$

Again, this translation is recursive. It is well-founded if classes are not recursive. We tackle the recursive case in Section 5.3.

5.2.2 Expressions and Region Expressions

Logic Functions We already assumed a model $[\![f]\!]$ for each user-defined logic function. We know there is a model for logic functions *empty*, *get* and *set*, and for predicate *inRegion*. We assume that the application of a Why logic function f returns the value returned by the model $[\![f]\!]$.

Region Expressions Region binders in programs, i.e. region parameters of functions and the **let region** statement, introduce these variables as references. Their type is $region\ (why(\mathcal{C}))$ where $\mathcal{C}$ is the class of the region.

Assume ρ is a well-typed region:

$$\Gamma \vdash \rho: \mathcal{C}$$

and assume some permissions $\overline{\Sigma}$. We define the Why translation $why(\rho)$ of ρ with respect to $\overline{\Sigma}$ as follows.

- $why(r) = !r$;
- if x has type $[\sigma]$, if $\sigma^{\emptyset}$, σ°, $\sigma^{\times}$ or σ^{G} is in $\overline{\Sigma}$ or if σ is transitively owned by a closed region (i.e. $\Gamma, \overline{\Sigma} \vdash_{\times} \sigma$), then $why(x.s)$ is:

  $proj_s(get(why(\sigma), x))$

  where $proj_s$ is the projection corresponding to owned region s in tuple $why(\mathcal{C})$;
- if x has type $[\sigma]$, if $\sigma' \multimap \sigma$ is in $\overline{\Sigma}$ for some region σ', then $why(x.s)$ is:

  $proj_r(get((\textbf{if}\ inRegion(x, why(\sigma'))\ \textbf{then}\ why(\sigma')\ \textbf{else}\ why(\sigma)), x))$

  Note that the duplication of $why(\sigma')$ can be avoided using a let-binding;

This translation is not always defined. In particular, $why(x.s)$ is not defined if there is no available permission on the region of x.

It is important to note that if ρ is being focused, then $why(\rho)$ denotes a region from which the only pointers that should be read are the ones which have not been focused. In other words, $why(\rho)$ denotes $\mathcal{H}_s(\rho)$, but there may be a location p for which $get(why(\rho), p)$ does not denote $\mathcal{H}_s(p)$. This happens if $\mathcal{H}_s(p?)$ is not ρ.

Expressions Assume e is a well-typed expression:

$$\Gamma, \overline{\Sigma} \vdash e\colon \tau$$

We define the Why translation $why(e)$ of e as follows:

- $why(c) = c$;
- $why(x) = x$;
- $why(e_1\ op\ e_2) = why(e_1)\ op\ why(e_2)$;
- $why(f\ arg_1 \cdots arg_n) = f\ (why(arg_1), \cdots, why(arg_n))$ where $why(arg) = why(\rho)$ if arg is a region parameter $[\rho]$, and $why(arg) = why(e)$ if arg is a regular parameter (e);
- $why(x.f)$ is defined below.

The translation $why(x.f)$ depends on the type of x and on available permissions. Typing rules ensures that x has type $[\rho]$ for some region ρ of class $\mathcal{C}$, of which f is a field. Then $why(x.f)$ is the projection $proj_f$ on the tuple component of $why(\mathcal{C})$ corresponding to f, of the object $obj(x, \rho)$ obtained by looking for x in region ρ. We now define $obj(x, \rho)$:

- if ρ°, $\rho^\times$ or ρ^G is in $\overline{\Sigma}$, then $obj(x, \rho)$ is $get(why(\rho), x)$;
- if $\sigma \multimap \rho$ is in $\overline{\Sigma}$, then $obj(x, \rho)$ is:
  if $inRegion(x, why(\sigma))$ **then** $obj(x, \sigma)$ **else** $get(why(\rho), x)$.

The main difficulty is selection $x.f$, as the object of x must be read from its active region. Usually the active region is the region given by the type of x. But this is not the case if x is being focused, in which case the active region may be the focus region, or even another one if another focus happened. Note that from a language design standpoint, it is possible to consider preventing the user from reading a pointer which is in a region being focused. Indeed, experiments tend to show that this kind of access is not useful very often.

Example In the following environment:

$$x\colon [y.s],\ y\colon [r],\ r\colon \mathcal{C}$$

and with permissions:

$$y.s^\times,\ r^\circ$$

then the translation of $x.f$ is:

$$\begin{array}{rcl} why(x.f) & = & proj_f(obj(x, y.s)) \\ & = & proj_f(get(why(y.s), x)) \\ & = & proj_f(get(proj_s(get(!r, y)), x)) \end{array}$$

where $proj_f$ is the projection on the f field component of the class tuple of x, and $proj_s$ is the projection on the s owned region component of the class tuple of $\mathcal{C}$.

The expression we obtain corresponds to the following access in our separated model:

$$\mathcal{H}_s(r)(y)(s)(x)(f)$$

5.2.3 Logic

Assume an environment which, given a region variable r, states whether r is a *program region* (i.e. a region bound as a function region parameter or a **let region** statement) or a *logic region* (i.e. a region bound using a term or predicate parameter, or a $\forall$**region** or $\exists$**region** quantifier).

Region Terms Assume RT is a well-typed region term. We define the Why translation $why(RT)$ of RT as follows:

- $why(r@\textbf{here}) = !r$ if r is a program region;
- $why(r@L) = !r@L$ if r is a program region and $L \neq$ **here**;
- $why(r@\textbf{here}) = r$ if r is a logic region (typing ensures the label is **here**);
- $why(get(RT,\ LT,\ s))$ is the projection on the s component of: $get(why(RT),\ why(LT))$.

Terms Assume LT is a well-typed term. We define the Why translation $why(LT)$ of LT as follows:

- $why(c) = c$;
- $why(x) = x$;
- $why(LT_1\ termop\ LT_2) = why(LT_1)\ termop\ why(LT_2)$;
- $why(f\ larg_1 \cdots larg_n) = f\ (why(larg_1), \cdots, why(larg_n))$ where $why(larg) = why(RT)$ if $larg$ is a region parameter $[RT]$, and $why(larg) = why(LT)$ if $larg$ is a regular parameter (LT);
- $get(RT,\ LT,\ f)$ is the projection on the f component of $get(why(RT),\ why(LT))$.

Predicates Assume P is a well-typed predicate. We define the Why translation $why(P)$ of P as follows:

- $why(P_1 \wedge P_2) = why(P_1) \wedge why(P_2)$;
- $why(P_1 \vee P_2) = why(P_1) \vee why(P_2)$;
- $why(P_1 \Rightarrow P_2) = why(P_1) \Rightarrow why(P_2)$;
- $why(P_1 \iff P_2) = why(P_1) \iff why(P_2)$;
- $why(\neg P) = \neg why(P)$;
- $why(\mathsf{T}) = \mathsf{T}$;
- $why(\mathsf{F}) = \mathsf{F}$;
- $why(LT_1 = LT_2) = (why(LT_1) = why(LT_2))$;
- $why(LT_1 \neq LT_2) = (why(LT_1) \neq why(LT_2))$;
- $why(LT_1 > LT_2) = (why(LT_1) > why(LT_2))$;

- $why(LT_1 \geq LT_2) = (why(LT_1) \geq why(LT_2))$;
- $why(\forall x\colon \tau.\ P) = \forall x\colon why(\tau).\ why(P)$;
- $why(\exists x\colon \tau.\ P) = \exists x\colon why(\tau).\ why(P)$;
- $why(\forall \textbf{region}\ r\colon \mathcal{C}.\ P) = \forall r\colon region\ (why(\mathcal{C})).\ why(P)$;
- $why(\exists \textbf{region}\ r\colon \mathcal{C}.\ P) = \exists r\colon region\ (why(\mathcal{C})).\ why(P)$;
- $why(p\ larg_1\ \cdots\ larg_n) = p(why(larg_1), \cdots, why(larg_n))$;
- $why(LT \in RT) = inRegion(why(LT),\ why(RT))$.

This is pretty straightforward.

5.2.4 Statements

Assigning Any Region Assume some typing environment Γ and some permissions $\overline{\Sigma}$. We define operation $assign(\rho,\ w)$ where ρ is a Capucine region (i.e. r or $x.s$) and w is any why expression. This operation ensures the value of ρ is changed to w.

- $assign(r,\ w) =$

  $r := w$

- If x has type $[\sigma]$ and $\sigma^{\emptyset}$, σ°, $\sigma^{\times}$ or σ^{G} is in $\overline{\Sigma}$ or if σ is transitively owned by a closed region (i.e. $\Gamma,\ \overline{\Sigma} \vdash_{\times} \sigma$), then $assign(x.s,\ w) =$

  $assign(\sigma,\ set(why(\sigma),\ x,\ set_s(get(why(\sigma),\ x),\ w)))$

  where $set_s(o,\ w)$ sets the s field of an object tuple o to w.

- If x has type $[\sigma]$ and $\sigma' \multimap \sigma$ is in $\overline{\Sigma}$ then $assign(x.s,\ w) =$

  if $inRegion(x,\ why(\sigma'))$ **then**
  $assign(\sigma',\ set(why(\sigma'),\ x,\ set_s(get(why(\sigma'),\ x),\ w)))$
  else
  $assign(\sigma,\ set(why(\sigma),\ x,\ set_s(get(why(\sigma),\ x),\ w)))$

  For instance, consider the following environment:

$$r\colon \mathcal{C},\ r'\colon \mathcal{C}',\ x\colon [r],\ y\colon [r'],\ y\colon [x.s]$$

and the following permissions:

$$r'^{\times},\ r' \multimap x.s,\ r^{\circ}$$

where s is an owned region of class $\mathcal{C}$ and t is an owned region of the class of s. This situation is illustrated in Figure 5.1. Then $assign(z.t,\ w)$ is:

if $inRegion(z,\ !r')$ **then**
  $r' := set(!r',\ z,\ set_t(get(!r',\ z),\ w))$
else
  $r := set(!r,\ x,\ set_s(get(!r,\ x),\ set(proj_s(get(!r,\ x)),\ z,\ set_t(get(!r,\ z),\ w))))$

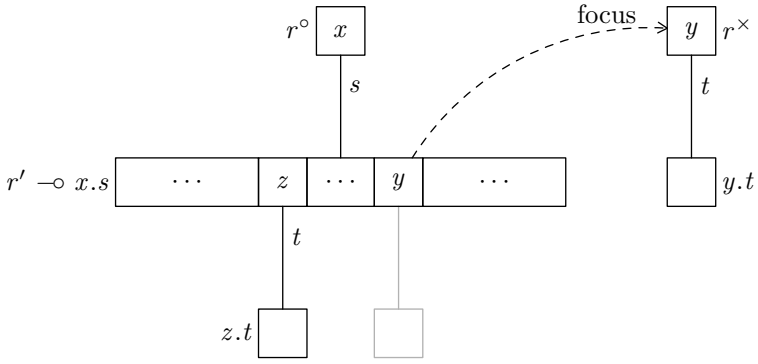

Figure 5.1: Example situation to illustrate the *assign* operation (y and z may be equal)

Translation of Statements Assume s is a well-typed statement. We define the Why translation $why(s,\ cont)$ of s with continuation $cont$, where $cont$ is a Why expression. First we define $why(S,\ cont)$ where S is a statement block recursively:

$$why((s;\ S),\ cont) = why(s,\ why(S,\ cont))$$

Then we consider each possible statement:

- $why($**let** $x = e,\ cont) =$

  let $x = why(e)$ **in**
  $cont$

- $why($**if** e **then** S_1 **else** $S_2,\ cont) =$

  (**if** $why(e)$ **then** $why(S_1,\ ())$ **else** $why(S_2,\ ())$);
  $cont$

- $why($**let** $x = f\ arg_1\ \cdots\ arg_m,\ cont) =$

  let $r_1 =$ **ref** $(why(\rho_1))$ **in**
  $\cdots$
  let $r_n =$ **ref** $(why(\rho_n))$ **in**
  let $x = f\ (whyarg(arg_1),\ \cdots,\ whyarg(arg_n))$ **in**
  $assign(\rho_1,\ !r_1)$;
  $\cdots$
  $assign(\rho_n,\ !r_n)$;
  $cont$

  where $arg_1, \cdots, arg_m$ restricted to region arguments is $[\rho_1], \cdots, [\rho_n]$, where $whyarg((e))$ is $why(e)$, and where $whyarg([\rho_i])$ is r_i.

- $why($**let region** r: $\mathcal{C},\ cont) =$

  let $r =$ **ref** $empty$ **in**
  $cont$

- $why($**let** $x =$ **new** $\mathcal{C}\ [\rho],\ cont) =$

let x = **any** *location* **in**
$assign(\rho,\ set(why(\rho),\ x,\ \textbf{any}\ why(\mathcal{C})))$;
$assign(x.r_1,\ empty)$;
...
$assign(x.r_n,\ empty)$;
cont

The pointer is initialized to some unknown value of *location* type. We do not actually require this value to be "the next fresh location" as in the model. Indeed, we only need to now that the location is fresh in the region it is added to. Here it is added to an empty region, so the location is necessarily fresh. Later it might be added into a group region ρ using adoption, and we would then be able to prove that the location is fresh in ρ as the location was not in ρ before the adoption.

Note that the only reason we initialize the object in ρ is to ensure the domain of ρ is $\{x\}$. Otherwise, from typing we could prove that $Dom(\rho) = \{x\}$ while $Dom(\rho)$ would actually be empty. For the same reason we initialize owned regions $r_1, \cdots, r_n$.

- $why(x.f \leftarrow e,\ cont) =$

  $assign(\rho,\ set(why(\rho),\ x,\ set_f(get(why(\rho),\ x),\ why(e))))$;
  cont

  where x has type $[\rho]$ and $set_f(x,\ y)$ allows to change the f component of tuple x to y.

- $why($**adopt** x: σ **as** $\rho,\ cont) =$

  $assign(\rho,\ set(why(\rho),\ x,\ get(why(\sigma),\ x)))$;
  cont

- $why($**focus** x: ρ **as** $\sigma,\ cont) =$

  $assign(\sigma,\ set(empty,\ x,\ get(why(\rho),\ x)))$;
  cont

  Note that as $why(\sigma) = empty$ before the focus operation, we could actually have used the same translation than for adoption.

- $why($**unfocus** x: σ **as** $\rho,\ cont) =$

  $assign(\rho,\ set(why(\rho),\ x,\ get(why(\sigma),\ x)))$;
  cont

  Note that this is exactly the same as adoption.

- $why($**pack** $x,\ cont) =$

  assert $why(Inv(x))$;
  cont

  where $Inv(x)$ is the invariant predicate of the class $\mathcal{C}$ of x, applied to $x.s_1, \cdots, x.s_n$ and $x.f_1, \cdots, x.f_m$ where $s_1, \cdots, s_n$ are the owned regions of $\mathcal{C}$ and $f_1, \cdots, f_m$ are the fields of $\mathcal{C}$.

- $why($**unpack** $x,\ cont) =$

  cont

- $why($**weaken empty** ρ, $cont) =$
  $cont$
- $why($**weaken single** ρ, $cont) =$
  $cont$
- $why($**assert** P, $cont) =$
  assert $why(P)$;
  $cont$
- $why($**label** L, $cont) =$
  label L;
  $cont$

Example: Function Call Translation of function calls is tricky, as regions which are passed as arguments are copied to temporary references, and then put back to their original location. Here is an example. Assume the following environment:

$$r\colon \mathcal{C},\ x\colon [r],\ y\colon [x.s]$$

and the following permissions:

$$r^{\circ},\ x.s^{G}$$

Then the translation of the following statement:

$$\textbf{let } z = f\ [x.s]\ (y)$$

with continuation *cont* is the following Why expression:

let $r_1 =$ **ref** $(proj_s(get(!r,\ x)))$ **in**
let $z = f\ (r_1,\ y)$ **in**
$r := set(!r,\ x,\ set_s(get(!r,\ x),\ !r_1))$;
$cont$

5.2.5 Declarations

Logic Types Capucine logic types declarations are of the form:

type $t\ (\alpha_1, \cdots, \alpha_n)$

They are kept as it in the translated Why programs.

Logic Functions and Predicates Capucine logic functions are declared as follows:

logic $f\ \overline{param}\colon \tau = LT$

They are translated into Why logic function declarations as follows:

logic $f\ \overline{why(param)}\colon why(\tau) = why(LT)$

where $why(param)$ is defined as:

- $why((x\colon \tau))$ is $x\colon why(\tau)$;

- $why([r\colon \mathcal{C}])$ is r: *region* $(why(\mathcal{C}))$;

Of course, if the body LT is omitted, it is omitted in the translated declaration as well.

Predicates are translated in exactly the same way. The only difference is that the body, if any, is a predicate and that there is no return type to translate.

Axioms Capucine axiom declarations are of the form:

axiom a: P

They are translated into Why axioms as follows:

axiom a: $why(P)$

Functions Capucine function declarations are of the form:

fun f $\overline{param}$: τ
  consumes $\overline{\Sigma}$
  produces $\overline{\Sigma}'$
  pre P
  post Q
{
  S;
  return e
}

Assume r is the set of references read by Why expression $why(S,\ why(e))$, and w is the set of references written by $why(S,\ why(e))$. Function f above is translated into a Why parameter and a Why function:

fun f $\overline{why'(param)}$: $why(\tau)$
  pre $why(P)$
  post $why(Q)$
  reads r
  writes w

fun f_{goal} $\overline{why'(param)}$: $why(\tau)$
  pre $why(P)$
  post $why(Q)$
{
  $why(S,\ why(e))$
}

where $why'(param)$ is defined as:

- $why'((x\colon \tau))$ is x: $why(\tau)$;
- $why'([r\colon \mathcal{C}])$ is r: **ref** $(region\ (why(\mathcal{C})))$;

Thus $why'(param)$ is the same as $why(param)$, except that region variables are references.

We translate the function into a parameter f which can be called from inside the body of any function. Function f_{goal} will not be called, but generates the required proof obligations. This is how we encode mutually recursive functions.

```
class PosInt
{
  value: int;
  invariant value > 0;
}

class Triple
{
  single Ra: PosInt;
  group Rbc: PosInt;
  a: [Ra];
  b: [Rbc];
  c: [Rbc];
}
```

Figure 5.2: Definition of classes *PosInt* and *Triple*

5.2.6 Illustration

We illustrate our encoding on a simple example. We consider class *PosInt*, with one *value* integer field which must be positive, and class *Triple* with three pointer fields of class *PosInt*. Two of those fields may be aliased. We program a function which increments all pointers of the triple. Figure 5.2 contains the Capucine code defining both classes, and Figure 5.3 contains the Capucine code of our incrementation function *incrTriple*, fully annotated and with no syntactic sugar.

Translation We now construct, step by step, the Why code corresponding to the translation of function *incrTriple*. Here is the header of the function, including the post-condition:

```
fun incrTriple
  (r: ref (region (region (int) × region (int) × location × location × location)),
   x: location): unit
  post
    proj_value(get(proj_Ra(get(!r, x)), proj_a(get(!r, x)))) =
      proj_value(get(proj_Ra(get(!r@pre, x)), proj_a(get(!r@pre, x)))) + 1 ∧
    proj_value(get(proj_Rbc(get(!r, x)), proj_b(get(!r, x)))) =
      proj_value(get(proj_Rbc(get(!r@pre, x)), proj_b(get(!r@pre, x)))) + 1 ∧
    proj_value(get(proj_Rbc(get(!r, x)), proj_c(get(!r, x)))) =
      proj_value(get(proj_Rbc(get(!r@pre, x)), proj_c(get(!r@pre, x)))) + 1
```

Note that $proj_{value}$ is actually the identity function, as the tuple corresponding to class *Long* has only one integer element. Similarly, $set_{value}(x, y)$ is y. We continue with the translation of the incrementation of $x.a.value$ and the **pack** statement:

```
{
  let a = proj_a(get(!r, x)) in
  r :=
    set(!r, x,
      set_Ra(get(!r, x),
```

```
fun incrTriple [r: Triple] (x: [r]): unit
  consumes r^×
  produces r^×
  post
    get(get(r, x, Ra), get(r, x, a), value) =
      get(get(r, x, Ra), get(r, x, a), value)@pre + 1 ∧
    get(get(r, x, Rbc), get(r, x, b), value) =
      get(get(r, x, Rbc), get(r, x, b), value)@pre + 1 ∧
    get(get(r, x, Rbc), get(r, x, c), value) =
      get(get(r, x, Rbc), get(r, x, c), value)@pre + 1
{
  unpack x;

  let a = x.a;
  unpack a;
  a.value ← a.value + 1;
  pack a;

  let b = x.b;
  let region Fb: PosInt;
  focus b: r.Rbc as Fb;
  unpack b;
  b.value ← b.value + 1;
  pack b;
  unfocus b: Fb as r.Rbc;

  let c = x.c;
  if b ≠ c then
  {
    let region Fc: PosInt;
    focus c: r.Rbc as Fc;
    unpack c;
    c.value ← c.value + 1;
    pack c;
    unfocus c: Fc as r.Rbc;
  }
  else {}

  pack x;
}
```

Figure 5.3: Function *incrTriple*

$set(proj_{Ra}(get(!r,\ x)),\ a,$
$\quad set_{value}($
$\quad\quad get(proj_{Ra}(get(!r,\ x)),\ a),$
$\quad\quad proj_{value}(get(proj_{Ra}(get(!r,\ x)),\ a))\ +\ 1))));$
assert $proj_{value}(get(proj_{Ra}(get(!r,\ x)),\ a))\ >\ 0;$

The assignment to r comes from the assignment to $a.value$. The **assert** statement comes from the **pack** statement. We continue with the focusing of $x.b$:

let $b = proj_b(get(!r,\ x))$ **in**
let $Fb =$ **ref** $empty$ **in**
$Fb := set(empty,\ b,\ get(proj_{Rbc}(get(!r,\ x)),\ b));$

Next is the assignment to $x.b.value$ and the **pack** statement:

$Fb := set(!Fb,\ b,\ set_{value}(get(!Fb,\ b),\ proj_{value}(get(!Fb,\ b))\ +\ 1));$
assert $proj_{value}(get(!Fb,\ b))\ >\ 0;$

The incrementation of $x.b$ ends with the unfocus:

$r := set(!r,\ x,\ set_{Rbc}(get(!r,\ x),\ set(proj_{Rbc}(get(!r,\ x)),\ b,\ get(!Fb,\ b))));$

The remaining of the function is the conditional assignment to $x.c.value$, which is similar to the one to $x.b.value$, so we do not show it here.

Proof Obligations There are several proof obligations:

- after the assignment to $x.a.value$, it is still positive (invariant of $x.a$);
- after the assignment to $x.b.value$, it is still positive (invariant of $x.b$);
- after the assignment to $x.c.value$, it is still positive (invariant of $x.c$);
- if $x.b = x.c$, then $x.a.value$, $x.b.value$ and $x.c.value$ all have been incremented by one;
- if $x.b \neq x.c$, then $x.a.value$, $x.b.value$ and $x.c.value$ all have been incremented by one.

Here is the proof obligations stating that the post-condition holds if $x.b = x.c$:

$\forall r$: $(int\ region \times int\ region \times location \times location \times location)\ region.$
$\forall a$: $location.$
$a = proj_a(get(r,\ x)) \Rightarrow$
$\forall r_1$: $(int\ region \times int\ region \times location \times location \times location)\ region.$
$r_1 =$
$\quad set(r,\ x,$
$\quad\quad set_{Ra}(get(r,\ x),$
$\quad\quad\quad set(proj_{Ra}(get(r,\ x)),\ a,$
$\quad\quad\quad\quad set_{value}($
$\quad\quad\quad\quad\quad get(proj_{Ra}(get(r,\ x)),\ a),$
$\quad\quad\quad\quad\quad proj_{value}(get(proj_{Ra}(get(r,\ x)),\ a))\ +\ 1)))) \Rightarrow$
$\forall b$: $location.$
$b = proj_b(get(r_1,\ x)) \Rightarrow$
$\forall Fb$: $int\ region.$
$Fb = empty \Rightarrow$

$\forall Fb_1$: *int region*.
$Fb_1 = set(empty,\ b,\ get(proj_{Rbc}(get(r_1,\ x)),\ b)) \Rightarrow$
$\forall Fb_2$: *int region*.
$Fb_2 = set(Fb_1,\ b,\ set_{value}(get(Fb_1,\ b),\ proj_{value}(get(Fb_1,\ b)) + 1)) \Rightarrow$
$\forall r_2$: (*int region* $\times$ *int region* $\times$ *location* $\times$ *location* $\times$ *location*) *region*.
$r_2 = set(r_1,\ x,\ set_{Rbc}(get(r_1,\ x),\ set(proj_{Rbc}(get(r_1,\ x)),\ b,\ get(Fb_2,\ b)))) \Rightarrow$
$\forall c$: *location*.
$c = proj_c(get(r_2,\ x)) \Rightarrow$
$b = c \Rightarrow$
$proj_{value}(get(proj_{Ra}(get(r_2,\ x)),\ proj_a(get(r_2,\ x)))) =$
$\quad proj_{value}(get(proj_{Ra}(get(r,\ x)),\ proj_a(get(r,\ x)))) + 1 \wedge$
$proj_{value}(get(proj_{Rbc}(get(r_2,\ x)),\ proj_b(get(r_2,\ x)))) =$
$\quad proj_{value}(get(proj_{Rbc}(get(r,\ x)),\ proj_b(get(r,\ x)))) + 1 \wedge$
$proj_{value}(get(proj_{Rbc}(get(r_2,\ x)),\ proj_c(get(r_2,\ x)))) =$
$\quad proj_{value}(get(proj_{Rbc}(get(r,\ x)),\ proj_c(get(r,\ x)))) + 1$

Automatic provers can prove this logic formula easily. It is only a matter of substituting all equalities, rewriting axioms regarding *get* and *set* and unfolding the definitions of the projections and tuple makers such as $proj_{value}$ and set_{value}.

However, this logic formula, while stating a simple fact, is heavy and hard to read. While the axiom instantiations are simple, they are quite numerous. This translation do not take enough advantage of the separation information granted by typing. Indeed, when modifying regions $x.Ra$ and $x.Rbc$, it is the whole region of x which is modified. So if we modify $x.Ra$ and then we modify $x.Rbc$, to access the value of $x.Ra$ we have to use the fact that they are two different fields of a tuple, and thus that modifying $x.Rbc$ does not modify $x.Ra$. In the next sections we explore ways to use type information to produce simpler proof obligations.

5.2.7 Soundness

In order to prove that our transformation from Capucine programs to Why programs is sound, we apply a methodology similar to the one we used in Section 4.5.2. We define a relation between Capucine separated heaps $\mathcal{H}_s$ and Why heaps H_w. Then we prove theorems stating that expressions, terms, predicates and statements and their respective translation evaluate and reduce equally in related heaps.

In this section, we will often say that a separated heap $\mathcal{H}_s$ is coherent. By that we mean that $\mathcal{H}_s$ is in relation to a coherent intuitive heap $\mathcal{H}$.

Relation Assume a bijection between Capucine locations p, and Why values of *location* type. We will allow ourself to use Capucine locations as Why values, and vice versa.

Let $\mathcal{C}$ be a Capucine class. We define relation $\mathcal{R}_o(o,\ v)$ between a separated object o and a Why value v of type $why(\mathcal{C})$, and relation $\mathcal{R}_r(g,\ v)$ between a separated region function g and a Why value v of type *region* $(why(\mathcal{C}))$, as the smallest relations such that:

- if $\mathcal{R}_r(g,\ v)$ then for all location p, if $g(p)$ is defined then:
  $\mathcal{R}_o(g(p),\ get(v,\ p))$;
- if $\mathcal{R}_o(o,\ v)$ then for all region name s, if $o(s)$ is defined then:
  $\mathcal{R}_r(o(s),\ proj_s(v))$;

- if $\mathcal{R}_o(o, v)$ then for all field name f, if $o(f)$ is defined then:
  $o(f) = proj_f(v)$.

We define relation $\mathcal{R}(\mathcal{H}_s, H_w)$ between separated Capucine heap $\mathcal{H}_s$ and Why heap H_w as the smallest relation such that:

- for all variable name x, $\mathcal{H}_s(x)$ is defined if, and only if $H_w(x)$ is defined and if so, then $\mathcal{H}_s(x) = H_w(x)$;
- for all region name r, $\mathcal{H}_s(r)$ is defined if, and only if $H_w(r, \textbf{here})$ is defined and if so, then $\mathcal{R}_r(\mathcal{H}_s(r), H_w(r, \textbf{here}))$.
- for all region name r, for all label L, $\mathcal{H}_s(L)(r)$ is defined if, and only if $H_w(r, L)$ is defined and if so, then $\mathcal{R}_l(\mathcal{H}_s(L)(r), H_w(r, L))$ where $\mathcal{R}_l$ is defined below.

We finally define relation $\mathcal{R}_l(g, v)$ between flattened region function g and Why region value v. Indeed, in the logic, functions and predicates expect *flattened* region functions and not *separated* region functions. A flattened region function is a function obtained using *lor* or lor_s (see paragraphs "From Programs to Logic" in Section 4.3 and Section 4.5.1, respectively). Relation $\mathcal{R}_l(g, v)$ is the smallest relation such that for all location p, for all field f, if there are locations $p_1, \cdots, p_n$ and region names $r_1, \cdots, r_n$ such that if for all $i > 1$, $v_i = proj_{r_i}(get(v_{i-1}, p_i))$ with $v_1 = v$ and $inRegion(p_i, v_{i-1})$, and if $inRegion(p, v_n)$, then this path to p is unique and $g(p)(f) = proj_f(get(v_n, p))$. In other words, $\mathcal{R}_l(g, v)$ if v is the smallest region value describing g.

Theorem 7 (Interpretation of Terms) Let LT be a Capucine term such that:

$$\Gamma \vdash LT : \tau$$

and RT be a Capucine region term such that:

$$\Gamma \vdash RT : \mathcal{C}$$

Let $\mathcal{H}_s$ be a separated Capucine heap coherent with respect to Γ and some permissions $\overline{\Sigma}$. Let H_w be a Why heap such that $\mathcal{R}(\mathcal{H}_s, H_w)$. If why term $why(LT)$ evaluates to value v in H_w then $(\!|LT|\!)_{loh_s(\mathcal{H}_s)} = v$. If why term $why(RT)$ evaluates to value v in H_w then $\mathcal{R}_l((\!|RT|\!)_{loh_s(\mathcal{H}_s)}, v)$.

Proof. By induction on RT and LT. More precisely, our induction hypothesis is the following. Let x be either a well-typed region term or a well-typed term. Then for all sub-region-term RT of x, and for all sub-term LT of x, RT and LT verify the theorem statement.

First, we consider the cases where x is a term.

- c

  Constants are trivially evaluated as the same values.

- x

  $\mathcal{L}_s(\mathcal{H}_s)(x)$ is, by construction, equal to $\mathcal{H}_s(x)$. $(\!|x|\!)_{loh_s(\mathcal{H}_s)}$ is thus equal to $\mathcal{H}_s(x)$, which is equal to $H_w(x)$ by $\mathcal{R}$. And x evaluates to $H_w(x)$ in H_w.

- LT_1 *termop* LT_2

  We apply the induction hypothesis on LT_1 and LT_2. Operations *termop* then evaluate equally.

- $f\ \overline{larg}$

  For each argument *larg*, if *larg* is of the form $[RT]$ then we apply the induction hypothesis on RT, else *larg* is of the form (LT) and we apply the induction hypothesis on LT. The application of $[\![f]\!]$ thus yields the same result value in both models.

- $get(RT,\ LT,\ f)$

  We apply the induction hypothesis on RT and LT. The term is evaluated to $(\!|RT|\!)_{\mathcal{L}_s}((\!|LT|\!)_{\mathcal{L}_s})(f)$ where $\mathcal{L}_s = loh_s(\mathcal{H}_s)$ in the separated model, and is translated to $proj_f(get(why(RT),\ why(LT)))$. The region variable r appearing in RT is either bound using $\forall$**region** or $\exists$**region**, or is a focus-free program region. In both cases we use the definition of $\mathcal{R}_l$ to show that:

  $$(\!|RT|\!)_{\mathcal{L}_s}((\!|LT|\!)_{\mathcal{L}_s})(f) = proj_f(get(why(RT),\ why(LT)))$$

  In the case that r is a program region, the fact that it is focus-free is fundamental to obtain that active regions are the ones in which we actually read pointers.

Then we consider the cases where x is a region term.

- $r@L$

  First, assume L is **here**. By definition of $\mathcal{R}$ we have:

  $$\mathcal{R}_r(\mathcal{H}_s(r),\ H_w(r,\ \textbf{here}))$$

  From this we show that:

  $$\mathcal{R}_l(lor_s(\mathcal{H}_s(r)),\ H_w(r))$$

  Indeed, let p be a location in the flattening of $\mathcal{H}_s(r)$. By construction of the flattening, there is a unique path to p in $\mathcal{H}_s$, by following active regions. We use this unique path to p to prove $\mathcal{R}_l$ for pointer p. For all field f, by $\mathcal{R}_r$ we have:

  $$\mathcal{H}_s(r)(p)(f) = proj_f(get(H_w(r,\ \textbf{here}),\ p))$$

  This proves $\mathcal{R}_l(lor_s(\mathcal{H}_s(r)),\ H_w(r))$.

  Then assume L is not **here**. We use a similar reasoning but using:

  $$\mathcal{R}_l(\mathcal{H}_s(L)(r),\ H_w(r,\ L))$$

- $get(RT,\ LT,\ s)$

  This case is basically a combination of the $get(RT,\ LT,\ f)$ case and the s case. Remember that there is no pointer being focused in RT, as logic regions are reconstructed entirely with lor_s.

  We apply the induction hypothesis on RT and LT. The region term is evaluated to $lor_s((\!|RT|\!)_{\mathcal{L}_s}((\!|LT|\!)_{\mathcal{L}_s})(s))$ where $\mathcal{L}_s = loh_s(\mathcal{H}_s)$ in the separated model, and is translated to $proj_s(get(why(RT),\ why(LT)))$. We have:

  $$\mathcal{R}_r((\!|RT|\!)_{\mathcal{L}_s}((\!|LT|\!)_{\mathcal{L}_s})(s),\ proj_r(get(why(RT),\ why(LT))))$$

and we show:

$$\mathcal{R}_l(lor_s((\!|RT|\!)_{\mathcal{L}_s}((\!|LT|\!)_{\mathcal{L}_s})(s)),\ proj_r(get(why(RT),\ why(LT))))$$

as we did for the r case.

□

Theorem 8 (Interpretation of Predicates) Let P be a Capucine predicate such that:

$$\Gamma \vdash P$$

Let $\mathcal{H}_s$ be a separated Capucine heap coherent with respect to Γ and some permissions $\overline{\Sigma}$. Let H_w be a Why heap such that $\mathcal{R}(\mathcal{H}_s,\ H_w)$. Why predicate $why(P)$ holds in H_w if, and only if $(\!|P|\!)_{loh_s(\mathcal{H}_s)}$ holds.

Proof. By induction on P. We apply Theorem 7 on terms and region terms. In particular for the $LT \in RT$ case we use the fact that definition of $\mathcal{R}_l$ uses the *inRegion* predicate. □

We now state an intermediary theorem which will be implicitely used in the proofs for interpretation of expressions and statements. This theorem states that if all necessary permissions are available on region ρ, then $why(\rho)$ is well-defined.

Theorem 9 (Interpretation of Regions) Assume ρ is a well-typed region: $\Gamma \vdash \rho\colon \mathcal{C}$. Assume some permissions $\overline{\Sigma}$. Assume for all $x\colon [\sigma]$ in Γ, $\sigma^{\emptyset}$ is not in $\overline{\Sigma}$. If $\Gamma,\ \overline{\Sigma} \vdash \rho$ then $why(\rho)$ is defined.

Proof. By induction on $\Gamma,\ \overline{\Sigma} \vdash \rho$.

If $\rho = r$, then $why(\rho)$ is defined and is $!r$. If $\rho = x.s$, then by $\Gamma,\ \overline{\Sigma} \vdash \rho$, variable x has type $[\sigma]$ for some region σ.

If x is in the open part of the region tree, i.e. $\Gamma,\ \overline{\Sigma} \vdash_\circ \sigma$, then $why(\sigma)$ is defined by induction on the derivation of $\Gamma,\ \overline{\Sigma} \vdash_\circ \sigma$, and thus $why(x.s)$ is defined.

Assume x is not in the open part of the region tree. It is thus in the closed part, i.e. $\Gamma,\ \overline{\Sigma} \vdash_\times \sigma$. We prove that $why(x.s)$ is defined by induction on the derivation of $\Gamma,\ \overline{\Sigma} \vdash_\times \sigma$.

- If x is below the border, i.e. $\sigma = y.s'$, $y\colon [\sigma']$ and $\Gamma,\ \overline{\Sigma} \vdash_\times \sigma'$, then we apply the induction hypothesis on σ' to prove that $why(\sigma')$ is defined, and thus that $why(\sigma)$ is defined, and thus that $why(x.s)$ is defined.

- If x is at the border, then we have a permission $\sigma^{\emptyset}$, $\sigma^{\times}$, σ^{G} or $\sigma' \multimap \sigma$ for some σ', and we have $\Gamma,\ \overline{\Sigma} \vdash_\circ \sigma$, so $why(\sigma)$ is defined (by induction on the derivation of $\Gamma,\ \overline{\Sigma} \vdash_\circ \sigma$). If the permission is $\sigma^{\emptyset}$, σ°, $\sigma^{\times}$ or σ^{G}, then $why(\rho)$ is defined as:

  $$proj_r(get(why(\sigma),\ x))$$

  Else, the permission is $\sigma' \multimap \sigma$ and we have $\Gamma,\ \overline{\Sigma} \vdash \sigma'$. By induction, $why(\sigma')$ is defined. So $why(\rho)$ is defined as:

  $$proj_r(get((\textbf{if } inRegion(x,\ why(\sigma')) \textbf{ then } why(\sigma') \textbf{ else } why(\sigma)),\ x))$$

□

Here is an example which shows the importance of this theorem and the $\Gamma, \overline{\Sigma} \vdash \rho$ typing judgement:

```
class C
{
  single s: Long;
}

fun f [r: C] (x: [r]) (y: [x.s])
  consumes x.s°
{
  y.value ← 3;
}
```

We cannot translate the assignment to $y.value$ as we do not have any permission on r, the region of x. We do not know x is actually in r, or if r being focused. Typing judgement $\Gamma, \overline{\Sigma} \vdash x.s$ requires some permission on the region of x to solve this problem.

Theorem 10 (Interpretation of Expressions) Let e be a Capucine expression such that:

$$\Gamma, \overline{\Sigma} \vdash e : \tau$$

Let $\mathcal{H}_s$ be a separated Capucine heap coherent with respect to Γ and $\overline{\Sigma}$. Let H_w be a Why heap such that $\mathcal{R}(\mathcal{H}_s, H_w)$. We have:

$$H_w,\ why(e) \overset{\text{Why}}{\Longrightarrow} (\!|e|\!)_{\mathcal{H}_s},\ H_w$$

Proof. By induction on e. We consider each case for e.

- c

  Constants are trivially equally evaluated.

- x

  $\mathcal{H}_s(x)$ is equal to $H_w(x)$ by $\mathcal{R}$.

- $e_1\ op\ e_2$

  We apply the induction hypothesis on e_1 and e_2. Operations op then evaluate equally.

- $f\ \overline{arg}$

  For each argument arg, if arg is of the form (e) we apply the induction hypothesis on e.

  Else arg is of the form $[\rho]$, and is evaluated in the separated model as:

  $$lor_s{}'(\mathcal{H}_s(\rho), \mathcal{H}_s)$$

  and as $why(\rho)$ in the Why translation. We show that they are in relation $\mathcal{R}_l$ as we did for region terms RT, as the definition of $lor_s{}'$ is similar to the definition of lor_s. We use coherence to ensure that all locations accessible in $why(\rho)$ are actually defined in $lor_s{}'(\mathcal{H}_s(\rho), \mathcal{H}_s)$.

  Function $[\![f]\!]$ is then exactly the Why function f.

- $x.f$

  This expression evaluates, in the separated model, to $\mathcal{H}_s(\mathcal{H}_s(x))(f)$. How it evaluates in the Why translation depends on the permission on the region ρ of x, and we prove the property by induction on the definition of $obj(x, \rho)$.

  If the region ρ of $\mathcal{H}_s(x)$ is not being focused, then the Why translation is $proj_f(get(why(\rho), x))$. We use the definition of $\mathcal{R}$ and we conclude.

  If region ρ is being focused, i.e. permission $\sigma \multimap \rho$ is available for some region σ, then the Why translation is:

  $$proj_f(\textbf{if } inRegion(x, \sigma) \textbf{ then } obj(x, \sigma) \textbf{ else } get(why(\rho), x))$$

  If x is in σ, we apply the induction hypothesis on $obj(x, \sigma)$. It x is not in σ, by item 8 of coherence we obtain that the active region of x is ρ. We use the definition of $\mathcal{R}$ and we conclude.

□

Theorem 11 (Progress) Let s be a Capucine statement such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ s\ \{\overline{\Sigma}'\}, \Gamma'$$

Let $\mathcal{H}_s$ be a separated Capucine heap coherent with respect to Γ and $\overline{\Sigma}$. Let H_w be a Why heap such that $\mathcal{R}(\mathcal{H}_s, H_w)$. Assume all proof obligations have been proved. Either the Capucine program diverges:

$$\mathcal{H}_s, s \Longrightarrow \infty$$

or there is a separated Capucine heap $\mathcal{H}_s{}'$ and some Why heap $H_w{}'$ such that $\mathcal{R}(\mathcal{H}_s{}', H_w{}')$ and:

$$H_w, why(s, unit) \overset{\text{Why}}{\Longrightarrow} (), H_w{}'$$

$$\mathcal{H}_s, s \Longrightarrow \mathcal{H}_s{}'$$

Proof. All proof obligations have been proved, so by Theorem 6, either:

$$H_w, why(s, unit) \overset{\text{Why}}{\Longrightarrow} \infty$$

or there is $H_w{}''$ such that:

$$H_w, why(s, unit) \overset{\text{Why}}{\Longrightarrow} (), H_w{}''$$

However, we do not necessarily have $\mathcal{R}(\mathcal{H}_s{}', H_w{}'')$. Indeed, the Why program is non-deterministic. For instance, black boxes such as **any** τ may return any value of type τ. This is the case for allocation. The idea is that when several non-deterministic choices are possible, we show that there is one which corresponds to the reduction in the Capucine model. Otherwise we choose $H_w{}' = H_w{}''$ and we show $\mathcal{R}(\mathcal{H}_s{}', H_w{}')$.

We first prove the statement of the theorem for some sequence $s_1; s_2$. Because proof obligations have been proven, for all Why heap which verify the pre-condition of $why(s_1, unit)$, executing $why(s_1, unit)$ either diverges or return a Why heap which verifies the pre-condition

of $why(s_2, unit)$. By induction, s_1 either diverges or terminates. If it diverges, $s_1; s_2$ diverges. Otherwise, the heap obtained after executing s_1 is in relation with a Why heap obtained by executing $why(s_1, unit)$ and we can apply the induction hypothesis on s_2. We generalize to any sequence $s_1; \cdots; s_n$ by induction.

First, we assume that $why(s, unit)$ terminates. We prove Theorem 11 by induction on the derivation of the reduction of $why(s, unit)$. We examine each possible case for s.

- **let** $x = e$

  We apply Theorem 10 on e to show that the value given to x is the same in the separated model and the Why interpretation.

- **if** e **then** S_1 **else** S_2

  We apply Theorem 10 on e. If it evaluates to *true*, we apply the induction hypothesis on $why(S_1, unit)$. Else we apply it on $why(S_2, unit)$.

- **let** $x = f\ \overline{arg}$

  Function f is translated into Why parameter f as well as Why function f_{goal}. Why parameter f modifies the heap non-deterministically in several possible ways. One of them is the one provided by Why function f_{goal}, as proof obligations ensure that f_{goal} verifies its contract, which is the contract of Why parameter f. We use this one.

  Let H_{w2} be the Why heap obtained after the bindings of Why variables $r_1, \cdots, r_n$. The way these variables are initialized allows parameters given to f to be in relation with the one given to their version in separated heap $\mathcal{H}_{s2}$ of the separated execution rule. We can thus execute the Why version of the body of f in H_{w2} and obtain H_{w3}, which is in relation with $\mathcal{H}_{s3}$ in the separated execution rule by induction. The fact that the body reduces shows that the pre-condition holds in H_{w2} and that the post-condition holds in H_{w3}, and we apply Theorem 8 to show that it also holds in the separated model. Let $H_w{}'$ be the Why heap obtained after the $assign(\rho_i, !r_i)$ operations. The way these assignments is done allows us to conclude that $H_w{}'$ is in relation with the separated heap $\mathcal{H}_s{}'$ obtained by reducing the call in the separated model.

  Let's be more precise about $\mathcal{H}_{s2}$ and H_{w2}. In the Why translation, references r_i are given initial values $why(\rho_i)$. In the separated model, regions r_i are assigned to $\mathcal{H}_s(\rho_i)$. Their pointers' active region is now r_i, because the typing rule implies that $\rho_i \neq \rho_j$ for all $i \neq j$. By $\mathcal{R}(\mathcal{H}_s, H_w)$, they are the same region values and thus $\mathcal{R}(\mathcal{H}_{s2}, H_{w2})$ holds. We use a similar reasoning to prove $\mathcal{R}(\mathcal{H}_s{}', H_w{}')$, by also applying Theorem 10 for the return value and using the fact that $root(\rho_i) \neq root(\rho_j)$ for all $i \neq j$ as we did in the proof for the $\mathcal{R}$ relation.

- **let region** r: $\mathcal{C}$

  Statement always reduce in both model. New region r is empty in both cases, and thus $\mathcal{R}_r(\mathcal{H}_s{}'(r), H_w{}'(r))$ holds.

- **let** $x =$ **new** $\mathcal{C}$ $[\rho]$

  We bind x to the next fresh location p and p to the empty object in the separated version of ρ. In the Why translation, p is defined by a black box. One particular reduction of this black box is thus p, and we choose this one. The same goes for the initial value of the object in the Why translation, which is also initialized to a black

box, whose value we choose to be in relation $\mathcal{R}_o$ with the empty object of the separate reduction rule. Such a value exists.

- $x.f \leftarrow e$

  By construction of *assign*, as the region ρ of x has permission ρ° and is thus not being focused. We apply Theorem 10 on e.

- **focus** x: ρ **as** σ

  Let $p = (\!|x|\!)_{\mathcal{H}_s}$, which is also equal to $H_w(x)$ by Theorem 10. Permission ρ^G is consumed, so region ρ is not being focused. Permission $\sigma^\emptyset$ is consumed, so σ is empty before the operation. By definition of *assign*, location p in σ is set to its value in ρ. Its active region is now σ, and thus the two obtained heaps are thus in relation.

- **adopt** x: σ **as** ρ

  This is the same reasoning than the one we used for the **focus** operation, except that target region is group instead of empty, but the reduction rule only changes the value of location x.

- **unfocus** x: σ **as** ρ

  This is the same reasoning than the one we used for the **adopt** operation, except that target region is being focused, but the reduction rule only changes the value of location x which was in the target region of the anyway, so its value was of no importance.

- **pack** x

  The statement reduces if, and only if the invariant predicate on x holds. We apply Theorem 8 to obtain that the statement reduces in the separated model if, and only if it reduces in its Why translation. The heaps are not modified and thus are still in relation.

- **unpack** x, **weaken empty** x, **weaken single** x

  Statement always reduce and heap is not modified.

- **assert** P

  This is the same proof than for packing, except with predicate P instead of the invariant.

- **label** L

  In $H_w{}'$, all region references r are copied in $H_w{}'(r, L)$, which is in relation $\mathcal{R}_l$ with $\mathcal{H}_s{}'(L)(r)$.

If the Why program never terminates, then it diverges and we prove by coinduction that the Capucine program diverges. □

5.3 Support for Recursive Classes

Our translation cannot encode recursive classes such as:

class *List*
{

group *r*: *List*;
next: [*r*];
value: *int*;
}

Class *List* is recursive because its owned region *r* has class *List*. So *why*(*List*) is not defined.

One could consider removing the argument of the *region* type:

type *region*

Then *why*(*List*) would simply be (*region* × *location* × *int*). However, this solution is not sound when proving proof obligations using provers which do not assume that all types are inhabited. Logic function *get* is defined as:

logic *get* (*region*, *location*): α

Given a value of type *region* and a value of type *location*, this function can create a value of any type, including the false predicate. Here is an example of a Coq [Coq] script which illustrates this problem:

```
Parameter region: Type.

Parameter get:
  forall A: Type,
  region -> location -> A.

Parameter r: region.

Parameter l: location.

Theorem bad: False.
apply get.
apply r.
apply l.
Qed.
```

Parameter `region` is the translation of Why type *region*. Parameter `get` is the translation of Why logic function *get*. We assume a region `r` and a location `l` are available, which is the case in proof obligations. Theorem `bad` illustrates how `False` can be proven: we simply construct the Coq term `get False r l`.

We can, however, change our translation in a sound way, even when targetting provers such as Coq. A first solution is to introduce one Why type $region_C$ per Capucine class $\mathcal{C}$. Logic function *get* is then duplicated for each class $\mathcal{C}$, with type:

logic $get_{\mathcal{C}}$ ($region_{\mathcal{C}}$, *location*): $why(\mathcal{C})$

For instance, here is the *get* function for class *List*:

logic get_{List} ($region_{List}$, *location*): ($region_{List}$ × *location* × *int*)

The problem is that all other logic functions and predicates (*set*, *empty* and *inRegion*) must be specialized for each class in the same way as *get*. Axioms must be duplicated as well.

Another solution is to introduce a Why type C for each class C. Type C encapsulates the object tuple for class C. We introduce a function get_C from C to $why(C)$ to read an

encapsulated tuple and a function $make_C$ from $why(C)$ to C to make an encapsulated tuple. For instance, here is the encoding of class *List*:

type *EncapsulateList*

logic *getList* (*EncapsulateList*): (*region* (*EncapsulateList*) × *location* × *int*)
logic *makeList* ((*region* (*EncapsulateList*) × *location* × *int*)): *List*

axiom *getmakeList*:
$\forall x$: (*region* (*EncapsulateList*) × *location* × *int*).
$getList(makeList(x)) = x$

In Coq, we could model type *EncapsulateList* with an inductive type. We can then keep the polymorphic type α *region*, but the α is instanciated with type *EncapsulateList* instead of $why(List)$. This is obviously a better encoding of recursive classes.

Both solutions are compatible with the other extensions that we present in Section 5.4 and Section 5.5.

5.4 Simplify Singleton Regions

5.4.1 Singleton Maps Are Values

Proposal Our first proposal to simplify proof obligations is to encode singleton regions without a map. Ideally, we would represent singleton regions by one single value corresponding to the object tuple stored in the region. The way singleton regions r were encoded in the previous section is using a map from a single location p to its object tuple o. To read o requires applying a logic function: $get(r, p)$. To change o to o' requires applying another logic function: $set(r, p, o')$. But if region r is singleton, then typing ensures that the only location we will read from r is its unique location p, which is thus irrelevant. However, singleton regions were actually empty before being singleton, and sometimes become group through the weakening operation.

What we actually propose is to represent each region ρ using three components:

- the ρ_P component, containing the unique location p of region ρ when ρ is singleton;
- the ρ_O component, containing the unique object tuple o of region ρ when ρ is singleton;
- the ρ_G component, containing the map representing ρ when ρ is group.

If the region is a region name and is thus represented as a reference, it is now represented as three references. If the region is owned and is thus one component of a tuple, it is now represented as three components of the tuple.

We then use permissions to know which component to read:

- if ρ is empty, we do not access it;
- if ρ is singleton, we access it through ρ_P and ρ_O;
- if ρ is group, we access it through ρ_G.

In particular, we need to be careful regarding statements which change the status of a region from empty to singleton, from empty to group or from singleton to group.

In the logic though, we do not change anything: regions are still maps. This is because there is no permission in the logic, and thus we would not know which component ρ_P, ρ_O or ρ_G to use when translating accesses, especially in axioms or predicate bodies.

Region Expressions Instead of having one single function $why(\rho)$, we define three translation functions $why_P(\rho)$, $why_O(\rho)$ and $why_G(\rho)$ corresponding to the three components of ρ. They are defined in a similar fashion than $why(\rho)$, except that they return the expected component P, O or G.

Expressions We change how expression $x.f$ is translated. Let ρ be the region of x. If $\rho^\circ\{f_1, \cdots, f_k\}$ $(k \geq 0)$ or $\rho^\times$ is available, the translation of $x.f$ is $proj_f(why_O(\rho))$. For instance, if ρ is a region name r, then $why(x.f)$ is $proj_f(!r_O)$, which is simpler than what we had before: $proj_f(get(!r, x))$. If, however, ρ^G or $\sigma \multimap \rho$ is available, we translate as before using $why_G(\rho)$.

Statements Instead of having one single function $assign(\rho, v)$, we define three functions $assign_P(\rho, v)$, $assign_O(\rho, v)$ and $assign_G(\rho, v)$ which respectively assign the ρ_P, ρ_O and ρ_G components of ρ. Now we can define the new translation for statements dealing with regions.

- $why($**let region** r: $\mathcal{C}$, $cont) =$

  let $r_P =$ **ref** (**any** $location$) **in**
  let $r_O =$ **ref** (**any** $why(\mathcal{C})$) **in**
  let $r_G =$ **ref** $empty$ **in**
  $cont$

  Black boxes construct unknown values, which will not be used as the region being empty, none of the three components will be accessed anyway.

- $why($**let** $x =$ **new** $\mathcal{C}$ $[\rho]$, $cont) =$

  let $x =$ **any** $location$ **in**
  $assign_P(\rho, x)$;
  $assign_O(\rho,$ **any** $why(\mathcal{C}))$;
  $cont$

  Interestingly enough, we could initialize x to $why_P(\rho)$, as this component of empty regions is initialized to a black box as well. Same goes for the initial object tuple.

- $why(x.f \leftarrow e, cont) =$

  $assign_O(\rho, set_f(why_O(\rho), why(e)))$;
  $cont$

  Region ρ is the region of x. This is quite simpler than before as we know that ρ is singleton, as permission $\rho^\circ\{f_1, \cdots, f_k\}$ $(k \geq 0)$ is available.

- $why($**adopt** x: σ **as** ρ, $cont) =$

  $assign_G(\rho, set(why_G(\rho), x, why_O(\sigma)))$;
  $cont$

  We simplify how we read the object of x, as its region σ is singleton.

- $why($**focus** x: ρ **as** σ, $cont) =$

  $assign_P(\sigma, x)$;
  $assign_O(\sigma, get(why_G(\rho), x))$;
  $cont$

  This is no longer an instance of adoption, as target region is singleton after the operation.

- $why($**unfocus** x: σ **as** ρ, $cont) =$

  $assign_G(\rho, set(why_G(\rho), x, why_O(\sigma)))$;
  $cont$

  Once again, this is exactly the same as adoption.

- $why($**weaken empty** ρ, $cont) =$

  $assign_G(\rho, empty)$;
  $cont$

  We assign the ρ_G component to **empty**, but it is not necessary as it already was **empty** (see the **let region** statement). It is simply a little easier for the user when ρ is a region parameter of the current function, as then the fact that ρ_G is **empty** would otherwise have to be given as a pre-condition.

- $why($**weaken single** ρ, $cont) =$

  $assign_G(\rho, set(empty, why_P(\rho), why_O(\rho)))$;
  $cont$

  Interestingly enough, while the weakening operation from singleton to group did not do anything before, it now has to transfer the ρ_P, ρ_O components into ρ_G.

Further Simplification If a region ρ is never singleton, we may omit the ρ_P and ρ_O components in the Why translation. Conversely, if ρ is always singleton, we may omit the ρ_G component.

Note that this can even be done if ρ is a parameter of a function. Consider function *incr*:

fun *incr* [r: *Long*] (x: [r]): *unit*
  consumes $r^\times$
  produces $r^\times$
{
  $r.value \leftarrow r.value + 1$;
}

Region r is always singleton during the execution of *incr*. Instead of expecting three reference arguments r_P, r_O and r_G, function *incr* expects only the first two. Why parameter *incr* becomes:

parameter *incr* (r_P: **ref** (*pointer*), r_O: **ref** (*int*), x: *pointer*): *unit*
  ...

Note that in practice, *incr* will always be called with $x = !r_P$. We know this from the fact that r is singleton. Another simplification could thus be to automatically remove argument r_P or x.

5.4.2 Illustration

We reuse the example of Figure 5.2 and Figure 5.3. We show how it is translated using our simplification of singleton regions, and then show the new proof obligation for the post-condition.

Translation We construct the Why code corresponding to the new translation of function *incrTriple*. Here is the header of the function, including the post-condition:

fun *incrTriple*
 (r: **ref** (*region* (*int* × *region* (*int*) × *location* × *location* × *location*)),
 x: *location*): *unit*
 post
 $proj_{value}(proj_{Ra}(get(!r,\ x))) =$
 $proj_{value}(proj_{Ra}(get(!r@\textbf{pre},\ x))) + 1 \wedge$
 $proj_{value}(get(proj_{Rbc}(get(!r,\ x)),\ proj_{b}(get(!r,\ x)))) =$
 $proj_{value}(get(proj_{Rbc}(get(!r@\textbf{pre},\ x)),\ proj_{b}(get(!r@\textbf{pre},\ x)))) + 1 \wedge$
 $proj_{value}(get(proj_{Rbc}(get(!r,\ x)),\ proj_{c}(get(!r,\ x)))) =$
 $proj_{value}(get(proj_{Rbc}(get(!r@\textbf{pre},\ x)),\ proj_{c}(get(!r@\textbf{pre},\ x)))) + 1$

Region $x.Ra$ is singleton during the whole execution of the function, and we will not need the P component as no adoption, focus or unfocus will be done. So we model $x.Ra$ using a simple integer. Region $x.Rb$ is group during the whole execution, so we model it as before using a region map. The other components are not needed. As you can see, the result is a simplification of the first part of the post-condition. We continue with the translation of the incrementation of $x.a.value$:

{
 let $a = proj_{a}(get(!r,\ x))$ **in**
 r :=
 $set(!r,\ x,$
 $set_{Ra}(get(!r,\ x),$
 $set_{value}($
 $proj_{Ra}(get(!r,\ x)),$
 $proj_{value}(proj_{Ra}(get(!r,\ x))) + 1)));$
 assert $proj_{value}(proj_{Ra}(get(!r,\ x))) > 0;$

Once again, the result is slightly simplified thanks to the fact that Ra is singleton. We continue with the focusing of $x.b$:

 let $b = proj_{b}(get(!r,\ x))$ **in**
 let Fb_P = **ref** (**any** *location*) **in**
 let Fb_O = **ref** (**any** *int*) **in**
 $Fb_P := b;$
 $Fb_O := get(proj_{Rbc}(get(!r,\ x)),\ b);$

As the target region Fb of the focus is always either empty or singleton, we do not need the G component of the region. We do need the P component for the unfocus which is to come. Next is the assignment to $x.b.value$ and the **pack** statement:

 $Fb_O := set_{value}(!Fb_O,\ proj_{value}(!Fb_O) + 1);$
 assert $proj_{value}(!Fb_O) > 0;$

The incrementation of $x.b$ ends with the unfocus:

$r := set(!r,\ x,\ set_{Rbc}(get(!r,\ x),\ set(proj_{Rbc}(get(!r,\ x)),\ b,\ !Fb_O)))$;

The remaining of the function is the conditional assignment to $x.c.value$, which is similar to the one to $x.b.value$, so we do not show it here.

Proof Obligations Here is the proof obligations stating that the post-condition holds if $x.b = x.c$:

$\forall r$: $(region \times int\ region \times location \times location \times location)\ region.$
$\forall a$: $location.$
$a = proj_a(get(r,\ x)) \Rightarrow$
$\forall r_1$: $(region \times int\ region \times location \times location \times location)\ region.$
$r_1 =$
$\quad set(r,\ x,$
$\quad\quad set_{Ra}(get(r,\ x),$
$\quad\quad\quad set_{value}($
$\quad\quad\quad\quad proj_{Ra}(get(r,\ x)),$
$\quad\quad\quad\quad proj_{value}(proj_{Ra}(get(r,\ x))) + 1))) \Rightarrow$
$\forall b$: $location.$
$b = proj_b(get(r_1,\ x)) \Rightarrow$
$\forall Fb_P$: $location.$
$\forall Fb_O$: $int.$
$Fb_P = b \Rightarrow$
$Fb_O = get(proj_{Rbc}(get(r_1,\ x)),\ b) \Rightarrow$
$\forall Fb_{O,1}$: $int.$
$Fb_{O,1} = set_{value}(Fb_O,\ proj_{value}(Fb_O) + 1) \Rightarrow$
$\forall r_2$: $(region \times int\ region \times location \times location \times location)\ region.$
$r_2 = set(r_1,\ x,\ set_{Rbc}(get(r_1,\ x),\ set(proj_{Rbc}(get(r_1,\ x)),\ b,\ Fb_{O,1}))) \Rightarrow$
$\forall c$: $location.$
$c = proj_c(get(r_2,\ x)) \Rightarrow$
$b = c \Rightarrow$
$proj_{value}(proj_{Ra}(get(r_2,\ x))) =$
$\quad proj_{value}(proj_{Ra}(get(r,\ x))) + 1\ \wedge$
$proj_{value}(get(proj_{Rbc}(get(r_2,\ x)),\ proj_b(get(r_2,\ x)))) =$
$\quad proj_{value}(get(proj_{Rbc}(get(r,\ x)),\ proj_b(get(r,\ x)))) + 1\ \wedge$
$proj_{value}(get(proj_{Rbc}(get(r_2,\ x)),\ proj_c(get(r_2,\ x)))) =$
$\quad proj_{value}(get(proj_{Rbc}(get(r,\ x)),\ proj_c(get(r,\ x)))) + 1$

As you can see, the proof obligation is simpler regarding singleton regions $x.Ra$ and Fb, as there is no need to access the value with *get* and to change it using *set*. It is thus more readable and we do not have to apply axioms defining *get* and *set* as much. The complexity regarding group regions is exactly the same.

5.4.3 Soundness

We shall not prove that our new translation is sound with as much details as we did for our original translation. We do, however, sketch the modifications that must be done to the proof. Relation $\mathcal{R}_r$ now takes three Why values v_P, v_O and v_G instead of only one. Each of

these value models one of the respective component of the region. Then we use permissions to know which component should be used to model the separated region function g. If the region is singleton, then g should be of singleton domain $\{v_P\}$, and $g(v_P)$ should be in relation $\mathcal{R}_o$ with v_O. If the region is group, then all location p in the domain of g should be in relation $\mathcal{R}_o$ with $get(p,\ v_G)$.

Knowing whether the considered region is singleton or group requires adding more arguments to the relations $\mathcal{R}_r$, $\mathcal{R}_o$ and $\mathcal{R}$. We need to keep track of the region being considered as well as permissions $\overline{\Sigma}$.

5.5 Flattening Regions for More Separation

5.5.1 Generalizing The Component-as-Array Model

The simplification we introduced for singleton regions is nice, but it does not introduce more static separation. In this section, we use the idea of the Burstall-Bornat component-as-array model [Bornat00], which we introduced in Section 2.4.3, and generalize it for Capucine. We introduce the intuition behind this generalization using the example of Figure 5.2 and Figure 5.3.

Reordering Heap Arguments In the separated heap $\mathcal{H}_s$ at the beginning of function *incrTriple*, the following paths are defined, where p is any location:

$\mathcal{H}_s(r)(p)(Ra)(p)(value)$: *int*
$\mathcal{H}_s(r)(p)(Rbc)(p)(value)$: *int*
$\mathcal{H}_s(r)(p)(a)$: *location*
$\mathcal{H}_s(r)(p)(b)$: *location*

We can *reorder* the arguments of function $\mathcal{H}_s$. In particular, here is an interesting reordering:

$\mathcal{H}_s{}'(r)(Ra)(value)(p)(p)$: *int*
$\mathcal{H}_s{}'(r)(Rbc)(value)(p)(p)$: *int*
$\mathcal{H}_s{}'(r)(a)(p)$: *location*
$\mathcal{H}_s{}'(r)(b)(p)$: *location*

This reordering has several consequences. One of them is that it is harder to achieve partial application $\mathcal{H}_s(r)(p)(Ra)$, for instance. However, it also has benefits which we explore in the next paragraphs.

Flattening Finite Arguments The first argument of function $\mathcal{H}_s{}'$ above is a region name. There is a *finite* number of region names, which can be computed from the body of *incrTriple*. Because this number is finite, we can split the function into one function per argument. Here, there is only one possible argument: region r. So we transform $\mathcal{H}_s{}'$ into one function $\mathcal{H}_{sr}$ with one less argument. The following paths are defined:

$\mathcal{H}_{sr}(Ra)(value)(p)(p)$
$\mathcal{H}_{sr}(Rbc)(value)(p)(p)$
$\mathcal{H}_{sr}(a)(p)$: *location*
$\mathcal{H}_{sr}(b)(p)$: *location*

The next argument can also only take a finite number of values: the names of owned regions and fields of the class of pointers in r, namely Ra, Rbc, a and b. So we can split function $\mathcal{H}_{sr}$ into four functions $\mathcal{H}_{sr,Ra}$, $\mathcal{H}_{sr,Rbc}$ and $\mathcal{H}_{sr,a}$ and $\mathcal{H}_{sr,b}$. The following paths are defined:

$\mathcal{H}_{sr,Ra}(value)(p)(p)$
$\mathcal{H}_{sr,Rbc}(value)(p)(p)$
$\mathcal{H}_{sr,a}(p)$: *location*
$\mathcal{H}_{sr,b}(p)$: *location*

Similarly, class *Long* has only one field *value*, and thus the first argument of functions $\mathcal{H}_{sr,Ra}$ and $\mathcal{H}_{sr,Rbc}$ is actually irrelevant. We delete this argument and rename these functions $\mathcal{H}_{sr,Ra,value}$ and $\mathcal{H}_{sr,Rbc,value}$, respectively. The following paths are defined:

$\mathcal{H}_{sr,Ra,value}(p)(p)$
$\mathcal{H}_{sr,Rbc,value}(p)(p)$
$\mathcal{H}_{sr,a}(p)$: *location*
$\mathcal{H}_{sr,b}(p)$: *location*

To sum up, what we did is simply:

- reorder the arguments of the heap function so that finite domains appear first;
- split functions with a finite domain into several functions with one less argument.

This is a generalization of the Burstall-Bornat component-as-array model. Indeed, in a naive model the heap is a function h of locations p and field names f. We reorder $h(p)(f)$ into $h(f)(p)$, and then flatten the function into $h_f(p)$, and we obtain the component-as-array model.

Singleton Regions This approach combines well with the simplification of singleton regions we introduced in Section 5.4. The three components P, O and G can be seen as an additional argument of the heap function. This argument can only take a finite number of values, namely P, O and G. Only the G component has a pointer argument, as it is a map.

In the above example, the following paths would be defined:

$\mathcal{H}_s(r)(P)$: *location*

$\mathcal{H}_s(r)(O)(Ra)(P)$: *location*
$\mathcal{H}_s(r)(O)(Ra)(O)(value)$: *int*
$\mathcal{H}_s(r)(O)(Ra)(G)(value)(p)$: *int*
$\mathcal{H}_s(r)(O)(Rbc)(P)$: *location*
$\mathcal{H}_s(r)(O)(Rbc)(O)(value)$: *int*
$\mathcal{H}_s(r)(O)(Rbc)(G)(value)(p)$: *int*
$\mathcal{H}_s(r)(O)(a)$: *location*
$\mathcal{H}_s(r)(O)(b)$: *location*
$\mathcal{H}_s(r)(O)(c)$: *location*

$\mathcal{H}_s(r)(G)(p)(Ra)(P)$: *location*
$\mathcal{H}_s(r)(G)(p)(Ra)(O)(value)$: *int*
$\mathcal{H}_s(r)(G)(p)(Ra)(G)(value)(p)$: *int*
$\mathcal{H}_s(r)(G)(p)(Rbc)(P)$: *location*
$\mathcal{H}_s(r)(G)(p)(Rbc)(O)(value)$: *int*

$\mathcal{H}_s(r)(G)(p)(Rbc)(G)(value)(p)$: *int*
$\mathcal{H}_s(r)(G)(p)(a)$: *location*
$\mathcal{H}_s(r)(G)(p)(b)$: *location*
$\mathcal{H}_s(r)(G)(p)(c)$: *location*

When a region is always singleton, we can remove the G component. When a region is always group, we can remove the P and O components. Region r is always singleton, region Ra is also always singleton, and region Rbc is always group. We are left with the following paths:

$\mathcal{H}_s(r)(P)$
$\mathcal{H}_s(r)(O)(Ra)(P)$
$\mathcal{H}_s(r)(O)(Ra)(O)(value)$
$\mathcal{H}_s(r)(O)(Rbc)(G)(value)(p)$
$\mathcal{H}_s(r)(O)(a)$
$\mathcal{H}_s(r)(O)(b)$
$\mathcal{H}_s(r)(O)(c)$

After applying our reorder-flatten transformation, we obtain:

$\mathcal{H}_{sr,P}$
$\mathcal{H}_{sr,O,Ra,P}$
$\mathcal{H}_{sr,O,Ra,O,value}$
$\mathcal{H}_{sr,O,Rbc,G,value}(p)$
$\mathcal{H}_{sr,O,a}$
$\mathcal{H}_{sr,O,b}$
$\mathcal{H}_{sr,O,c}$

In other words, we can encode the initial heap of the *incrTriple* function using six Why values:

- $\mathcal{H}_{sr,P}$, the unique pointer x of region r;
- $\mathcal{H}_{sr,O,Ra,P}$, the unique pointer $x.a$ of region $x.Ra$;
- $\mathcal{H}_{sr,O,Ra,O,value}$, the integer value of $x.a.value$;
- $\mathcal{H}_{sr,O,Rbc,G,value}$, the map from locations to integers for region $x.Rbc$;
- $\mathcal{H}_{sr,O,a}$, the location value of $x.a$;
- $\mathcal{H}_{sr,O,b}$, the location value of $x.b$.
- $\mathcal{H}_{sr,O,c}$, the location value of $x.c$.

5.5.2 Computing Prefix Trees

Compromises In Section 5.5.1, we gave the intuition for our new simplification. However, one important question is: *how deep should we flatten the heap ?* Flattening more is great to introduce more separation and using less arguments for functions, i.e. less maps in the Why translation. However, flattening introduces more reference variables. Assume for instance that there are n regions of pointers of class C, and class C has m integer fields. If we flatten as much as possible, we obtain $n \times m$ variables. If instead class C has m regions of pointers with o fields, we obtain $n \times m \times o$ variables. The number of variables thus grows quickly when the region tree becomes deeper. Another issue is if the tree is not statically finite, i.e.

if it involves recursive classes. Then there is an infinite number of ways we can flatten the tree.

Prefix Trees We introduce the notion of *prefix tree*. Prefix trees describe how deep into the heap tree we want to flatten. We first compute a prefix tree, and only then we translate the code according to this prefix tree. Here are some possible prefix trees:

- the empty tree, which says not to flatten at all, and from which we obtain the regular encoding of Section 5.2;
- the whole heap tree, which is only defined if there is no recursive class involved;
- the paths that are actually used by the function;
- the paths that are actually used by the function, with a fixed depth limit.

This fourth way of computing prefix trees is the one we are interested in. If, say, path $\mathcal{H}_s(r)(p)(value)$ is accessed by the function being translated, then it is a good idea to flatten this path as $\mathcal{H}_{sr,value}(p)$.

Formally, a prefix tree is a tree whose root is labeled $\mathcal{H}_s$. This root can be flattened. If so, we obtain n children, with n being the number of root regions involved in the translation. Each child is then labeled with a region name. Each region node can also be flattened, and then children labeled with the name of each owned region and field are added to the region node. Each owned region node can be flattened in the same fashion.

Computing Prefix Trees To compute paths which are actually used by function bodies, we add each used path in the code to the tree by flattening all nodes involved in the path. A path is a sequence of a region name, followed by some owned region names, possibly ended by a field name. Each expression $x.f$ appearing in the body adds the path to $x.f$, computing from the type of x and permissions, to the prefix tree. For instance, if x has type $[y.s]$ and y has type $[r]$, the path r, s, f is added to the prefix tree. The same goes if $x.f$ is assigned by a statement. Adoption, focus, unfocus allocation and region binders all add paths to the prefix tree as they involve creating, reading or writing regions.

Alternative Way of Flattening Instead of flattening a node by adding *all* its children into it, we can keep only the needed children. The remaining ones will be represented as before using a function. This limits the number of reference variables involved. We will not detail this approach in this thesis.

5.5.3 Using Prefix Trees When Translating

Expressions and Statements When accessing a path $\mathcal{H}_s(x_1)\cdots(x_n)$, we first split the path into two parts: $\mathcal{H}_s(x_1)\cdots(x_k)(y_1)\cdots(y_l)$ where $k+l=n$, and where either:

- path $\mathcal{H}_s(x_1)\cdots(x_k)$ leads to a leaf in the prefix tree;
- or $l=0$ and the whole path is in the prefix tree.

In other words, we look for the part of the path which is in the prefix tree.

If $l > 0$, $\mathcal{H}_s(x_1)\cdots(x_k)$ is represented using a reference variable and we read from this variable. We access the remainder $y_1, \cdots, y_l$ of the path using *get* and tuple projections. If we are assigning the path, we use the usual *set* and tuple makers.

If $l = 0$ and $\mathcal{H}_s(x_1)\cdots(x_k)$ does not lead to a leaf in the prefix tree, then the prefix tree is bigger than what is being read. In other words, $\mathcal{H}_s(x_1)\cdots(x_k)$ is not represented by a single reference variable, but by several. The following cases are to be considered:

- the node is being read as a whole, for instance as a region given to a logic function;
- the node is being copied into another node, for instance when focusing, adopting or unfocusing.

In the first case, we reconstruct the node from its parts. Path $\mathcal{H}_s(x_1)\cdots(x_k)$ leads to a node with several children, represented as several reference variables. We take all these variables and reconstruct one Why value representing $\mathcal{H}_s(x_1)\cdots(x_k)$. In the second case, we do not necessarily need to reconstruct the node if the target node into which we are copying is itself flattened. In this case, we copy each source child node into each target child node.

Function Calls Instead of using a Why parameter to encode function calls, we can *inline* the function as a black box with the substituted pre-condition, post-condition, reads clause and writes clause of the function. We can then compute the prefix tree using the inlined versions of these. This allows the prefix tree to be deeper. If, for instance, function f takes x as an argument and assigns $x.f$, and $f(y.g)$ is called, then $y.g$ can be flattened so that $y.g.f$ appears as a separated reference variable.

It is also worth noting that the reads and writes clause can be computed in a finer fashion using our flattening methodology. Say, for instance, function f takes argument x of type $[r]$, where r is a region containing *Triple* pointers. Say f only assigns $x.a.value$. With our direct encoding, there is only one reference: r, which must appear in the writes clause. With our flattening methodology, r can be flattened into several references, and we can put only the one corresponding to $x.a.value$ in the writes clause. Then from the outside world we know that when calling f, $x.b.value$ is not modified. This is quite powerful as the user does not have to put annotations in the post-condition to state that $x.b.value$ is unchanged. Instead, the reference corresponding to $x.b.value$ is simply not assigned.

Region Arguments of Logic Functions and Predicates In axioms, logic functions and predicates, there is no assignment, and thus the need for flattening is lesser. Actually, flattening in the logic has bad properties. Consider function f:

logic f [r: *Triple*] (x: [r]): *int*

Should region r be flattened? If so, how deep should it be flattened? It not only depends on the definition of f, which may not even be provided; but it also depends on the code which applies f. Knowing whether r should be flattened is thus not modular. To be modular we need to choose one flattening at the definition of f, not after. Candidates for flattening include no flattening or full flattening. Full flattening is only possible when involved classes are not recursive. We could also choose to define several Why versions of f, for instance one for each program function.

What we have chosen in our particular implementation of Capucine is not to flatten logic regions at all. Regions given as arguments to logic functions and predicates have to

be reconstructed. It is a bit heavy sometimes, and it can be argued that a global analysis, although not modular, would lead to simpler proof obligations.

For instance, consider the following definition for class *List* and logic function *length*, which returns the length of a list:

```
class List
{
  group Rnext: List;
  next: option ([Rnext]);
}

logic length [r: List] (l: [r]): int
```

We choose not to flatten r at all, and the Why translation of the declaration of *length* is thus:

logic *length* (r: $region_{List}$, l: *pointer*): *int*

However, if *length* is always applied on lists which are flattened once, i.e. their region is represented as two references, one for *Rnext* and one for *next*, then it would be a good idea to change the Why declaration to:

logic *length* (*Rnext*: $region_{List}$, *next*: *pointer*, l: *pointer*): *int*

Indeed, with this translation we do not have to reconstruct r from *Rnext* and *next* each time we apply *length*.

5.5.4 Illustration

We once again construct the Why code corresponding to the translation of function *incrTriple*, which was introduced in Figure 5.2 and Figure 5.3. We also use the simplification of singleton regions.

Choosing The Prefix Tree Instead of using names such as $\mathcal{H}_{sr,O,Ra,P}$, we choose to use simpler, well-chosen names:

- $\mathcal{H}_{sr,P}$, the unique pointer of r, is renamed into r_P;
- $\mathcal{H}_{sr,O,Ra,P}$, the unique pointer of $x.Ra$, is renamed into Ra_P;
- $\mathcal{H}_{sr,O,Ra,O,value}$, the value of $x.a.value$, is renamed into Ra_{value};
- $\mathcal{H}_{sr,O,Rbc,G,value}$, the region map of *Rbc*, is renamed into *Rbc*;
- $\mathcal{H}_{sr,O,a}$, the value of $x.a$, is renamed r_a;
- $\mathcal{H}_{sr,O,b}$, the value of $x.b$, is renamed r_b;
- $\mathcal{H}_{sr,O,c}$, the value of $x.c$, is renamed r_c;
- $\mathcal{H}_{sFb,P}$, the unique pointer of *Fb*, is renamed into Fb_P;
- $\mathcal{H}_{sFb,O,value}$ is renamed into Fb_{value}.

Translation Here is the header of the function:

fun *incrTriple*
(r_P: **ref** *location*,
Ra_P: **ref** *location*,
Ra_{value}: **ref** *int*,
Rbc: **ref** (*region* (*int*)),
r_a: **ref** *location*,
r_b: **ref** *location*,
r_c: **ref** *location*): *unit*
post
$!Ra_{value} = !Ra_{value}@\textbf{pre} + 1 \wedge$
$get(!Rbc, !r_b) = get(!Rbc@\textbf{pre}, !r_b@\textbf{pre}) \wedge$
$get(!Rbc, !r_c) = get(!Rbc@\textbf{pre}, !r_c@\textbf{pre})$

The benefits of our flattening methodology is already quite visible. We continue with the incrementation of $x.a.value$:

{
let $a = !r_a$ **in**
$Ra_{value} := !Ra_{value} + 1$;
assert $!Ra_{value} > 0$;

It cannot get simpler than this. Note that variable a is not actually useful here. We only keep it because it comes from the Capucine program. We continue with the focusing of $x.b$:

let $b = !r_b$ **in**
let Fb_P = **ref** (**any** *location*) **in**
let Fb_{value} = **ref** (**any** *int*) **in**
$Fb_P := b$;
$Fb_{value} := get(!Rbc, b)$;

Next is the assignment to $x.b.value$ and the **pack** statement:

$Fb_{value} := !Fb_{value} + 1$;
assert $!Fb_{value} > 0$;

Finally, here is the unfocus of $x.b$:

$Rbc := set(!Rbc, b, !Fb_{value})$;

The remaining of the function is similar.

Proof Obligation Here is the proof obligation stating that the post-condition holds if $x.b = x.c$:

$\forall r_P$: *location*.
$\forall Ra_P$: *location*.
$\forall Ra_{value}$: *int*.
$\forall$*Rbc*: (*region* (*int*)).
$\forall r_a$: *location*.
$\forall r_b$: *location*.
$\forall r_c$: *location*.
$\forall a$: *location*.

$a = r_a \Rightarrow$
$\forall Ra_{value,1}$: *int*.
$Ra_{value,1} = Ra_{value} + 1 \Rightarrow$
$\forall b$: *location*.
$b = r_b \Rightarrow$
$\forall Fb_P$: *location*.
$\forall Fb_{value}$: *int*.
$Fb_P = b \Rightarrow$
$Fb_{value} = get(Rbc,\ b) \Rightarrow$
$\forall Fb_{value,1}$: *int*.
$Fb_{value,1} = Fb_{value} + 1 \Rightarrow$
$\forall Rbc_1$: (*region* (*int*)) $\Rightarrow$
$Rbc_1 = set(Rbc,\ b,\ Fb_{value,1}) \Rightarrow$
$\forall c$: *location*.
$c = r_c \Rightarrow$
$b = c \Rightarrow$
$Ra_{value,1} = Ra_{value} + 1 \wedge$
$get(Rbc_1,\ r_b) = get(Rbc,\ r_b) + 1 \wedge$
$get(Rbc_1,\ r_c) = get(Rbc,\ r_c) + 1$

This proof obligation is arguably much more readable than the ones we had with previous encodings. It is also simpler to prove, as axioms *getSetEq* and *getSetNeq* have to be instanciated less often. In particular this helps theorem provers. It takes advantage of the separation information granted by typing. Indeed, when modifying $x.a.value$, for instance, the only reference which is modified is the one representing $x.a.value$. The reference representing $x.Rbc$, in particular, is left unchanged.

5.5.5 Soundness

Once again, we shall not prove that our new translation is sound with as much details as we did for our original translation. We simply remark that there is a simple bijection between the two representations, depending on the chosen prefix tree.

5.6 Experiments

In this section we give the results of experiments aimed at comparing whether automatic theorem provers are more efficient with the prefix-tree encoding of Section 5.5 or with the direct encoding of Section 5.2. We conduct our experiments using two theorem provers: Alt-Ergo [AltErgo] and Z3 [Z3], and on three programs: the *bounded arrays*, the *sparse arrays* and the *course and students* examples of Section 3.4.

We conduct our experiments on an Intel Core 2 Duo CPU E6850 at 3.00GHz with 3GB of memory. To lower the impact of memory swapping, we run the experiments once before running them again, and we only keep the second result. This way we ensure that the amount of memory required by the prover is actually free when running the second run.

Sparse Arrays Assertions The first result we observe is that some assertions that we put in the code of the *sparse arrays* example to help automatic provers are needed to prove the program entirely with the direct encoding, but are not needed with the prefix-tree encoding.

Bounded arrays: *arrayCreate*

invariant of the created array	0.120s	0.068s
post-condition	0.004s	0.000s

Bounded arrays: *arrayGet*

post-condition	0.000s	0.000s

Bounded arrays: *arraySet*

invariant of the modified array	0.004s	0.000s
post-condition (contents)	0.004s	0.004s
post-condition (length)	0.004s	trivial

Figure 5.4: Experiment results for bounded arrays

There are 3 of them. We thus have removed them when running the experiments on sparse arrays with the prefix-tree encoding.

Alt-Ergo We first verify our programs with the Alt-Ergo theorem prover. The protocol used is simple: we run Alt-Ergo on the file which contains all the goals. Alt-Ergo returns, for each goal, whether it was capable of proving it or not. It also returns the time taken to verify each verification condition.

Results are given in Figures 5.4, 5.6, and 5.7. The left column is a description of the verification condition. The middle column is the time taken by Alt-Ergo to prove the verification condition with the direct encoding. The right column is the time taken by Alt-Ergo to prove the verification condition with the prefix-tree encoding. Some proof obligations are so trivial with the prefix-tree encoding that they are discharged automatically by Why. Those are marked as "trivial" in the tables. Assertions which were removed from the sparse arrays example with the prefix-tree encoding are marked as "not used". Assertions which are not proved automatically by Alt-Ergo are marked as "not proved".

The total time taken to verify bounded arrays is 0.136s with the direct encoding, and 0.072s with the prefix-tree encoding. These number are too small to be significant. What *is* significant though is the fact that one proof obligation is so trivial with prefix trees that it is discharged by Why and is thus not even sent to the theorem prover. This is due to the separation of the heap between several variables. On the other hand, the proof obligation is proven almost instantly anyway even with the direct encoding.

The total time taken to verify the *course and students* example is not significant: it takes much less time (about 1.2s instead of about 14s) to prove obligations which are proved, but one proof obligation is no longer proved with the prefix-tree encoding. It is, however, proved using Z3 in less than 1s. This does not take into account the *main* function, which is proved automatically and in a quite fast fashion with both encoding. For this reason, we omitted the results for this function. Some proof obligations are not proved by Alt-Ergo in both cases. They are proved by another theorem prover. One proof obligation becomes trivial thanks to separation.

The total time taken to verify sparse arrays is more than 120s with the direct encoding, and is about 26s with the prefix-tree encoding. This is a pretty significant gain. The most impressive case is the verification condition for the invariant of the sparse array in function

Course and students: *createCourse*

invariant of the created course	0.168s	0.048s
post-condition	0.024s	0.012s

Course and students: *addStudent*

invariant of the created student	0.008s	0.004s
invariant of the modified course	not proved	not proved
post-condition (mark)	**12.917s**	**0.928s**
post-condition (count)	0.068s	0.024s
post-condition (sum)	0.080s	0.020s
post-condition (other marks)	0.228s	0.040s

Course and students: *changeMark*

invariant of the student	0.008s	0.004s
assertion to help provers	not proved	not proved
invariant of the course	**24.733s**	**not proved**
post-condition (mark)	0.080s	0.036s
post-condition (sum)	0.072s	0.032s
post-condition (count)	0.000s	trivial
post-condition (other marks)	0.004s	0.004s

Figure 5.5: Experiment results for course and students

sset: it takes about 27s with the direct encoding, and a third of a second with the prefix-tree encoding. What is also quite interesting is that many proof obligations are trivial with the prefix-tree encoding, and that assertions to help provers are not needed.

Z3 In order to check that the prefix-tree encoding does not only help Alt-Ergo but other theorem provers as well, we also measured the time Z3 takes to prove the verification conditions. Note that while Alt-Ergo give the time taken to prove each verification condition, Z3 does not. As a result, we choose to only measure the time taken for the whole program. It also follows that the two methods are not comparable, and thus that one should not deduce from these experiments that one prover is better than the other. However, it is sufficient to check whether the prefix-tree encoding helps automatic provers or not.

The protocol used is as follows. We run Z3 on each verification condition, one at a time, with a timeout of 10 seconds for each verification condition. If Z3 fails to prove a proof obligation after 10 seconds it will abort and go on to the next one. We measure the time it takes for Z3 to run on all proof obligations. We also count the number of timeouts. Note that when Z3 cannot prove a verification condition, it will continue to run until the timeout. We also tried to run the experiment with no timeout, but we aborted it after memory started swapping too much.

The results are the following:

- it takes about 72 seconds, including 7 timeouts, for the *course and students* example using the direct encoding;
- it takes about 31 seconds, including 3 timeouts, for the *course and students* example using the prefix-tree encoding;

Sparse arrays: *create*

pre-condition of *arrayCreate*	0.004s	0.004s
assertion to help provers	0.328s	not used
invariant of the created sparse array	**2.696s**	**0.032s**
post-condition (size)	0.184s	0.016s
post-condition (model)	1.044s	0.184s

Sparse arrays: *sget*

pre-condition of *arrayGet* on *a.idx*	0.008s	0.008s
pre-condition of *arrayGet* on *a.idx*	0.008s	0.004s
pre-condition of *arrayGet* on *a.back*	0.008s	0.008s
pre-condition of *arrayGet* on *a.value*	0.012s	0.008s
post-condition	0.044s	0.192s
post-condition	0.024s	0.156s
assertion to help provers	0.024s	not used
post-condition	0.040s	0.320s

Sparse arrays: *sset*

pre-condition of *arraySet* on *a.value*	0.008s	0.004s
pre-condition of *arraySet* on *a.value*	0.008s	0.008s
pre-condition of *arrayGet* on *a.idx*	0.040s	0.008s
pre-condition of *arrayGet* on *a.back*	0.040s	0.008s
assertion to help provers	0.020s	not used
invariant of the sparse array	0.468s	0.036s
post-condition	0.068s	trivial
post-condition	0.824s	1.420s
post-condition	0.084s	0.232s
assertion $a.n < a.size$	not proved	not proved
pre-condition of *arraySet* on *a.back*	0.636s	0.032s
pre-condition of *arraySet* on *a.back*	0.074s	0.012s
assertion to help provers	0.204s	not used
invariant of the sparse array	**6.004s**	**0.204s**
post-condition	0.152s	trivial
post-condition	**3.480s**	**2.000s**
post-condition	0.240s	0.216s
assertion $a.n < a.size$	not proved	not proved
pre-condition of *arraySet* on *a.back*	0.968s	0.036s
pre-condition of *arraySet* on *a.back*	0.120s	0.016s
assertion to help provers	1.028s	not used
invariant of the sparse array	**27.198s**	**0.364s**
post-condition	0.272s	trivial
post-condition	**6.740s**	**3.900s**
post-condition	0.448s	0.420s

Figure 5.6: Experiment results for sparse arrays (1/2)

Sparse arrays: *test*

pre-condition of *create*	0.004s	0.004s
pre-condition of *sget* on a	0.004s	0.004s
pre-condition of *sget* on b	0.008s	0.004s
assertion	0.008s	0.012s
assertion	0.012s	0.012s
pre-condition of *sget* on a	0.008s	0.004s
pre-condition of *sget* on b	0.004s	0.008s
assertion	0.008s	0.004s
assertion	0.008s	0.008s
pre-condition of *sget* on a	0.008s	0.004s
pre-condition of *sget* on b	0.008s	0.008s
assertion	**8.144s**	**2.372s**
assertion	**7.905s**	**2.324s**
pre-condition of *sget* on a	0.008s	0.008s
pre-condition of *sget* on b	0.008s	0.008s
assertion	**25.842s**	**6.000s**
assertion	**24.793s**	**5.620s**
pre-condition of *sget* on a	0.008s	0.008s
pre-condition of *sget* on b	0.012s	0.008s

Figure 5.7: Experiment results for sparse arrays (2/2)

- it takes about 156 seconds, including 10 timeouts, for the *sparse arrays* example using the direct encoding;
- it takes about 53 seconds, including 3 timeouts, for the *sparse arrays* example using the prefix-tree encoding.

Once again the prefix-tree encoding helps the automatic theorem prover, especially for the sparse arrays example where it is about 3 times faster and is capable of proving more verification conditions. Some of the other proof obligations cannot be proved automatically in a reasonable time with the direct encoding.

Conclusion Experiments show that the prefix-tree encoding greatly helps automatic theorem provers. Not only do they run faster, but they are also capable of proving more parts of the program in a reasonable time. We believe that this is due to the fact that the prover has to instanciate axioms for *get* and *set* much less often using the prefix-tree encoding.

Before conducting these experiments, we thought that the need to reconstruct regions when using them in the logic would have a negative impact on theorem provers. It does make proof obligations a little less readable. However, in practice it seems that this disadvantage is largely overcome by the advantage given by separation. It might be possible to build examples where this is not the case, but on our two practical examples the prefix-tree encoding is strictly superior.

We did not run the same experiments to test the encoding of Section 5.4. The reason is that there is no version of the implementation which features both simplified singleton regions and the prefix-tree encoding. There is a version of Capucine which features simplified singleton regions, but it is old and is not comparable with the current version. We believe

that, as the encoding of Section 5.4 further lowers the need to instanciate axioms, it can only be beneficial. However, the impact would probably be much lower than the one provided by the prefix-tree encoding.

There are very few existing work comparing experimentally the relative power of memory models. A notable recent exception is the work of Sascha Böhme and Michał Moskal in the context of the VCC program verifier and using Z3 [Böhme11]. They reach the same conclusion that models which include separation perform better than others.

5.7 Using Typing Information in Proofs

The example of Figure 5.2 and Figure 5.3 generates a proof obligation stating that the invariant of $x.a$ still holds after it has been incremented. The proof obligation, using the prefix tree encoding of Section 5.5.4 and after removing unused variables and useless hypotheses, is the following logic formula:

$$\begin{array}{l}\forall Ra_{value,1}\colon int.\\ Ra_{value,1} = Ra_{value} + 1 \Rightarrow\\ Ra_{value,1} > 0\end{array}$$

This formula cannot be proven. We miss the hypothesis that the invariant of $x.a$ held before the assignment. The invariant is *known by typing*. Indeed, permission $x.Ra^{\times}$ is available before the **unpack** a operation.

There are several facts that are known by typing and that we want to use in proof obligations. We review several of them in this section and we show how to introduce them in proof obligations using **assume** clauses. Soundness is given by coherence. Indeed, the Why heap is in relation with a separated heap, which is itself in relation with an intuitive coherent heap. From this we show that each property we introduce in the hypotheses of proof obligations using **assume** clauses actually hold.

5.7.1 Invariants

The first fact we want to use from typing is that the invariant of closed pointers hold. We introduce a new statement:

$$\textbf{use invariant } x$$

where x is a variable name. Typing requires that x is closed, i.e. that x: $[\rho]$ is in the current environment and either:

- $\rho^{\times}$ or ρ^{G} is in $\overline{\Sigma}$;
- or ρ is $y.s$, y: σ is in Γ, and σ is closed in $\overline{\Sigma}$.

This operation is translated to the following Why statement:

$$\textbf{assume } why(Inv(x))$$

where $Inv(x)$ is the invariant predicate of x applied to its owned regions and fields.

We can also allow **use invariant** x to be applied when x has type $[\rho]$ and permission $\sigma \multimap \rho$ is available for some σ, but then the operation translates to:

$$\textbf{assume } \neg inRegion(x,\ why(\sigma)) \Rightarrow why(Inv(x))$$

In other words, to use the invariant of x the user has to prove that x is not the pointer of ρ being focused.

Examples Section 3.4 contains several instances of the **use invariant** statement. In particular, functions *arraySet*, *sget* and *sset* of the *sparse arrays* example of Section 3.4.3 all use the invariant of their parameter.

5.7.2 Region Disjointness

In this section, we do not introduce a new statement, but we remark that we can sometimes deduce from available permissions that two regions ρ and σ are disjoint. In the next sections, we will use this to prove pointer disequalities, as well as to prove that some pointer is not in some region. Our condition on permissions is not necessary, but it is sufficient.

Theorem 12 (Sufficient Condition for Disjointness) Let ρ and σ be regions. Let Γ be an environment and $\overline{\Sigma}$ be a set of permissions. If $\overline{\Sigma} = \overline{\Sigma}', \Sigma_\rho, \Sigma_\sigma$ such that:

- Σ_ρ is either $\rho^\emptyset$, ρ°, $\rho^\times$ or ρ^G;
- Σ_σ is either $\sigma^\emptyset$, σ°, $\sigma^\times$ or σ^G,

then for all separated heap $\mathcal{H}_s$ coherent with respect to Γ and $\overline{\Sigma}$, then $Dom(\mathcal{H}_s(\rho))$ and $Dom(\mathcal{H}_s(\sigma))$ are disjoint.

Proof. Assume there is p in $Dom(\mathcal{H}_s(\rho))$ and in $Dom(\mathcal{H}_s(\sigma))$. Then by item 8 of coherence and permissions Σ_ρ and Σ_σ we have $\mathcal{H}_s(p?) = \rho$ and $\mathcal{H}_s(p?) = \sigma$, thus $\rho = \sigma$. Item 1 of coherence states that this is not possible, because we have the two permissions Σ_ρ and Σ_σ available. □

5.7.3 Pointer Region

Positive Another fact we want to use from typing is that if a pointer has type $[\rho]$, then it is actually in ρ. We introduce a new statement:

$$\textbf{use } x \in \rho$$

Typing requires that x: $[\rho]$ is in the current environment. This operation is translated into the following Why statement:

$$\textbf{assume } inRegion(x,\ why(\rho))$$

Negative We can also use from typing that a pointer is not in a given region. We introduce a new statement:

$$\textbf{use } x \notin \rho$$

Typing rules are the following:

$$\frac{x : [\sigma] \in \Gamma \qquad \rho^{\emptyset} \in \overline{\Sigma}}{\Gamma \vdash \{\overline{\Sigma}\}\ \textsf{use}\ x \notin \rho\ \{\overline{\Sigma}\},\ \Gamma}$$

$$\frac{x : [\sigma] \in \Gamma \qquad \overline{\Sigma} = \overline{\Sigma}',\ \Sigma_\sigma,\ \Sigma_\rho \qquad \Sigma_\sigma \in \{\sigma^\circ,\ \sigma^\times,\ \sigma^G\} \qquad \Sigma_\rho \in \{\rho^\circ,\ \rho^\times,\ \rho^G\}}{\Gamma \vdash \{\overline{\Sigma}\}\ \textsf{use}\ x \notin \rho\ \{\overline{\Sigma}\},\ \Gamma}$$

The first rule states that if ρ is empty, then x is not in ρ. The second rule uses the sufficient condition of Section 5.7.2 to conclude that the region of x is disjoint from ρ, and thus that x is not in ρ.

This operation is translated to the following Why statement:

$$\textsf{assume}\ \neg \mathit{inRegion}(x,\ \mathit{why}(\rho))$$

For instance, we can prove that a pointer is *fresh* in a given region. Consider the following program:

let $x =$ **new** *Long* $[\sigma]$;
use $x \notin \rho$;
adopt x: σ **as** ρ;

After the adoption, we know that x is in ρ, but that it was not before. As it is the only new pointer in ρ, we can prove that x is different than all pointers that were in ρ before the adoption.

Example The Course example of Section 3.4.5 illustrates statement **use** $x \in \rho$, in function *changeMark*.

5.7.4 Pointer Equalities

We can prove that two pointers are equal if they are in the same singleton region. We introduce a new statement:

$$\textsf{use}\ x = y$$

Typing requires x and y to be in the same region ρ. It requires a permission of the form $\rho^\circ\{f_1, \cdots, f_n\}$ (n may be zero) or $\rho^\times$.

This operation is translated to the following Why statement:

$$\textsf{assume}\ x = y$$

5.7.5 Pointer Disequalities

The **use** $x \notin \rho$ statement allows the user to prove pointer disequalities. For instance:

let region σ: *Long*;
let region ρ: *Long*;
let $x =$ **new** *Long* $[\sigma]$;
let $y =$ **new** *Long* $[\rho]$;
use $x \notin \rho$;

use $y \in \rho$;
assert $x \neq y$;

The last assertion can be proven. But it generates a proof obligation.

To avoid the proof obligation, we introduce a new statement:

$$\textbf{use } x \neq y$$

The typing rule is the following:

$$\frac{x : [\sigma] \in \Gamma \qquad y : [\rho] \in \Gamma \qquad \overline{\Sigma} = \overline{\Sigma}', \Sigma_\sigma, \Sigma_\rho \qquad \Sigma_\sigma \in \{\sigma^\circ, \sigma^\times, \sigma^G\} \qquad \Sigma_\rho \in \{\rho^\circ, \rho^\times, \rho^G\}}{\Gamma \vdash \{\overline{\Sigma}\}\ \textbf{use}\ x \neq y\ \{\overline{\Sigma}\}, \Gamma}$$

One again, we use the sufficient condition of Section 5.7.2 to prove that the regions of x and y are disjoint. We use the fact that x has type $[\sigma]$ to prove that x is in σ, and the fact that y has type $[\rho]$ to prove that y is in ρ. Because σ and ρ are disjoint, we thus have $x \neq y$.

This operation is translated to the following Why statement:

$$\textbf{assume } x \neq y$$

Note that if a pointer comparison $x = y$ or $x \neq y$ appears inside a predicate or an expression, and if we can prove from typing that the locations are always equal or are always different, then we can emit a warning to the user. For instance, in the following program:

let region r: *Long*;
let x = **new** *Long* $[r]$;
let region s: *Long*;
let y = **new** *Long* $[s]$;
if $x = y$ **then** $\cdots$

we can conclude from typing that $x \neq y$, and thus that the **then** branch of the **if** statement will never be executed.

Example The Course example of Section 3.4.5 illustrates statement **use** $x \neq y$, in function *addStudent*. The latter actually needs a small extension of the statement:

use
$\forall i$: *int*. $i \geq 0 \wedge i < c.count \Rightarrow$
$select(c.students, i) \neq s$

Indeed, x is no longer a variable but a term (here $select(c.students, i)$), which is itself deep inside a predicate. The idea is that we replace $select(c.students, i) \neq s$ by T, and then simplify the predicate using (among others) the following laws: $P \Rightarrow \mathsf{T}$ is equivalent to T, and $\forall x$: τ. T is also equivalent to T. Moreover, the typing rules for predicates must be changed slightly to be able to state that $select(c.students, i)$ has the more precise type $[c.Rstudents]$ instead of [_]. All these extensions are straightforward, and we shall not detail them in this thesis. They are available in the implementation of Capucine.

5.8 Conclusion

In this chapter, we showed how the separated model of Capucine can be encoded in the Why intermediate language. The Why tool then computes proof obligations. We showed that if those verification conditions are valid, then the Capucine program executes in the separated model of Capucine (Theorem 11). It thus also executes in the intuitive model and all assertions, all function contracts and all data invariants for closed objects are valid in this model.

We showed several methods to use region information from typing to produce simpler verification conditions. The fact that a region is singleton can be used to further simplify logic formulas. The separated model already benefits from separation between regions, and this reflects in logic formulas that we have to prove. We can use flattening techniques based on the region ownership tree to statically separate regions. Finally, we showed how the typing information that some invariant holds, or that a pointer belongs to a region, or that two regions are disjoint can be added as an hypothesis in verification conditions. Being able to use information from typing in proof obligations is an important feature of Capucine.

Chapter 6

Inference of Region Annotations

The Capucine language features many region annotations and operations. These may become tedious to write, especially for larger programs. In this chapter we explore how this burden can by lightened using inference. In Section 6.1 we motivate the need for inference using examples. In Section 6.2 we remark that there is no best way to infer region annotations, and we discuss which of them should actually be inferred. In Section 6.3 we present the core of the inference algorithm, and in Section 6.5 we propose extensions to further simplify the work of the programmer. Finally, in Section 6.6 we illustrate how inference helps the programmer using examples.

6.1 Need for Inference

Simple Example Simple C programs such as:

```
typedef struct {
  int value;
} long;

void incr(long* i)
{
  i->value = i->value + 1;
}
```

need region annotations in Capucine:

```
class long
{
  value: int;
}

fun incr [r: long] (i: [r]): unit
  consumes r^×
  produces r^×
{
  unpack i;
```

```
  i.value ← i.value + 1;
  pack i;
}
```

If the region r is group, **focus** and **unfocus** operations have to be inserted as well:

```
  let region s: long;
  focus i: r as s;
  unpack i;
  i.value ← i.value + 1;
  pack i;
  unfocus i: s as r;
```

All these annotations are needed just for one single assignment. It quickly becomes tedious to write them.

Most of these annotations are obvious. For instance, if $i.value$ is assigned, then i must be in an open region r. If r is closed, the **unpack** is obvious. If r is in a group region, the fact that a **focus** operation is needed is obvious.

Moreover, common patterns emerge, such as **let region** – **focus** – **unpack** – **pack** – **unfocus** illustrated above. Another one is allocation into a group region r:

```
  let region s: long;
  let i = new [s];
  pack i;
  adopt i in [r];
```

The **let region** – **pack** – **adopt** sequence is a common pattern.

These observations motivate the need for inference of region annotations in Capucine. The question is: what can we infer?

Candidates for Inference In an ideal world, we could infer every region annotation and be as close as possible to languages such as C. These annotations are:

- region binders, be they function arguments or bound using **let region**;
- regions of pointers, be they function arguments or allocated using **new**;
- **consumes** and **produces** clauses of functions;
- adoption, focus, unfocus, packing and unpacking operations.

In the next sections, we try and see which of these are good candidates for inference, and which should be better left for the user to write.

6.2 There Is No Principality

Principal Types An important property of type systems when considering them for inference is the existence of *principal types*. ML expressions, for instance, have a *most general* type for each expression. We call it the principal type of the expression. All other types can be derived from the principal type. For instance, the following OCaml expression:

```
fun x → x
```

Has types `int` $\rightarrow$ `int` and `bool` $\rightarrow$ `bool` (among others), but both of these types can be derived from instancing the more general type $\alpha \rightarrow \alpha$, which happens to be the principal type of the expression. Good inference algorithms infer principal types, thus ensuring that no expressivity is lost in the process.

Are Regions Separated In Capucine however, in general there is no principality. Consider the following pseudo-Capucine program:

fun *incrPair*(x: *long*; y: *long*)
{
  $x.value \leftarrow x.value + 1$;
  $y.value \leftarrow y.value + 1$;
}

We need to give regions to x and y. Shall we give the same region for both, or should they be in different, separated regions? Regions are polymorphic, and we could think that if we put x and y in two different regions, we could instantiate *incrPair* with the same region for x and y. The code would become:

fun *incrPair* [r: *long*, s: *long*] (x: [r], y: [s])
  consumes $r^\times$ $s^\times$
  produces $r^\times$ $s^\times$
...

But if a call to *incrPair* gives the same region ρ twice:

incrPair [ρ, ρ] (x, y)

then permission $\rho^\times$ is consumed twice, which is prevented by the type system. So this version of *incrPair* is not more general than the one where x and y share the same region.

Moreover, consider the same function *incrPair*, annotated with the following post-condition:

$$x.value = x.value@\textbf{pre} + 1$$

This post-condition only holds if $x \neq y$. One could thus argue that with this post-condition, pointers x and y must be separated in different regions. But, the problem of knowing whether pre- and post-conditions require pointers to be separated is undecidable. And it is not necessary for x and y to be in different regions to be different: they can be in the same group region.

Is The Invariant Needed The next issue is the **consumes** and **produces** clauses. Consider the following pseudo-Capucine program:

class *posint*
{
  value: *int*;
  invariant $value > 0$;
}

fun *incr* [r: *posint*] (x: [r])
  post $x.value > 0$

{
$x.value \leftarrow x.value + 1$;
}

Shall this function consume permission r°, $r^\times$ or r^G? Permission r° could be perceived as more general as the permission is needed anyway for the assignment. The responsibility of unpacking and packing would then fall to the caller. However, the invariant of x is needed to prove the post-condition, so permission $r^\times$ is actually required by *incr*. And of course, knowing whether the invariant is needed is undecidable.

Conclusion We have shown that there is no *best way* to choose the regions of pointer arguments, and the permissions on these regions. Notice that "best way" is another way of saying "principal". In Capucine, we thus choose not to try to infer regions of pointer arguments and **consumes** and **produces** clauses. From a language design point of view, this makes sense: these annotations are part of the *contract* of the function. They give information on the intent of the programmer.

6.3 Inferring Most Region Annotations

We have chosen not to infer what is visible from *outside* of function definitions: regions of pointer arguments, and **consumes** and **produces** clauses. From these annotations, we are capable of inferring some annotations which should be added *inside* the body of functions.

Forward Inference We state the problem as follows. Given available permissions $\overline{\Sigma}$, an environment Γ and a statement s, can we compute a sequence of statements S; s', a list of produced permissions $\overline{\Sigma}'$ and a new environment Γ' such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S;\ s'\ \{\overline{\Sigma}'\},\ \Gamma'$$

Moreover, S should not change the operational semantics of the program. More precisely, the program should return a heap in which well-typed expressions evaluate to unchanged results. Sequence S may only be composed of local binders (**let** or **let region**), adoptions, focus, unfocus, unpack, pack and weakening operations. Finally, if S contains let-bindings such as **let** $x = e$, then replacing all such variables x by their corresponding expression e in s' should result in s. Remember that there is no side-effect in expressions. We say that S; s' and s are *equivalent*.

We call this *forward inference* because we compute S given *available* permissions $\overline{\Sigma}$, instead of *produced* permissions $\overline{\Sigma}'$. Other forms of inference might also be interesting, such as:

- given $\overline{\Sigma}'$, compute S and $\overline{\Sigma}$ such that s; S is well-typed;
- given S, compute both $\overline{\Sigma}$ and $\overline{\Sigma}'$ while inserting new statements in S such that S is well-typed.

The former kind of inference would be *backward inference*. We think it is less predictable for the user and thus harder to work with in practice. The latter kind of inference would be the strongest. However, not only would it be even less predictable, it also implies being able to infer permissions which are consumed and produced by functions. We already explained why this was not what we wanted for Capucine.

Unification Our inference mechanism uses the usual ML inference algorithm based on unification. In particular, this solves the issue of finding substitutions σ when applying polymorphic functions. The unification mechanism is applied not only on type variables but also on region variables. If unsolvable constraints result from unification, the inference mechanism fails, with the exception on region variables described in the next paragraph.

Main Loop Our inference algorithm takes as input:

- a list of available permissions $\overline{\Sigma}_0$;
- an initial environment Γ_0;
- a sequence of statements S_0.

It may either fail or return a sequence S' equivalent to S_0 such that there is $\overline{\Sigma}'$ and Γ' such that:

$$\Gamma_0 \vdash \{\overline{\Sigma}_0\}\ S'\ \{\overline{\Sigma}'\}, \Gamma'$$

It uses the following variables:

- a sequence (represented as a list) of statements S;
- the list of currently available permissions $\overline{\Sigma}$;
- the current environment Γ;
- the result being built S'.

Here is the algorithm. The definition of *region constraints* and how to apply them is given in a later paragraph. The way S_{prod} is computed, and how branches are *joined*, are also given in later paragraphs.

1. Initialize S to S_0, $\overline{\Sigma}$ to $\overline{\Sigma}_0$, Γ to Γ_0 and S' to the empty list.
2. While S is not empty, do the following.
   (a) Let s be the first statement of S.
   (b) If s is **if** e **then** S_1 **else** S_2:
      i. call the algorithm recursively on S_1 and S_2, thus obtaining well-typed statements $S_1{}'$ and $S_2{}'$ respectively equivalent to S_1 and S_2;
      ii. using typing rule SBlock, compute permissions $\overline{\Sigma}_1$ and $\overline{\Sigma}_2$ produced by S_1 and S_2;
      iii. join $\overline{\Sigma}_1$ and $\overline{\Sigma}_2$ in $S_1{}'$ and $S_2{}'$, obtaining $S_1{}''$ and $S_2{}''$;
      iv. add **if** e **then** $S_1{}''$ **else** $S_2{}''$ at the end of S';
      v. remove s from S;
      vi. do not execute the rest of the current loop iteration.
   (c) Let rc be the region constraints of s with respect to Γ and $\overline{\Sigma}$.
   (d) If rc is not empty, apply region constraints rc.
   Else:
      i. compute the set of permissions $\overline{\Sigma}_s$ consumed by s;

ii. compute a sequence S_{prod} such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S_{prod}\ \{\overline{\Sigma}_s, \overline{\Sigma}_{rem}\}, \Gamma'$$

iii. change the value of Γ to Γ', and the value of $\overline{\Sigma}$ to $(\overline{\Sigma}_s, \overline{\Sigma}_{rem})$;

iv. add S_{prod}; s at the end of S';

v. remove s from S.

3. Return S'.

Compute Region Constraints The first step of our inference mechanism is to collect *region constraints* from statement s. A region constraint is a triple (e, σ, ρ), whose informal meaning is that expression e, of type $[\sigma]$, should actually be of type $[\rho]$.

Region constraints come from *unification failures* between regions. In Capucine, region constraint (e, σ, ρ) may be generated in any of the following circumstances:

- s is $x.f \leftarrow e$ and $x.f$ has type $[\rho]$;
- s is a function call and e is one of the parameters, whose expected type is $[\rho]$;
- s contains a sub-expression e' which is a logic function application, of which e is a parameter whose expected type is $[\rho]$.

Unification failure on regions may also happen in other circumstances. For instance, unification of *Array* ($[\rho]$) and *Array* ($[\sigma]$) leads to unification of ρ and σ. However, in general we do not have any expression e whose type $[\sigma]$ should be $[\rho]$. So instead of generating a region constraint, the inference mechanism just fails. Only in the three cases above do we have such an expression e.

For instance, consider the following Capucine program.

```
class Long
{
  value: int;
}

class Pair
{
  group Rleft: Long;
  group Rright: Long;
  left: [Rleft];
  right: [Rright];
}

fun f [r: Pair] (p: [r], i: [p.Rright]): unit
  ...

fun initPair [r: Pair] (p: [r]): unit
  consumes r°{left, right}, p.Rleft^∅, p.Rright^∅
  ...
{
  let region s: Long;
  let x = new Long [s];
```

$p.left \leftarrow x$; (1)
$f\ [r]\ (p,\ p.left)$; (2)
}

Line (1) generates region constraint $(x,\ s,\ p.Rleft)$, i.e. pointer x is in s but it really should now be in $p.Rleft$. Assuming it now is, line (2) generates region constraint $(p.left,\ p.Rleft,\ p.Rright)$.

Additionally we want to be able to infer a focus operation for x when $s = x.f \leftarrow e$ and the region of x is group. If x: $[\rho]$ is in Γ, and if ρ^G or $\sigma \multimap \rho$ is available or if $\rho = y.s$ is transitively owned by a closed region and s is defined as **group**, then ρ is group and a focus is needed. In this case, we bind a fresh region r using **let region** and we generate region constraint $(x,\ \rho,\ r)$.

Apply Region Constraints The next step of our inference mechanism is to apply region constraints. First, we apply a transformation which ensures that for all region constraint $(e,\ \sigma,\ \rho)$, expression e is already a variable. The transformation is the following. If e is already a variable, do nothing. Else:

- choose a fresh variable name x and add statement **let** $x = e$ to the S', the result sequence being built by the main loop;
- syntactically replace e by x in the current statement s, both in variable s of the main loop and in the first statement of variable S of the main loop (which is equal to s).

Ensuring that all expressions e are actually variables is necessary, as adoption, focus and unfocus operations only change the type of variables in the environment. For instance, consider the constraint generated by line (2) in the *initPair* example. The type of expression $p.left$ does not change from $[p.Rleft]$ to $[p.Rright]$ even if we adopt the pointer. However, if we insert **let** $x = p.left$ before and adopt x, then the type of x changes. Note that this transformation does not change the meaning of the program because expressions do not have side-effects.

We are left with region constraints of the form $(x,\ \sigma,\ \rho)$, with $\sigma \neq \rho$. Our goal is to insert an adoption, focus or unfocus operation ensuring that x is moved from σ to ρ. We choose the operation depending on available permissions on σ and ρ. Or, rather, on permissions *we could obtain* on σ and ρ.

To unfocus x from σ to ρ, we need permission $\sigma \multimap \rho$. The only operation (beside function calls) which produces $\sigma \multimap \rho$ is the focus operation. It is a property of our inference algorithm that the only way focus operations are inferred is by applying region constraints. So if $\sigma \multimap \rho$ is not available, we know that applying region constraint $(x,\ \sigma,\ \rho)$ should not be done using an unfocus operation. However, is $\sigma \multimap \rho$ is available, then we know that unfocusing is the way to go. To sum up, the operation we try to infer is **unfocus** x: σ **as** ρ if, and only if $\sigma \multimap \rho$ is in $\overline{\Sigma}$.

To focus x from σ to ρ, we need permission $\rho^{\emptyset}$. The only operations which produce $\rho^{\emptyset}$ are region binding and allocation, which we do not infer during the region constraint application phase. So if $\rho^{\emptyset}$ is not available, we shall not infer a focus operation to apply the region constraint. Instead we shall try to infer an adoption.

If $\rho^{\emptyset}$ is available, we have two choices: either focus into ρ, or weaken ρ to obtain ρ^G and then adopt into ρ. This choice depends on whether σ is group or not. Region σ is group if σ^G or $\sigma' \multimap \sigma$ is available for some region σ' *different than* σ, or if σ is a group region

transitively owned by a closed region. In all other cases, σ is singleton, and we do not wish to weaken it as a group region just to focus from it. So we choose to try to infer an adoption.

Once we have chosen which operation s' we want to try to infer (either adoption **adopt** x: σ **as** ρ, focus **focus** x: σ **as** ρ or unfocus **unfocus** x: σ **as** ρ), we add s' at the beginning of variable S of the main loop. In other words, we leave the work of producing the correct permissions needed by s' to the other parts of the inference mechanism.

If statement s generates several constraint, we insert their respective operation in an unspecified order. The reasoning behind this is that if applying the constraints in some order fails, then applying them in any order will fail, because it basically means that a pointer should be in two regions at the same time.

Let's illustrate region constraint application on function *initPair*. The first constraint (x, s, $p.Rleft$) is solved by inserting an adoption, as s is singleton and $p.Rleft$ is not being focused. Here is the intermediate body of function *initPair* up to line (1) after inference:

let region s: *Long*;
let x = **new** *Long* [s];
weaken empty $p.Rleft$;
pack x;
adopt x: s **as** $p.Rleft$;
$p.left \leftarrow x$;

The weakening and packing are added by another part of the inference mechanism. Now we have permissions $p.Rleft^G$ and $p.Rright^{\emptyset}$. So the second constraint ($p.left$, $p.Rleft$, $p.Rright$) is solved by a focus operation, after replacing $p.left$ by a variable:

let y = $p.left$;
focus y: $p.Rleft$ **as** $p.Rright$;
f [r] (p, y);

The function is now well-typed, assuming the **consumes** clause of f and the **produces** clause of *initPair* is:

$$p^{\circ}\{right\},\ p.Rright \multimap p.Rleft,\ p.Rright^{\times}$$

Note that if the **consumes** clause of *initPair* is the more normal $p^{\times}$, the inference mechanism will fail as there is no way to put the pointer of $p.Rright$ back into $p.Rleft$ without losing the permission on $p.Rright$, which is required to pack p.

Compute Permission Constraints The next step of our inference algorithm requires being able to compute the set of permissions $\overline{\Sigma}_s$ consumed by any statement s, given current permissions $\overline{\Sigma}$ and environment Γ, and given that s is not an **if** statement. The set $\overline{\Sigma}_s$ is given by typing rules of Figures 4.10 and 4.11 for statements. Here is a summary.

- Statements **let** $x = e$, **let region** r: $\mathcal{C}$, **assert** P and **label** L consume no permissions.
- Test **if** e **then** S_1 **else** S_2 is a special case which is handled separately.
- Function call **let** $x = f\ \overline{arg}$ consumes the permissions consumed by f, after having applied the substitution σ obtained by unification.
- Allocation **let** x = **new** $\mathcal{C}$ [ρ] consumes $\{\rho^{\emptyset}\}$.
- Focus **focus** x: ρ **as** σ consumes $\{\rho^G, \sigma^{\emptyset}\}$.

- Assignment $x.f \leftarrow e$ consumes an open permission on a region ρ such that x: $[\rho]$ is in Γ. If $\rho^\circ\{f_1, \cdots, f_n\}$ is available in $\overline{\Sigma}$, we define $\overline{\Sigma}_s = \{\rho^\circ\{f_1, \cdots, f_n\}\ \}$. Else, we can only obtain ρ° by unpacking ρ, so all fields are initialized, and we can thus define $\overline{\Sigma}_s = \{\rho^\circ\}$.
- Adoption **adopt** x: σ **as** ρ consumes $\{\sigma^\times, \rho^G\}$.
- Unfocus **unfocus** x: σ **as** ρ consumes $\{\sigma^\times, \sigma \multimap \rho\}$.
- Packing **pack** x, if x: $[\rho]$, ρ: $\mathcal{C}$ is in Γ, consumes $\{\textbf{single}(x, \mathcal{C})^\times, \textbf{group}(x, \mathcal{C})^G, \rho^\circ\}$.
- Unpacking **unpack** x, if x: $[\rho]$ is in Γ, consumes $\{\rho^\times\}$.
- Weakening **weaken empty** ρ consumes $\{\rho^\emptyset\}$.
- Weakening **weaken single** ρ consumes $\{\rho^\times\}$.

Solve Permission Constraints Now that we know which permissions $\overline{\Sigma}_s$ we have to produce, we look for S such that there is $\overline{\Sigma}'$ and Γ' such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S\ \{\overline{\Sigma}_s, \overline{\Sigma}'\}, \Gamma'$$

Our methodology is the following.

- First we define function *operations*(Γ, Σ) which returns the list of statements which produce Σ under the environment Γ. The list only contains statements which we are willing to insert as inferred operations.
- Then we define function *produce*$(\Gamma, \overline{\Sigma}, \Sigma)$ which returns a sequence which produces, among others, permission Σ from $\overline{\Sigma}$.
- Then we define function *producelist*$(\Gamma, \overline{\Sigma}, \overline{\Sigma}')$ which returns a sequence which produces, among others, the set of permissions $\overline{\Sigma}'$ from $\overline{\Sigma}$. Functions *produce* and *producelist* are mutually recursive.
- We are then capable of producing $\overline{\Sigma}_s$ using *produce*$(\Gamma, \overline{\Sigma}, \overline{\Sigma}_s)$.

Function *operations* We define function *operations*(Γ, Σ) by pattern-matching on Σ.

- *operations*$(\Gamma, \rho^\emptyset) = \emptyset$

  The only operation which can produce $\rho^\emptyset$ are **let region**, only if ρ is a region variable, and **new**. Region binders are only needed by **focus** and **new** operations. Section 6.5 discusses how region binders are inferred in these two cases. This is why *operation*$(\Gamma, \rho^\emptyset)$ returns the empty set of operations.

- *operations*$(\Gamma, \rho^\circ\{f_1, \cdots, f_k\}) = \emptyset$

  This assumes $k \geq 1$. Only **new** can produce such kind of permission, and we do not wish to infer **new** operations.

- *operations*$(\Gamma, \rho^\circ) = \{\textbf{unpack}\ x\}$

  This assumes x: $[\rho]$ is in Γ. If several variables of type $[\rho]$ are in Γ, we pick one at random, as their value is the same (the unique location of ρ).

- $\mathit{operations}(\Gamma,\, \rho^{\times}) = \{\textbf{pack}\ x\} \cup A$

  This assumes x: $[\rho]$ is in Γ. Operation set A is defined as:

  - $A = \emptyset$ if ρ is a region variable;
  - $A = \{\textbf{unpack}\ y\}$ if ρ is an owned region $y.s$.

- $\mathit{operations}(\Gamma,\, \rho^{G}) = \{\textbf{weaken empty}\ \rho, \textbf{weaken single}\ \rho\} \cup A \cup B$

  Operation set A is defined as:

  - $A = \emptyset$ if ρ is a region variable;
  - $A = \{\textbf{unpack}\ y\}$ if ρ is an owned region $y.s$.

  Operation set B is defined as the set of all **unfocus** x: σ **as** ρ where x: $[\sigma]$ is well-typed with respect to Γ. In practice, we may restrict this set to regions σ such that $\sigma \multimap \rho$ is available.

Function *produce* We define function $\mathit{produce}(\Gamma,\, \overline{\Sigma},\, \Sigma,\, \overline{\Sigma}_{no})$ using the following algorithm. Argument $\overline{\Sigma}_{no}$ stores the set of permissions which we are already trying to produce, to ensure the algorithm terminates. This argument is also added to *producelist*.

1. If Σ is in $\overline{\Sigma}$, return the empty sequence.
2. If Σ is in $\overline{\Sigma}_{no}$, fail.
3. Let *ops* be $\mathit{operations}(\Gamma,\, \Sigma)$.
4. For each s in *ops*, let $\overline{\Sigma}_0$ be the permissions consumed by s and try to compute: $S = \mathit{producelist}(\Gamma,\, \overline{\Sigma},\, \overline{\Sigma}_0,\, (\overline{\Sigma}_{no},\, \Sigma))$.
   (a) If it fails, continue with the next operation; if there is no more operation, fail.
   (b) Else return S; s.

We then define:

$$\mathit{produce}(\Gamma,\, \overline{\Sigma},\, \Sigma) = \mathit{produce}(\Gamma,\, \overline{\Sigma},\, \Sigma,\, \emptyset)$$

Function *producelist* We define function $\mathit{producelist}(\Gamma,\, \overline{\Sigma},\, \overline{\Sigma}',\, \overline{\Sigma}_{no})$ using the following algorithm.

1. If $\overline{\Sigma}'$ is empty, return the empty sequence.
2. Select any Σ from $\overline{\Sigma}' = \Sigma,\, \overline{\Sigma}_{rem}$.
3. Try to compute $S = \mathit{produce}(\Gamma,\, \overline{\Sigma},\, \Sigma,\, \overline{\Sigma}_{no})$.
4. If it fails, fail.
5. Let $\overline{\Sigma}_1$ and Γ_1 such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S\ \{\overline{\Sigma}_1\},\, \Gamma_1$$

6. Try to compute $S' = \mathit{producelist}(\Gamma_1,\, \overline{\Sigma}_1,\, \overline{\Sigma}_{rem},\, \overline{\Sigma}_{no})$.

7. If it fails, fail.
8. Let $\overline{\Sigma}_2$ and Γ_2 such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S;\ S'\ \{\overline{\Sigma}_2\},\ \Gamma_2$$

9. If Σ is in $\overline{\Sigma}_2$, return S; S'.
10. Else, fail.

We then define:

$$\mathit{producelist}(\Gamma,\ \overline{\Sigma},\ \overline{\Sigma}') = \mathit{producelist}(\Gamma,\ \overline{\Sigma},\ \overline{\Sigma}',\ \emptyset)$$

This function can also be used to produce the permissions of **consumes** clauses of functions.

Joining Branches We detail the case when s is an **if** e **then** S_1 **else** S_2 statement. We apply the inference algorithm recursively on S_1 and S_2, obtaining $S_1{}'$ which produces permissions $\overline{\Sigma}_1$ and $S_2{}'$ which produces permissions $\overline{\Sigma}_2$, and such that s is equivalent to $s' =$ **if** e **then** $S_1{}'$ **else** $S_2{}'$. However, s' is not well-typed if $\overline{\Sigma}_1 \neq \overline{\Sigma}_2$. We now show how we try to compute operations S_a and S_b such that there is $\overline{\Sigma}'$ such that:

$$\Gamma \vdash \{\overline{\Sigma}_1\}\ S_a\ \{\overline{\Sigma}'\}$$

$$\Gamma \vdash \{\overline{\Sigma}_2\}\ S_b\ \{\overline{\Sigma}'\}$$

This ensures that:

$$s'' = \textbf{if}\ e\ \textbf{then}\ S_1{}';\ S_a\ \textbf{else}\ S_2{}';\ S_b$$

is equivalent to s and is well-typed.

We consider each region ρ appearing in $\overline{\Sigma}_1$ or $\overline{\Sigma}_2$ *and* in Γ. Assume it appears in $\overline{\Sigma}_1$; the other case is symmetric. We consider each possible case $(\Sigma_1,\ \Sigma_2)$ where Σ_1 is the permission on ρ in $\overline{\Sigma}_1$ and Σ_2 is the permission on ρ in $\overline{\Sigma}_2$ if there is any, or $\bot$ if there is no such permission.

- If $\Sigma_1 = \Sigma_2$, do nothing.
- $\Sigma_1 = \rho^{\emptyset}$
  - $\Sigma_2 = \rho^{\circ}\{f_1, \cdots, f_k\}$ $(k \geq 0)$ or $\Sigma_2 = \rho^{\times}$
    Fail. We do not infer allocations.
  - $\Sigma_2 = \rho^{G}$
    Add **weaken empty** ρ at the end of S_a.
  - $\Sigma_2 = \sigma \multimap \rho$
    Add **weaken empty** ρ at the end of S_a, and add **unfocus** x: σ **as** ρ at the end of S_b, where x: $[\sigma]$ is in Γ. If there is no such x, fail.
  - $\Sigma_2 = \bot$
    Fail.

- $\Sigma_1 = \rho^\circ\{f_1, \cdots, f_k\}$ $(k \geq 1)$
  Fail.
- $\Sigma_1 = \rho^\circ$
  - $\Sigma_2 = \rho^\times$
    Add **unpack** x at the end of S_b, where x: $[\rho]$ is in Γ. If there is no such x, fail.
  - $\Sigma_2 = \rho^G$
    Add **pack** x; **weaken single** ρ at the end of S_a, where x: $[\rho]$ is in Γ. If there is no such x, fail. Ensure all owned permissions needed by the **pack** statement are available, by recursively applying the inference algorithm.
  - $\Sigma_2 = \sigma \multimap \rho$
    Do the same than for the $\Sigma_2 = \rho^G$ case, except also add **unfocus** x: σ **as** ρ at the end of S_b where x: $[\sigma]$ is in Γ. If there is no such x, fail. Ensure permission $\sigma^\times$ needed by the **unfocus** statement are available, by recursively applying the inference algorithm.
  - $\Sigma_2 = \bot$
    If $\rho = x.s$ and s is declared as **single**, try unpacking the owners of ρ, ending with x, from the first available permission on a transitive owner of ρ in $\overline{\Sigma}_2$. If there is no such permission, fail. Else, add all these unpacking statements in S_b. Then unpack y of type $[\rho]$. If there is no such y, fail.
- $\Sigma_1 = \rho^\times$
  - $\Sigma_2 = \rho^G$
    Add **weaken single** ρ at the end of S_a.
  - $\Sigma_2 = \sigma \multimap \rho$
    Add **weaken single** ρ at the end of S_a. Add **unfocus** x: σ **as** ρ at the end of S_b where x: $[\sigma]$ is in Γ. If there is no such x, fail. Ensure permission $\sigma^\times$ needed by the **unfocus** statement are available, by recursively applying the inference algorithm.
  - $\Sigma_2 = \bot$
    If $\rho = x.s$ and s is declared as **single**, try unpacking the owners of ρ, ending with x, from the first available permission on a transitive owner of ρ in $\overline{\Sigma}_2$. If there is no such permission, fail. Else, add all these unpacking statements in S_b.
- $\Sigma_1 = \rho^G$
  - $\Sigma_2 = \sigma \multimap \rho$
    Add **unfocus** x: σ **as** ρ at the end of S_b where x: $[\sigma]$ is in Γ. If there is no such x, fail. Ensure permission $\sigma^\times$ needed by the **unfocus** statement are available, by recursively applying the inference algorithm.
  - $\Sigma_2 = \bot$
    If $\rho = x.s$ and s is declared as **group**, try unpacking the owners of ρ, ending with x, from the first available permission on a transitive owner of ρ in $\overline{\Sigma}_2$. If there is no such permission, fail. Else, add all these unpacking statements in S_b.

- $\Sigma_1 = \sigma \multimap \rho$
  - $\Sigma_2 = \sigma' \multimap \rho$ (with $\sigma \neq \sigma'$)
    Add **unfocus** x: σ **as** ρ at the end of S_a where x: $[\sigma]$ is in Γ. If there is no such x, fail. Add **unfocus** y: σ' **as** ρ at the end of S_b where y: $[\sigma']$ is in Γ. If there is no such y, fail. Ensure permission $\sigma^\times$ and $\sigma'^\times$ needed by the respective **unfocus** statement are available, by recursively applying the inference algorithm.
  - $\Sigma_2 = \bot$
    Fail.

The remaining cases are symmetric. We start with S_a and S_b being empty. Each rule ensures that Σ_1 and Σ_2 are consumed and replaced by identical permissions. Each time a rule is applied, we remove these identical permissions from $\overline{\Sigma}_1$ and $\overline{\Sigma}_2$. All these rules are applied in any order until either one of them fails, or $\overline{\Sigma}_1 = \overline{\Sigma}_2 = \emptyset$.

Notice that when a region is opened in a branch and closed in the other, we made the choice of unpacking the region in the latter instead of packing it in the former. The reason is that packing generates a proof obligation. If the region is required to be closed after the **if** statement, the inference mechanism will unpack it *after* the **if** statement. The proof obligation will then be slightly more complicated, with the two branches to take into account, one of them being trivial as the region was closed so the invariant held.

6.4 Termination, Soundness and Completeness

Termination We now prove some properties of our inference algorithm. First, we prove that it terminates.

Theorem 13 (Termination) The inference algorithm terminates on all inputs.

Proof. Proving that the computation of region constraints and their application terminate is straightforward. Proving that functions *produce* and *producelist* terminate is also easy: the main argument is that argument $\overline{\Sigma}_{no}$ strictly grows each time *produce* is called recursively, and cannot grow more than the set of all permissions on well-typed regions, which is finite.

We now prove that joining branches terminates. Recursive calls to the inference algorithm are only done to produce permissions of the form $\rho^\times$, so the recursive call will not try to join other branches. We remove permissions from $\overline{\Sigma}_1$ and $\overline{\Sigma}_2$ each time a rule is applied successfully, but we also sometimes obtain new permissions when unpacking. However, these new permissions are only on *owned regions*, on which *there already was a permission in the other branch*. The intuition behind the termination argument is thus that we cannot grow a permission set more than some boundary given by the permission set of the other branch at the beginning of the algorithm. □

Soundness We then prove that the algorithm is sound.

Theorem 14 (Soundness) Given inputs Γ, $\overline{\Sigma}$ and S, if the inference algorithm does not fail, it returns statement S' which is equivalent to S. Moreover, there are $\overline{\Sigma}'$ and Γ' such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S'\ \{\overline{\Sigma}'\}, \Gamma'$$

Proof. First we prove that S' is equivalent to S. The main loop always adds s to S' before removing s from S. This s may have been modified from the input, but only by replacing some sub-expressions e with variables x whose value is bound to e just before s.

Then we prove that S' is well-typed. The unification mechanism (along with region constraints) ensures that the *types* are correct. We now prove that *permissions* are correct. In the main loop, the set of permissions consumed by s, denoted $\overline{\Sigma}_s$, is computed from typing rules. Assuming S_{prod} is computed such that:

$$\Gamma \vdash \{\overline{\Sigma}\}\ S_{prod}\ \{\overline{\Sigma}_s, \overline{\Sigma}_{rem}\}, \Gamma'$$

then $\overline{\Sigma}_s$ are available and thus S_{prod}; s is well-typed. We now prove that S_{prod} is computed in such a fashion. First we prove from typing rules that *operations*(Γ, Σ) returns a set of well-typed statements which all happen to produce Σ. Then we prove by induction that *produce* and *producelist* return well-typed sequences returning the required permission or set of permissions. For *produce* it is straightforward. For *producelist*, we note that variable $\overline{\Sigma}_2$ contains $\overline{\Sigma}_{rem}$ by induction, and Σ thanks to the test. Finally, we prove that branches are joined correctly by noting that all inferred statements are well-typed. □

Completeness? Ideally, we would like to prove a theorem stating that if a Capucine program can be annotated without changing its result such that the program becomes well-typed, then the inference algorithm finds such annotations. We could not find any counter-example, and we believe that such a theorem could be proven. However, we are more interested in using the algorithm to help us write more concise examples than we are in proving that it is actually complete. So far, we are pleased with the results.

We nevertheless give some intuitions about completeness. The inference algorithm is defined in such a way that more backtracking could be done to find valid annotations. However, our intuition is that this is not necessary thanks to properties about region trees.

We already stated that when applying multiple region constraints, we can apply them in any order because if any order fails, then it means that a pointer should be in two regions at the same time and thus all other orders will fail as well.

The order in which we apply the rules to join branches does not matter either. The intuition is that we can first unpack everything that needs to be unpacked. Indeed, assume we apply a rule A which does not imply unpacking before a rule B which implies unpacking. If we cannot apply A after B, then A implies some weakening or unfocusing which would consume some permission which would prevent B from being applied after A. So we can assume that all unpacking have been done and that each permission has a corresponding permission in the other branch. We then show for each pair of rules that do not imply any unpacking that we can switch the order of the two rules, as they deal with different regions. There is one exception, which is if an unfocus operation is generated. But once again, it would imply that the permission on the source region is needed afterwards.

It remains to be understood whether *produce* and *producelist* could use more backtracking or not. We believe that they do not, but it remains to be proved.

6.5 Simplify Allocation and Focus

We now introduce alternative versions of allocation and focus in order to ease the work of the programmer. This implies adding a new step of inference *before* the steps we already presented. Indeed, this new step is done at the binding stage of typing.

Allocation The new version of the allocation statement is the following:

$$\textsf{let } x = \textsf{new } \mathcal{C}$$

Compared to the old one, the region annotation has been removed. The idea is that this operation is automatically replaced by:

$$\textsf{let } x = \textsf{new } \mathcal{C}\ [r]$$

where r is a *fresh* region name. A region binder:

$$\textsf{let region } r\text{: } \mathcal{C}$$

is added at the beginning of the current innermost *block*. A block is either a branch of an **if** statement, or the body of the function. It is not necessary for the scope of r to be longer than the scope of x, and the scope of x is the current innermost block.

Focus The new version of the focus statement is the following:

$$\textsf{focus } x\text{: } \rho$$

Compared to the old one, the target region annotation has been removed. This operation is automatically replaced by:

$$\textsf{focus } x\text{: } \rho \textsf{ as } r$$

where r is a fresh region name, bound in the same fashion than the fresh region is bound for allocation. It is not necessary for the scope of r to be longer than the scope during which the type of x changes from ρ to r, i.e. the current innermost block.

Synergy With Adoption Inference The question which comes to mind immediately is: but what if the user did not want to put the pointer in a fresh region, but in an existing region instead? First, the old operation is still available if needed. Second, the adoption inference mechanism we introduce in Section 6.3 can sometimes automatically insert an **adopt** operation to move the pointer from the fresh region to its intended region.

However, adoption can only be done from a singleton region to a group region. If the target region ρ is empty (permission $\rho^{\emptyset}$), adopting a pointer into it results in permission ρ^{G} while $\rho^{\times}$ would actually be possible. Permission $\rho^{\times}$ is stronger and is thus preferable. It is particularly important in the example of the next paragraph. This is one motivation for adding more adoption operations to Capucine. See Section 7.2.4 for more discussion.

Example Consider the following constructor:

```
fun newLong [r: Long] (): [r]
  consumes r^∅
  produces r^×
{
  let x = new Long;
  return x
}
```

The body is automatically expanded into:

```
{
 let region s: Long;
 let x = new Long [s];
 return x
}
```

The unification of the type $[s]$ of the returned expression x, with expected return type $[r]$, triggers the insertion of an adoption:

```
{
 let region s: Long;
 let x = new Long [s];
 adopt x as r;
 return x
}
```

Assuming adoption produces $r^{\times}$ instead of r^{G} (see Section 7.2.4), this body is now well-typed.

6.6 Experiments

Examples of Chapter 3 illustrate the inference mechanism. Indeed, there are almost no region operations: they are all inferred. The only operations that are left and that we would want to infer are:

- the **adopt** statement, the **let region** and the region of the **new** in function *addStudent*;
- the **focus** operation, its associated **let region** and the **unfocus** operation in function *changeMark*;
- the regions in which **new** statements allocate their pointers, in other functions.

All those statements are actually not necessary as they can be inferred as well. We only left them to illustrate their use. To sum up, the only annotations related to regions which are needed in the examples of Chapter 3 are function contracts (regions of pointer arguments, consumed and produced permissions) and **use** statements. This shows that the inference algorithm is quite handy in practice.

For instance, consider the fully-annotated version of *addStudent*:

```
fun addStudent [Rc: Course] (c: [Rc], m: int): [c.Rstudents]
 consumes Rc^×
 produces Rc^×
 post ···
{
 use invariant c;
 let region Rstud: Student;
 let s = new [Rstud];
 s.mark ← m;
 use ···;
 pack s;
 adopt s: Rstud as c.Rstudents;
 unpack c;
```

```
  c.students ← store (c.students, c.count, s);
  c.sum ← c.sum + m;
  c.count ← c.count + 1;
  pack c;
  return s
}
```

Compare this with the lightweight version which uses the inference mechanism:

```
fun addStudent [Rc: Course] (c: [Rc], m: int): [c.Rstudents]
  consumes Rc×
  produces Rc×
  post ···
{
  use invariant c;
  let s = new Student;
  s.mark ← m;
  use ···;
  c.students ← store (c.students, c.count, s);
  c.sum ← c.sum + m;
  c.count ← c.count + 1;
  return s
}
```

We removed 5 lines of code out of less than 20.

The time taken to infer and type a Capucine program is negligible in practice. It takes less than 16ms on an Intel Core 2 Duo CPU E6850 at 3.00GHz for the Sparse Array example, and less than 14ms for the Course example. Note that most of this time is actually spent logging various debugging information (163 lines of text for the Sparse Array example and 194 lines for the Course example) using *printf* commands. Also note that we did not try to optimize the algorithm at all.

6.7 Conclusion

We have shown how to infer tedious annotations to lighten the work of the user. In particular:

- most of the time, the user do not have to annotate the region in which pointers are allocated;
- focus and unfocus operations are automatically inserted if necessary when a pointer is modified;
- focus and unfocus are also inserted when a function call expects a singleton region, if a variable of the region is available;
- packing and unpacking are automatically inserted if necessary as well;
- adoption operations are automatically inserted when the pointer to be adopted is available as a variable.

Practice shows that this makes the Capucine language much easier to program with. However, some annotations are not inferred, notably:

- region arguments, **consumes** and **produces** clauses of functions;
- adoption of complex expressions.

We have explained why those were not good candidates for inference.

While Loops The Capucine language does not feature loops. We encode them using recursive function calls. If while loops were to be added to the core language, they should be annotated using a *permission invariant*, i.e. a list of permissions which should be available at the beginning and at the end of each iteration of the loop. The inference algorithm would then use this permission invariant as the set of permissions consumed and produced back by the loop body.

Chapter 7

Conclusion

Before concluding this thesis, we compare our work with related work. Then we give some ideas for extensions that could be considered for Capucine.

7.1 Related Work and Contributions

7.1.1 Other Memory Models

In Section 2.4.1 we introduced a simple memory model where the heap is represented as a single array from locations to values. This model is, in particular, used in the Spec# [Barnett04a] platform when producing Boogie [Barnett05] intermediate programs. We argued in Section 2.4.2 that this model lacked static separation between pointers. In Section 2.4.3 we showed how to obtain some static separation thanks to the Burstall-Bornat component-as-array model [Bornat00]. This model splits the array representing the heap into several arrays, one for each structure field.

In Section 2.4.4 we showed how Thierry Hubert and Claude Marché [Hubert07, Hubert08] proposed a separation analysis for the Why platform [Filliâtre07]. This separation analysis gives each pointer a region variable, unifies these region variables using the ML unification mechanism and then splits the heap into several arrays, one for each region. This mechanism has also been implemented [Moy09] in the Jessie intermediate language [Marché07] of the Why platform.

Comparison With Capucine In Capucine, pointers may own regions and the heap is thus a tree. Depending on how we represent this tree, we achieve more or less separation. In Section 5.2 we present a simple encoding of the heap where only *root regions*, i.e. regions at the top of the tree, are separated as statically different variables.

In Section 5.5 we show that using *prefix trees* we can split the heap not only by root regions, but also by owned regions and class fields. This allows Capucine to achieve separation similar to the one used in Jessie, except the region language is richer with the possibility for pointers to move from their region into another, the possibility to implement functions which allocate, initialize and return new pointers, and last but not least, the possibility for pointers to own regions. The *caveat* is that prefix trees are harder to use for region arguments of *logic* functions. However, the separation granted by having different variables for parts of the heap is most useful when these parts are assigned new values, which does not happen in the logic.

In Section 5.4 we show that we can represent pointers of singleton regions simply using their value. In other words, those pointers are statically separated from every other pointer, thus simplifying reasoning on the program. This simplification is inspired by work by Charguéraud and Pottier on translation of imperative programs into pure programs using *capabilities* [Charguéraud08]. Capabilities are similar to permissions.

7.1.2 Data Groups

K. Rustan M. Leino proposed *datagroups* to reason about side-effects in a modular fashion [Leino98, Leino02]. One may declare datagroups inside classes. One then declare that some fields belong to some datagroup. One may also declare that some datagroups are included in some others. Static restrictions prevent *pivot fields* from being aliased. A pivot field is a field whose type is a class which contains a datagroup H and which is declared in a class which contains a datagroup G. Note that in such a case we may want to specify that G includes H.

Datagroups can be used to hide parts of the implementation of classes. The fields of a datagroup may be hidden. Effects on the fields are then modeled by effects on the datagroup.

Comparison With Capucine Datagroups are comparable to regions of Capucine. The inclusion relation between datagroups is comparable to ownership. Intuitively, if a pointer x owns some region $x.r$, then $x.r$ is included in the region of x. Owned regions are encapsulated in their owners, and the permissions on owned regions are hidden when the owner is closed. This is comparable to the data abstraction mechanism provided by datagroups.

7.1.3 Spec# Ownership Methodology for Data Invariants

In Section 2.5 we presented the ownership methodology used in the Spec# platform [Barnett04b]. The ownership methodology of Capucine is largely based on the Spec# methodology. The main difference is that in Capucine, ownership is handled by the type system. Another key difference is that objects own *regions* instead of other objects. In other words, in Capucine objects own groups of potentially aliased other objects. Regions happen to be the main feature of Capucine which allows to control aliasing using the type system. This explains why objects own regions instead of single objects. Note that if a region is always singleton (i.e. is marked as **single** in the owner class), then it plays the same role as an owned object in Spec# (i.e. a field marked as **rep**).

The use of a type system to control ownership allows Capucine to generate less proof obligations. In Spec#, the user writes pre- and post-conditions such as `x.inv == ClassName` to state that the invariants of class `ClassName` hold for object x. This equality then appears in proof obligations. In particular, we have to prove it when calling functions which assume the invariants of their arguments. In Capucine however, this equality is replaced by permission $\rho^{\times}$, ρ^{G} or $\sigma \multimap \rho$, which is handled by the type system.

To some extent, the use of a type system also allows Capucine to infer the **pack** and **unpack** statements. Indeed, the inference mechanism of Capucine automatically unpacks objects being modified. The object is then automatically packed when the invariant is required.

On the other hand, the use of a type system implies some restrictions. Ownership transfer is possible in Capucine using adoption and/or focusing, but this consumes the permission on the source region, disabling the region completely. In general, it is not possible to remove a location x from a region ρ safely without disabling the region. We should ensure that the

type of all expressions denoting the same location as x changes so that the location cannot be accessed from ρ, but knowing which locations are equal to x is an undecidable property.

Also, as the state of invariants is encoded as a field in Spec#, the user can use complex, undecidable conditions such as: "the invariant of my argument should hold if $x + y > 0$". This is not possible with permissions, although the extension we sketch in Section 7.2.2 might allow such kind of conditions.

The ownership methodology of Spec# is compatible with object inheritance. In Capucine, there is no inheritance. As the ownership methodology is very similar, we believe that if inheritance were to be added to Capucine, we could use the ideas of Spec# to extend the ownership methodology of Capucine to handle it.

7.1.4 Ownership Types

David G. Clarke, John M. Potter and James Noble propose *ownership types* [Clarke98]. As the name suggests, it is a type system aimed at controlling aliasing using ownership. It is based on an object-oriented language. Types are annotated by their *context*. A context is a set of locations. There is a root context, denoted **norep**, owned by nobody. Each object owns a context. The context owned by the current object *this* is denoted **rep**. Classes can be parameterized by context variables m.

Comparison With Capucine The type system of Capucine generalizes ownership types. Regions are obviously similar to contexts. Capucine pointer types are also annotated by their region. In Capucine there are several root regions instead of only one root context **norep**. Those root regions are the set of region variables bound by the programmer. Similarly, an object may own zero, one or several regions in Capucine. Capucine is not object-oriented, and thus there is no special object *this*; relative context **rep** has no meaning. We can encode methods using functions with an argument called *this*. Context **rep** of ownership types then corresponds, in Capucine, to the union of owned regions of *this*. Capucine features region parameters, which are similar to context parameters.

Moreover, the Capucine language provides greater flexibility. In Capucine, we may write region $x.s$ for any variable x. We are thus not limited to contexts **rep** or **norep**, as x may be any argument of the function, or any local variable. In Capucine we also have region parameters for functions, not just for classes. Those region parameters may be used in particular to specify whether two arguments of the function may be aliased or not.

Capucine adds the ability to track the state of invariants to the type system. It also allows to move locations from their current context to another, using adoption, focus and unfocus operations.

7.1.5 Universe Types

Werner Dietl and Peter Müller propose *Universe types* [Dietl05]. The *Universe* type system provides ownership control for JML [Burdy04]. Universe types can be seen as a lightweight version of ownership types [Clarke98].

A key difference between ownership types and Universe types is that while ownership types enforce the *owner-as-dominator* property, the Universe type system enforces the *owner-as-modifier* property. The owner-as-dominator property ensures that the owner of an object x controls how x is read and modified. In other words, to access x, one must go through the

owner of x. By contrast, the owner-as-modifier property only restricts modifications: one can read any object. This property is sufficient to ensure that invariants are maintained.

Universe types have been implemented in the context of JML. The type system can thus be used in conjunction with either runtime checking or static verification using verification conditions. This is one of the strength of Universe types, as the combination of typing and a modelling language provides great expressiveness. In particular, JML features object invariants. The Universe type system is orthogonal to the invariant methodology, but as it provides ownership, it can be used to ensure the chosen invariant methodology is sound.

Universe types are annotated by either **rep**, **peer** or **readonly**. Context **rep** of Universe types is similar to context **rep** of ownership types: if an object is annotated with **rep**, then the object is owned by *this*. If an object x is annotated with **peer**, then the object has the same owner that *this*. In other words, x and *this* are brothers in the ownership tree. Finally, if an object is annotated with **readonly** then the object owner, and thus its context, is unknown. Subtyping allows a **rep** or a **peer** object to be seen as a **readonly** object.

Downcasting can be used to view a **readonly** object as a **rep** or a **peer** object. A pre-condition is required for downcasts, namely that the owner of the object is *this* (when downcasting to **rep**) or is the same as the owner of *this* (when downcasting to **peer**). This implies having, along with the type system, a way to express object owners in the modelling language. In JML, this is done using a ghost field called *owner*. Type annotations **rep**, **peer** and **readonly** of method arguments, return values and allocated objects can be translated as conditions on *owner* fields.

Only the fields of **rep** and **peer** objects can be modified. Fields of **readonly** objects cannot: they can only be read. This ensures the owner-as-modifier property.

The Universe type system does not feature context parameters for classes. Instead of using a context parameters, one uses the **readonly** annotation. To modify **readonly** variables x, one has to use a downcast. In other words, one has to prove that x is **rep** or **peer** before modifying it. This trade-off makes programs more readable and the type system is lighter, at the cost of additional conditions for the runtime checker or static verifiers.

By default, all references are **peer**, except for pure methods and exceptions which are **readonly**. In other words, by default everyone is in the same context. This allows to be backward compatible and type most already-existing programs.

Comparison With Capucine All remarks that we made when comparing Capucine and ownership types still hold when comparing Capucine with the Universe type system. In particular, contexts and regions are similar, although in Capucine there are several root regions and an object may own several regions.

In Capucine there is no equivalent to the **readonly** context annotation. Precise region annotation is thus required, which can sometimes be heavy and most importantly, does not provide the same level of expressiveness. On the other hand, region parameters and polymorphism proved to be enough for the examples we were interested in, and they do not generate verification conditions whereas downcasts from **readonly** to **rep** or **peer** do. In Capucine there is no need for an *owner* ghost field: ownership properties are verified by the type system, not by a runtime checker or a static verifier. Moreover, with Universe types there is sometimes the need for an invariant stating who is the owner of **readonly** references. There is no such need in Capucine: this information is contained in the type.

Ownership transfer in Capucine can be made using adoption and focus, at the cost of disabling the source region. With Universe types, it is not clear how such transfers could be done. If a location x is **rep** and if $x.owner$ is changed, then x still has type **rep** even though its

owner might not be *this* anymore. If one could ensure that all expressions denoting location x had type **readonly**, then one could allow changing the *owner* ghost field. But ensuring such a typing property does not seem reasonable. We found a similar problem with Capucine when we tried to extract a pointer from a region without disabling the region.

It is clear that Capucine enforces the owner-as-modifier property, in the sense that one must unpack all transitive owners of an object before modifying the object. However, it is not clear whether Capucine enforces the owner-as-dominator property. Indeed, on the one hand one does not have to unpack an object x and its owners to be able to read it, but on the other hand a permission on the region of x is required. Moreover, accessibility rules (Figure 4.2) require not only a permission on x but also permissions on the owners of x. However, this permission requirement is only here to ensure the separated translation of Chapter 5 is sound. If we had chosen a more classic translation with only one heap array, i.e. a translation closer to the intuitive model of Capucine (Section 4.3) and to the Spec#-to-Boogie translation, then we would not need to require any permission on x; but we would not benefit from the separation properties of our translation.

It is interesting to note that the default **peer** type in the Universe implementation corresponds to the use of a unique group region in Capucine. This would lead to the naive translation with only one non-separated heap, with no alias control, and thus to the same result.

7.1.6 Regions, Capabilities and Alias Types

Regions *Regions* are originally introduced by Mads Tofte and Jean-Pierre Talpin [Tofte94, Tofte97]. They structure the store as a stack of regions to control allocation and deallocation. Regions are allocated and deallocated as a whole. All points of allocation and deallocation are inferred automatically. The presence of a garbage collector is not required.

Capabilities Karl Crary, David Walker and Greg Morrisett propose a calculus of *capabilities* to type memory management [Crary99], using regions similar to those of Mads Tofte and Jean-Pierre Talpin. Capabilities are similar to permissions: they cannot be duplicated and give information about regions. Their work is presented as an alternative to garbage collectors. Capabilities are used in particular to know which regions can be deallocated.

John Tang Boyland, James Noble and William S Retert propose a type-system for capabilities [Boyland01] for object-oriented programs. It is a generalization of many usual Java annotations such as *unique* or *readonly*. The language is quite general and can be used to encode many such *policies*. The language of permissions separates read and write accesses. They also propose a permission which does not allow to access a pointer but which still allows the *location* of the pointer to be compared with others. Some permissions denote the fact that nobody else has the permission. These allow to control duplication of permission, and thus to encode linear or affine constraints. The authors extend their type-system with adoption [Boyland05]. They call this operation *nesting*, to emphasize the fact that the operation is more fine-grained than adoption as it can be applied to single fields instead of a whole object.

Alias Types Frederick Smith, David Walker and Greg Morrisett propose *alias types* [Smith00, Walker00], which are inspired by the calculus of capabilities. The idea is to control aliasing using regions and permissions. It allows the distinction between singleton

and group regions. This distinction allows *strong update*: as the type associated to a region is contained in its capability, which is linear, it can be changed safely.

Manuel Fähndrich and Robert DeLine introduce the adoption, focus and unfocus operations [Fähndrich02]. It extends the calculus of capabilities and alias types. These operations are similar to the ones used by Capucine.

Imperative Program Encoding Arthur Charguéraud and François Pottier propose a type system based on capabilities and use it to encode imperative programs into pure languages [Charguéraud08]. Their original plan was to use this as a basis for program verification. Capucine is largely based on this idea. Their translation into pure languages provides a solid basis to understand the semantics of capabilities. The idea is that capabilities are encoded as values in the pure language, each capability denoting the values of locations of its regions. In other words, if the region is singleton, the encoding of the capability denotes the value of the unique location of the region, and if the region is group, the capability is encoded as a map from locations to values. Allocation produces a permission on a new region, which translates to the creation of a new value in the pure program. Deallocation consumes a permission on the region being freed, which translates to the corresponding value in the pure program no longer being used. Modification consumes a permission and produces it back, which translates to the corresponding value being modified in the pure program. The type system of Arthur Charguéraud and François Pottier also features strong update. Contrary to previous versions of the calculus of capabilities [Crary99, Fähndrich02], their type system also allows regions to contain not only locations but also any value. This allows more expressive types and data structures such as sum types and thus linked lists. They also feature polymorphism *à la* System F [Girard71, Reynolds74]. In particular, the type system features existential types.

Concurrency Another application of permissions is to control which process may access some part of the memory in a concurrent setting. Let's show a simple example:

```
*x = 2;
*y = 49 / *x;
```

If the memory cell denoted by pointer `x` is assigned to zero between the two lines of this program, a division by zero occurs. The usual way to prevent this from happening is by using *locks* or *mutexes* to make these two lines *atomic*, i.e. to ensure that nothing will be executed between these two operations. Permissions are a way to prevent `x` from being modified. The process running the above two lines requires permissions x and y, and because permissions cannot be duplicated, no other process can modify `x`. Those two lines are no longer atomic, yet nothing that can break the behavior of the program can be inserted between them.

Protocol API Specification Kevin Bierhoff uses permissions in his PhD thesis to specify protocol APIs [Bierhoff09]. For instance, a channel may be opened, accessed and then closed, in this order. Access to a closed channel usually leads to a run-time error, but a type system using permissions can be used to catch this error when compiling. Indeed, one may specify channel creation to produce a permission which is consumed when the channel is closed, and which is required to access the channel. Kevin Bierhoff applies this approach to object-oriented languages and propose to split the state space of objects using hierarchical refinements. Each refinement describes one interface for an object. Permissions state in which exact refinement the object is, and thus which methods may be called.

The work of Kevin Bierhoff has been implemented as an Eclipse plugin called *Plural* [Bierhoff11]. The language of permissions describes the state of an object: exclusive access (*unique*), exclusive with right to modify (*full*), read-only (*pure*), immutable, and shared access (*shared*). Permissions may only coexist if they do not violate each other. They can be decomposed and recomposed. For instance, *unique* is equivalent to *pure* plus *full*. The language also features *fractional permissions* [Boyland03], although the authors claim not to need this particular feature that often. For instance, *full* is equivalent to one half of *shared* plus one other half of *shared*. The authors of Plural found permissions to be quite a powerful tool for specification.

Plaid Jonathan Aldrich et al. propose to design a permission-based programming language [Aldrich11]. Their work is inspired by *Plural*: the state of objects may change and permissions track the current interface of an object. They sketch their own proposal, called *Plaid*. The aim of the language is to provide permissions as a specification tool to, in particular, specify ownership in parallel programs, check protocol APIs and ensure that some values are not leaked outside the scope of a module. That last property can be useful to ensure security properties and encapsulation. It is also worth noting that the authors do not wish to restrict themselves to a type system: they consider *dynamic permissions*, which would be checked at run-time. This would provide more expressiveness when required.

Permissions and Ownership Yang Zhao and John Tang Boyland propose to use permissions to encode ownership [Zhao08]. Permissions may be *nested* inside others. As a result, the nested permission is "protected": one has to "un-nest" the nester permission to obtain the nested one. More precisely, permissions can be nested inside object fields. This is encoded using linear implication. The authors show, in particular, how the *owner-as-dominator* policy can be enforced. They discuss how fractional permissions could be used to provide multiple ownership. They also argue that this provides encapsulation.

Conditional Permissions Some of the above related work feature conditional permissions [Boyland05, Retert09, Zhao08]. They are permissions which are only available if some boolean predicate evaluates to *true*. They can in particular be used to specify the cases where a pointer is *null*.

Inference of Permissions Both William S Retert and Kevin Bierhoff discuss, in their PhD thesis, how to implement their permission type-system [Retert09, Bierhoff09]. The problem is: given a program which is partially annotated with permissions, can we statically prove that those permissions will indeed be available at run-time? The main issue seems to decide when permissions should be decomposed and recomposed. In particular, both systems feature fractional permissions. William S Retert introduces a *lattice of permissions*. A control-flow analysis generates constraints, and a fixpoint is found in this lattice. Kevin Bierhoff views permission-based assertions as a decidable fragment of linear logic.

Comparison With Capucine While the type system of Capucine is largely based on the ideas of regions and capabilities, it features some important differences. The first change is that Capucine does not feature polymorphism *à la* System F. It is closer to ML, and may thus be considered simpler, albeit less expressive. In particular, there is no need for

existential types, although owned permissions can be seen as existential regions. Allocation, in particular, takes an empty region as an argument.

Permissions of Capucine are capable of tracking the state of invariants. Permissions $\rho^\times$ and ρ^G imply that the invariants of all objects of ρ hold, and permission $\sigma \multimap \rho$ imply that the invariants of all objects of ρ, except possibly the objects which are also in σ, hold.

Another important change is that Capucine features a region ownership tree. A closed permission encapsulates other permissions on owned regions, as if these permissions were part of the invariant of the data structure. Moreover, the region ownership tree is used to ensure the invariant methodology is sound.

Compared to the type system of Arthur Charguéraud and François Pottier, the regions of Capucine may only contain locations. They cannot contain any value. Capucine is thus closer to the original calculus of capabilities and alias types.

Capucine does not feature deallocation. We assume the presence of a garbage collector. Adding deallocation would not be hard though: the operation would simply consume the permission of the region it deallocates, just as it is done in the calculus of capabilities.

The inference algorithm of Capucine is more specialized than the ones of William S Retert and Kevin Bierhoff. In particular, Capucine does not feature fractional permissions, conditional permissions, or permissions disjunctions. On the other hand, it does feature adoption, focus and unfocus, as well as region ownership (i.e. permission nesting). Moreover, Capucine infers region operations which have an effect in the resulting Why program. As a result, some care has to be taken to ensure that the program semantic is unchanged.

7.1.7 Separation Logic

John Reynolds propose *separation logic* to reason about potentially-aliased data structures [Reynolds02]. The main idea is that predicates describe the heap using the separative conjunction $P * Q$, which states that P holds in a part of the heap which is separated from Q. For instance, $i \hookrightarrow v * j \hookrightarrow w$ states that the heap contains exactly two *distinct* locations i and j of respective values v and w.

To reason about recursive data structures such as lists using separation logic, we use packing and unpacking operations. For instance, say $P_{list}(i)$ is a predicate stating that i is a location for a linked list. This predicate can be unpacked to obtain $i \hookrightarrow (v, j) * P_{list}(j)$ if the list is not empty. Value v is the value of the node stored at location i, and location j is the location of the next node.

Separative conjunction $*$ has a dual operation $P \mathrel{-\!\!*} Q$. This predicate describes a heap in which Q would hold if a separated heap in which P held were provided. Thus P denotes a "hole" in Q.

VeriFast [Jacobs08] is a program verification tool for programs annotated with pre-conditions and post-conditions written in separation logic. The programmer may use *open* and *close* operations to respectively unpack and pack predicates such as P_{list}.

Comparison With Capucine Separation logic focuses, as its name suggests, on the logic, whereas Capucine focuses on its type system. Separation logic encodes separation information in logic connectives whereas Capucine relies on its type system. As a consequence, proof obligations need special theorem provers to be proven, automatically or not. However, separation logic is also more expressive as a result.

Note that permissions are similar to separation logic predicates. Indeed, a permission denotes some property about some region, i.e. some part of the heap. If a pair of permissions

Σ_1, Σ_2 is available in Capucine, then the region of Σ_1 is disjoint than the region of Σ_2. Thus this pair of permission is similar to a separative conjunction. Moreover, permission $\sigma \multimap \rho$ is comparable to the magic wand $-\!\!*$ of separation logic. Indeed, permission $\sigma \multimap \rho$ denotes permission ρ^G on the condition that $\sigma^\times$ is given back.

The *open* and *close* operations of VeriFast are similar to the **unpack** and **pack** statements of Capucine. By unpacking a Capucine pointer x we obtain permissions on owned regions of x. With VeriFast, by opening a predicate we obtain "owned" predicates. For instance, by opening $P_{list}(i)$ we obtain $P_{list}(j)$ where j is the next node of list i.

7.1.8 Dynamic Frames

Ioannis T. Kassios proposed *dynamic frames* [Kassios06] to reason about pointer aliasing in a modular fashion. A *frame* is a set of locations. A *specification variable* is a variable which may only be used in the specification. In particular, specification variables may be frames. The programmer can thus use frames to state aliasing properties in the program specification, which is then proved as usual using deductive verification techniques. This implies that dynamic frames appear in proof obligations, hence the "dynamic" name.

Predicate $\Xi(f)$ states that frame f is *preserved*, i.e. no location of f is modified. This predicate implicitely takes two states s, s' as arguments, where s is the pre-state and s' is the post-state. It states that for every location l in f, $s(l) = s'(l)$.

Predicate $\Delta(f)$ states that frame f is *modified*, i.e. no location other than those in f are modified. If *all* is the set of all locations, $\Delta(f)$ can be defined as $\Xi(all - f)$.

Predicate $\Lambda(f)$ states that frame f has grown using fresh locations only. It is sometimes called the *swinging pivot requirement*. It is useful to deal with allocations.

Predicate $disjoint(f, g)$ states that frames f and g are *disjoint*. In other words, no location of f is in g and vice-versa.

Predicate f **frames** e states that if frame f is preserved (i.e. $\Xi(f)$), then e is not modified, where e is a side-effect-free expression, a term or a predicate. In other words, f includes the locations appearing in e.

For instance, the following specification states that variables x and y are not aliased:

$$f \textbf{ frames } x \wedge g \textbf{ frames } y \wedge disjoint(f, g)$$

Indeed, x and y are in disjoint frames and so they must denote different locations.

Usually, frame variables f are *self-framed*, i.e. we write f **frames** f in the specification. In other words, if no location is added to or removed from the set of location which f is intended to denote, then variable f is unchanged.

A data structure is typically specified using a self-framed frame specification variable named *rep*, which describes all locations used by the data structure. For instance, a linked list will have all its nodes and its values in *rep*. Adding a new node to the list will grow *rep* using the new node location.

Comparison With Capucine Dynamic frames can be easily added to existing deductive verification systems, as frames are variables as any other. However, Capucine requires the implementation of a type system.

Capucine puts the emphasis on *static* properties: it uses a type system capable of reasoning with location sets, which are called regions. Dynamic frames also aim at specifying location sets, called frames. Thus regions are similar to frames. However, the regions of

Capucine appear in types, whereas dynamic frames appear in the logic. Capucine regions do appear in the logic as well, but with less available operations. Thus dynamic frames put the emphasis on *dynamic* properties, i.e. properties which are verified using proof obligations. Naturally, this makes the expressive power of dynamic frames stronger than the one of Capucine, as a type system cannot handle undecidable properties without resorting to proof obligations. However, many proof obligations using dynamic frames could be automatically discharged using a type system such as Capucine.

The static means Capucine uses to specify region properties allow regions to be statically separated in proof obligations using different variables. This simplifies proof obligations and helps automatic provers greatly. Not only separation does not have to be proved, it is also easier to use to prove other proof obligations. Dynamic frames, being dynamic, do not provide such static separation.

7.1.9 Implicit Dynamic Frames

Jan Smans, Bart Jacobs and Frank Piessens recently introduced *implicit dynamic frames* [Smans08, Smans09] to unify dynamic frames [Kassios06] and separation logic [Reynolds02]. The idea is that instead of being explicitly specified, frames are computed from function pre- and post-conditions, which may contain instances of the *accessibility predicate* $\mathbf{acc}(e)$. This predicate states that locations mentioned by term t are accessible. If function reads or writes some location, it must be required to be accessible in the pre-condition. Dynamic frames are computed from the accessibility predicates contained in pre-conditions. The separating conjunction $*$ is available in predicates.

The accessibility predicate is encoded as a boolean ghost field **acc** in each object, stating whether the object is accessible. The verifier then checks before each field access that the corresponding location is accessible, i.e. the value of its **acc** field is true.

Pure methods can be used in the program specification, and are used to encode abstraction. For instance, instead of directly using some f field in a pre-condition, one can apply a method *getf* which returns the value of f. Field f can then be hidden.

Allocation implicitely returns a fresh location. This location is thus separated from previously-allocated ones.

The Chalice language features implicit dynamic frames [Leino09]. Implicit dynamic frames are seen as permissions. K. Rustan M. Leino and Peter Müller show how permissions can be encoded as integers in proof obligations. By contrast, separation logic verifiers usually perform symbolic execution.

Comparison With Capucine The accessibility predicate plays a role similar to permissions: they are both required to access some location. The difference is, once again, that permissions are part of the type system, while accessibility predicates are part of the logic and, as such, generate proof obligations. Implicit dynamic frames do not require effect annotations, as they are inferred. Capucine also infers effects. By "effects" we mean: which regions are read and which region are modified.

Implicit dynamic frames tackle the issue of modularity using pure methods. Capucine provides a framework for abstraction through encapsulation of owned regions: a closed permission encapsulates all its owned permissions, thus hiding the corresponding regions.

7.1.10 Regional Logic

Anindya Banerjee propose *regional logic* [Banerjee08] to reason about global invariants in a local fashion. His approach follows the ideas of dynamic frames [Kassios06]. More precisely, he again defines regions as sets of locations. These regions are *ghosts* variables, usable in specifications and effects. They can be assigned by specification annotations in the program, but they cannot be used to change the result of the program. Regions terms include the empty region, a singleton region defined by an expression denoting the unique location of the region, and the region composed of all locations. Region operations include union, intersection and difference. Region predicates include inclusion and disjointness. An important predicate is the *frame* predicate $P \vdash e$ **frm** P', which states that, if predicate P is valid, then predicate P' only depends on locations read by effects e. Another important predicate is $e_r \star e_w$, which states that effects e_w cannot modify what is read by effects e_r. This predicate is close to the separative conjunction of separation logic.

Comparison With Capucine As with dynamic frames [Kassios06] and separation logic [Reynolds02], the main difference between regional logic and Capucine is that Capucine is based on a type system to control alias properties, while regional logic checks separation at verification time. This results in regional logic being more expressive, at the cost of a need for more annotations from the programmer and more verification conditions.

7.1.11 Considerate Reasoning

Limits of Ownership Methodologies Matthew Parkinson argues [Parkinson07] that class invariants may not be the correct foundation for verifying object-oriented programs. Indeed, invariants often relate several objects together. This means that modifying one object may break the invariants of other objects. Ownership methodologies allow control of which objects may be modified, and they also structure the heap as a tree.

However, in some design patterns such as Subject-Observer there is not a clear ownership relation between objects. In other words, the heap is not necessarily a tree. The Subject-Observer pattern involves one subject which is observed by several observers. When the subject is modified, it notifies its observers so that they can act accordingly. There is an invariant relating the subject and the observers, and thus there must be an ownership relation between them. The observers cannot own the subject, because an object can only be owned by one other object. So the subject must own its observers. This prevent other objects from owning the observers.

The Composite Pattern Even in tree data structures, ownership may limit expressivity. A typical example is the Composite Pattern [Robby08]. It consists of a tree data structure where each node maintains some invariant about its subtree, such as the total number of children. Moreover, each node contains a back-pointer to its parent in the tree. This allows to add new nodes anywhere in the tree: given a node, one can add children to the node without starting from the root of the tree. The invariant is then re-established by following the back-pointers up to the root of the tree, instead of starting from the root and then going down. However, ownership is a methodology which requires unpacking from the root owner to modify a deeper node. It is thus not flexible enough to handle the Composite Pattern.

The following code is an example of an implementation following the Composite Pattern, using Java-like syntax.

```
class Node
{
  Node parent;
  List<Node> children;
  int count = 1;

  void Add(Node n)
  {
    n.parent = this;
    children.Add(n);
    IncrCount();
  }

  void IncrCount()
  {
    count = count + 1;
    if (parent != null)
      parent.IncrCount();
  }
}
```

The implicit invariant is that field `count` contains the sum of all `count` fields of all `children` nodes, plus 1. The `Add` method adds a new child to a `Node`, and restores the invariant of all `Nodes` by calling `IncrCount`.

Considerate Reasoning As an anwser to the position paper of Matthew Parkinson [Parkinson07], Alexander J. Summers, Sophia Drossopoulou and Peter Müller argue that class invariants are a useful specification tool (Section 1.3 covers some of their arguments), but they agree that they need to be more flexible [Summers09]. Alexander J. Summers and Sophia Drossopoulou propose Considerate Reasoning to provide such flexibility [Summers10]. In the Composite Pattern, the method which adds a child to a node must be *considerate* and re-establish the invariant of all broken objects, i.e. all transitive parents of the added node.

Considerate Reasoning requires being able to state which invariants of which objects may be broken when a given field is modified. In the above Composite Pattern implementation, adding a child to `this.children` field will break the invariant of `this`. An over-approximation of the set of objects whose invariants may break can usually be computed automatically.

Considerate Reasoning also requires being able to state that a method does not expect some invariant to hold. In the above Composite Pattern implementation, `IncrCount` does not require the invariant of `this` to hold. The method can thus be marked using the `broken` keyword, which states exactly this: the method will restore the broken invariant.

Capucine and the Composite Pattern Because Capucine implements an ownership methodology which requires unpacking from the root owner down to the object being modified, the Composite Pattern does not benefit from the Capucine ownership approach. The `Add` method would require a permission on `this`, and such permission would not be available if `this` had a packed parent. The caller would thus require to unpack the parent of `this`, and maybe the parent of the parent as well, and so on. This does not mean that the

Composite Pattern cannot be encoded in Capucine, but this means that parents cannot own their children, which is not the intuitive way to implement trees in Capucine. To encode the Composite Pattern in Capucine without changing the code to unpack from the top of the tree to the target node, we need to put all nodes in a single group region and use a complex invariant predicate [Hubert05]. This also means that the invariant could not be encoded using Capucine built-in invariants. However, regions could encode some separation properties. In particular, the `Add` method could require `n` to be in a region separated from the region of `this`, and then adopt it.

7.1.12 Liquid Types for Invariant Inference

Patrick M. Rondon, Ming W. Kawaguchi and Ranjit Jhala propose *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a type system where invariant predicates are associated to types [Rondon08]. The system is then capable of inferring such types, including invariants. Examples of inferred invariants include intervals: type $\{x : int \mid 1 \leq x \leq 99\}$ denotes the type of integers which are between 1 and 99.

Comparison With Capucine Capucine is not capable of invariant inference: invariants must be given by the user. In the body of functions, Capucine is capable to infer *when* invariants hold. It would be interesting to try and combine the two approaches. However, the inference mechanism for liquid types is not modular: the whole program must be analized to be able to infer invariants for function arguments.

7.2 Future Work

7.2.1 Encoding Mainstream Languages Into Capucine

Capucine is an intermediate language. It is not designed to program with directly. Its purpose is similar to the purpose of Jessie [Marché07]: serve as an intermediate language for the Why platform [Filliâtre07], to provide a memory model and the tools to reason on this model. The next step is thus to build a front end which would take programs written in more mainstream languages such as C, Pascal, Java or OCaml, and translate them into Capucine. This raises several questions.

A first question is: what should the annotation language look like? The *Java Modeling Language* (JML) [Burdy04] is well-known and the *ANSI/ISO C Specification Language* (ACSL) [Baudin09] is already used in the Why platform and by Frama-C [FramaC], but both languages lack the necessary region annotations that would be needed by Capucine. A possibility would be to work on inference of these annotations so that they are completely transparent, but as we argued in Chapter 6, this would probably not be fully satisfying, in particular because it would probably not be modular. Another possibility would be to design a new annotation language which would basically be JML or ACSL plus region annotations.

Another question is: how can we encode complex data structures provided by the source language into Capucine? In particular, how can we encode objects and inheritance? We already know how to encode objects into Jessie, and we already know that the invariant methodology of Spec# handles inheritance [Barnett04b]. In fact, we already applied the Spec# methodology, including inheritance, in Jessie [Bardou07]. We thus believe that we have all the tools we need for such an extension. It is however not clear how functional

programming languages such as OCaml should be encoded because of higher-order functions. Work by Johannes Kanig show one possible way to handle effects in such kind of languages [Kanig10].

7.2.2 Conditional Permissions

An idea we would like to investigate is the concept of *conditional permissions*, i.e. permissions of the form $P \Rightarrow \Sigma$ where P is a predicate and Σ is a regular permission such as $\rho^{\emptyset}$, ρ°, $\rho^{\times}$, ρ^{G} or $\sigma \multimap \rho$. Informally, the meaning of this permission is: "if P holds, then Σ is available". This idea has already been proposed in other permission-based type systems [Boyland05, Retert09, Zhao08]. We would add a new statement **use** Σ. This statement would consume $P \Rightarrow \Sigma$, produce Σ, and would require P as a pre-condition. At any time Σ could be weakened to $P \Rightarrow \Sigma$.

Our original motivation for conditional permissions is linked lists:

class *List*
{
 group *Rnext*;
 next: *option* ([*Rnext*]);
}

The fact that *Rnext* is a group region is not very satisfying, as *Rnext* contains at most one location. We cannot mark the region as **single** instead of **group** either, because then no closed list could be built as a closed list would necessarily be infinite. Instead, we propose to explicitly store the state of the region (i.e. its permission) in the invariant as follows:

class *List*
{
 region *Rnext*;
 next: *option* ([*Rnext*]);
 invariant $(next = none \Rightarrow Rnext^{\emptyset}) \wedge (next \neq none \Rightarrow Rnext^{\times})$;
}

The invariant contains two conditional permissions. The first one states that if *next* is *none* then *Rnext* is empty, while the second one states that if *next* is not *none* then *Rnext* is singleton and closed. To use the permissions of the invariants, one would first write a **use invariant** statement, followed by **use** $Rnext^{\emptyset}$ or **use** $Rnext^{\times}$.

Interesting questions arise. Can we simplify the encoding of *Rnext* as we did in Section 5.4? How much expressiveness would this extension add to Capucine? Are there other interesting examples that would benefit from conditional permissions?

7.2.3 Combine With Other Approaches for Group Regions

Even though Capucine provides many tools to separate pointers, sometimes we have no choice but to group many of them in a single group region. Also, as we already noticed in Section 7.1.11, some data structures do not benefit from the ownership approach. When we encounter these limitations we have to put all pointers of the data structure in a single group region.

Group regions correspond to the limit of what the Capucine type system can provide as a means for separation. Beyond this limit, we need tools which are not based on types. Right

now, inside a group region we only have access to the basic tools provided by deductive verification, although the invariant methodology of Capucine does help. For instance, a typical invariant we want to state is that all locations stored in an array are different. The Course example of Section 3.4.5 provides an example of such invariant.

This leads us to question the possibility of bringing deductive tools tailored for separation analysis to Capucine. For instance, separation logic, dynamic frames and regional logic all are interesting candidates. It would be interesting to see whether such kind of logic can replace the logic of Capucine. We would still have static separation through typing, but we would have more expressive power when typing is not enough.

7.2.4 More Adoption Operations

In our presentation of Capucine we chose to stay simple and we have only one kind of adoption:

$$\textbf{adopt } x\colon \sigma \textbf{ as } \rho$$

which takes a singleton region σ and a group region ρ, and puts the pointer of σ into ρ. This is sufficient most of the time when we write region annotations by hand. In practice however, our inference mechanism motivates us to add other kinds of adoptions. Here are potential candidates.

From Singleton To Empty Moving a pointer from a singleton region σ to an empty region ρ is already possible by weakening the empty region first, but the result is that ρ becomes group. We know, however, that after this kind of adoption, ρ is actually singleton, which is strictly stronger information. Permission $\rho^\times$ could thus be produced instead of ρ^G. It would not be difficult to prove that this operation is sound. It also combines well with the extension of Section 5.4, and is almost necessary to make good use of Section 6.5.

From Group Another interesting extension would be adoption from a group region instead of a singleton region. All pointers would be adopted at the same time. The result would be a union of the two regions, which are known to be disjoint before the operation.

From or Into Focus Say we want to adopt region σ into region ρ. Assume one of the two regions is being focused by a third, distinct region σ', i.e. either we have $\sigma^\times$ (or σ^G) and $\sigma' \multimap \rho$ or we have $\sigma' \multimap \rho$ and ρ^G. Right now we have to unfocus the focused region first. But in practice we could allow the adoption and produce permission $\sigma' \multimap \rho$.

Unfocus A Group Region This suggestion is not a new adoption operation but a new unfocus operation. Right now, if we focus from ρ into σ and then weaken σ to a group region, we cannot unfocus σ back into ρ. However, this would not necessarily pose any problem as long as σ^G is consumed. Indeed, the pointer which was focused would be back in ρ, and the fact that more pointers came with it is not an issue.

7.2.5 Multiple Focus

Another extension which might prove useful is being able to focus several pointers from the same region at the same time. The idea would be to improve the language of permissions,

to be able to write permissions such as $(\sigma, \sigma') \multimap \rho$. Such a permission would state that two pointers of ρ are being focused in σ and σ' respectively, and that ρ is disabled until $\sigma^\times$ and $\sigma'^\times$ are given back.

A proof obligation should be produced when focusing from ρ into σ' if $\sigma \multimap \rho$ is available. Indeed, we need to make sure that the pointer being focused is different than the pointer which was focused in σ.

We can then generalize with any list of permissions:

$$(\sigma_1, \cdots, \sigma_n) \multimap \rho$$

When focusing from ρ into σ, we would have a proof obligation stating that the pointer being focused is different than all pointers of regions $\sigma_1, \cdots, \sigma_n$. The produced permission on ρ would be:

$$(\sigma_1, \cdots, \sigma_n, \sigma) \multimap \rho$$

Thus growing the list of regions at the left of the lollipop. Another possibility would be to have group regions at the left of the lollipop, to focus several pointers into the same region σ. It remains to be explored whether this extension would actually be useful or if it would just complicate the type system.

7.3 Conclusion

Capucine combines several ingredients from various existing work. A first ingredient is *deductive verification*: Capucine features function contracts using first-order logic to describe the behavior of programs. A second ingredient is *regions*: Capucine features a type system where each pointer belongs to a region. A third ingredient is *ML polymorphism*. A fourth ingredient is *permissions*. A fifth ingredient is *ownership*: the heap is structured as an ownership tree. A sixth ingredient is *data invariants*.

Several synergies appear between each ingredient. Invariants are expressed in the logic used by the deductive verification ingredient. They also benefit from ownership which allows invariants to depend on more pointers. Permissions give information about regions, tracking the state of invariants and of the region ownership tree. They allow operations to move pointers from regions into others. Regions benefit from ML polymorphism, which is applied on region variables. The deductive verification ingredient benefits from regions as information about aliasing can be used when producing proof obligations. It also benefits from permissions as information about the state of invariants can be extracted from typing. Finally, deductive verification benefits from ownership as ownership structures the heap so that reasoning can be done in a modular fashion.

Thanks to all these ingredients, Capucine achieves what we have stated in the introduction to be the main contribution of this thesis: a type system using regions and permissions to structure the heap in a modular fashion, control pointer aliasing and data invariants and produce proof obligations where pointers are separated.

On an imaginary scale between typing and deductive verification, Capucine stands somewhere in the middle and benefits from both worlds. It features the power of first-order deductive verification while still putting a lot of emphasis on the type system. The type system discharges some proof obligations that can be considered uninteresting, and those that remain benefit from the information gained through typing.

Yet, there is still work to be done before the contributions of this thesis can be used more effectively. The language can be enhanced with more features to control pointer aliasing. Capucine is an intermediate language, and many features of mainstream languages such as inheritance still need to be encoded. Some of these features, such as higher-order functions, will prove to be quite challenging.

Until a highly-hypothetical day when programmers leave the imperative and object-oriented paradigms behind and focus on pure fonctional programming, I hope that this work will help to understand how to write safer programs with pointers.

List of Figures

Bibliography

[Aldrich11] Jonathan Aldrich, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, Roger Wolff. *Permission-Based Programming Languages (NIER Track).* (ICSE 2011)

[AltErgo] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, Alain Mebsout. *The Alt-Ergo Automated Theorem Prover.* 2008.
`http://alt-ergo.lri.fr/`

[Banerjee08] Anindya Banerjee, David A. Naumann, Stan Rosenberg. *Regional logic for local reasoning about global invariants.* (ECOOP 2008)

[Bardou07] Romain Bardou. *Invariants de classe et systèmes d'ownership.* Master Thesis, Master Parisien de Recherche en Informatique (MPRI), 2007.

[Barnett04a] Mike Barnett, K. Rustan M. Leino, Wolfram Schulte. *The Spec# Programming System: An Overview.* (CASSIS 2004)

[Barnett04b] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, Wolfram Schulte. *Verification of object-oriented programs with invariants.* Journal of Object Technology, vol. 3, no. 6, June 2004, Special issue: ECOOP 2003 workshop on FTfJP, pp. 22–56.
`http://www.jot.fm/issues/issue_2004_06/article2`

[Barnett05] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, K. Rustan M. Leino. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs.* (FMCO 2005)

[Barrett07] Clark Barrett, Cesare Tinelli. *CVC3.* (CAV 2007)

[Baudin09] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4.* 2009.
`http://frama-c.cea.fr/acsl.html`

[Bierhoff09] Kevin Bierhoff. *API Protocol Compliance in Object-Oriented Software.* PhD thesis, April 2009.

[Bierhoff11] Kevin Bierhoff, Nels E. Beckman, Jonathan Aldrich. *Checking Concurrent Typestate with Access Permissions in Plural: A Retrospective.* Peri L. Tarr and Alexander L. Wolf, editors, Engineering of Software: The Continuing Contributions of Leon J. Osterweil, 2011, pp. 35–48.

[Bornat00] Richard Bornat. *Proving Pointer Programs in Hoare Logic.* (MPC 2000)

[Boyland01] John Tang Boyland, James Noble, William S Retert. *Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only.* (ECOOP 2001)

[Boyland03] John Tang Boyland. *Checking Interference with Fractional Permissions.* (SAS 2003)

[Boyland05] John Tang Boyland, William S Retert. *Connecting Effects and Uniqueness with Adoption.* (POPL 2005)

[Burdy04] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, Erik Poll. *An overview of JML tools and applications.* International Journal on Software Tools for Technology Transfer, 2004.

[Böhme11] Sascha Böhme, Michał Moskal. *Heaps and Data Structures: A Challenge for Automated Provers.* (CADE 2011)

[Charguéraud08] Arthur Charguéraud, François Pottier. *Functional Translation of a Calculus of Capabilities.* (ICFP 2008)

[Clarke98] David G. Clarke, John M. Potter, James Noble. *Ownership Types for Flexible Alias Protection.* (OOPSLA 1998)

[Cohen10] Ernie Cohen, Michał Moskal, Wolfram Schulte, Stephan Tobies. *Local Verification of Global Invariants in Concurrent Programs.* (CAV 2010)

[Coq] *The Coq Proof Assistant.*
`http://coq.inria.fr/`

[Cousot77] Patrick Cousot, Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* (POPL 1977)

[Crary99] Karl Crary, David Walker, Greg Morrisett. *Typed memory management in a calculus of capabilities.* (POPL 1999)

[Damas82] Luis Damas, Robin Milner. *Principal type-schemes for functional programs.* (POPL 1982)

[Detlefs05] David Detlefs, Greg Nelson, James B. Saxe. *Simplify: a theorem prover for program checking.* J. ACM, 2005, pp. 365–473.

[Dietl05] Werner Dietl, Peter Müller. *Universes: Lightweight Ownership for JML.* Journal of Object Technology, vol. 4, no. 8, 2005, pp. 5–32. http://www.jot.fm/issues/issue_2005_10/article1

[Dijkstra76] Edsger W. Dijkstra. *A discipline of programming.* Series in Automatic Computation, 1976.

[Drossopoulou08] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, Alexander J. Summers. *A Unified Framework for Verification Techniques for Object Invariants.* (ECOOP 2008)

[Filliâtre03] Jean-Christophe Filliâtre. *Verification of Non-Functional Programs using Interpretations in Type Theory.* Journal of Functional Programming, vol. 13, no. 4, July 2003, pp. 709–745.

[Filliâtre07] Jean-Christophe Filliâtre, Claude Marché. *The Why/Krakatoa/Caduceus platform for deductive program verification.* (CAV 2007)

[Floyd67] Robert Floyd. *Assigning meanings to programs.* Proceedings of Symposia in Applied Mathematics, vol. 19, 1967, Mathematical Aspects of Computer Science, pp. 19–32.

[FramaC] *The Frama-C platform for static analysis of C programs.* http://www.frama-c.cea.fr/

[Fähndrich02] Manuel Fähndrich, Robert DeLine. *Adoption and focus: practical linear types for imperative programming.* (PLDI 2002)

[Gappa] Guillaume Melquiond. *Gappa: Génération Automatique de Preuves de Propriétés Arithmétiques.* http://gappa.gforge.inria.fr/

[Girard71] Jean-Yves Girard. *Une Extension de l'Interpretation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types.* (Second Scandinavian Logic Symposium 1971)

[Hoare69] C. A. R. Hoare. *An axiomatic basis for computer programming.* Communications of the ACM, vol. 12, no. 10, October 1969, pp. 576–580 and 583.

[Hubert05] Thierry Hubert, Claude Marché. *A case study of C source code verification: the Schorr-Waite algorithm.* (SEFM 2005)

[Hubert07] Thierry Hubert, Claude Marché. *Separation Analysis for Deductive Verification.* (HAV 2007) http://www.lri.fr/~marche/hubert07hav.pdf

[Hubert08] Thierry Hubert. *Analyse Statique et preuve de Programmes Industriels Critiques.* Thèse de doctorat, Université Paris-Sud, June 2008. http://www.lri.fr/~marche/hubert08these.pdf

[Hughes89] John Hughes. *Why Functional Programming Matters.* Computer Journal, vol. 32, no. 2, 1989, pp. 98–107.

[Jacobs08] Bart Jacobs, Frank Piessens. *The VeriFast program verifier.* Technical Report CW-520, Department of Computer Sciences, Katholieke Universiteit Leuven, Belgium, August 2008.

[Kanig10] Johannes Kanig. *Spécification et preuve de programmes d'ordre supérieur.* Thèse de doctorat, Université Paris-Sud, June 2010.

[Kassios06] Ioannis T. Kassios. *Dynamic frames: Support for framing, dependencies and sharing without restrictions.* (FM 2006)

[Leino98] K. Rustan M. Leino. *Data Groups: Specifying the Modification of Extended State.* (OOPSLA 1998)

[Leino02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, Yunhong Zhou. *Using data groups to specify and check side effects.* (PLDI 2002)

[Leino04] K. Rustan M. Leino, Peter Müller. *Object Invariants in Dynamic Contexts.* (ECOOP 2004)

[Leino05] K. Rustan M. Leino. *Efficient weakest preconditions.* Information Processing Letters, vol. 93, no. 6, March 2005, pp. 281–288.

[Leino09] K. Rustan M. Leino, Peter Müller. *A Basis for Verifying Multi-Threaded Programs.* (ESOP 2009)

[Leino10] K. Rustan M. Leino, Michał Moskal. *Verification of ample correctness of invariants of data-structures, edition 0.* Proceedings of Tools and Experiments Workshop at VSTTE, 2010.
`http://vacid.codeplex.com`

[Lu07] Yi Lu, John M. Potter, Jingling Xue. *Validity Invariants And Effects.* (ECOOP 2007)

[Marché07] Claude Marché. *Jessie: an Intermediate Language for Java and C Verification.* (PLPV 2007)

[Moy09] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs.* PhD Thesis, Université Paris-Sud, January 2009.
`http://www.lri.fr/~marche/moy09phd.pdf`

[Müller02] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs.* Lecture Notes in Computer Science, vol. 2262, 2002.

[Müller06] Peter Müller, Arnd Poetzsch-Heffter, Gary T. Leavens. *Modular Invariants for Layered Object Structures.* Science of Computer Programming, vol. 62, 2006, pp. 253–286.

[OCaml] *The Objective Caml Language.*
`http://caml.inria.fr/`

[Parkinson07] Matthew Parkinson. *Class Invariants: The end of the road?.* (IWACO 2007)

[Pottier05] François Pottier, Didier Rémy. *The Essence of ML Type Inference.* Advanced Topics in Types and Programming Languages, MIT Press, 2005.

[Retert09] William S Retert. *Implementing Permission Analysis.* PhD thesis, May 2009.

[Reynolds74] John Reynolds. *Towards a Theory of Type Structure.* (Colloque sur la Programmation 1974)

[Reynolds02] John Reynolds. *Separation logic: a logic for shared mutable data structures.* (LICS 2002)

[Robby08] Robby (editor). *Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems.* Technical Report CS-TR-08-07, University of Central Florida. (SAVCBS 2008)

[Rondon08] Patrick M. Rondon, Ming W. Kawaguchi, Ranjit Jhala. *Liquid Types.* (PLDI 2008)

[Smans08] Jan Smans, Bart Jacobs, Frank Piessens. *Implicit Dynamic Frames.* (FTfJP 2008)

[Smans09] Jan Smans, Bart Jacobs, Frank Piessens. *Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic.* (ECOOP 2009)

[Smith00] Frederick Smith, David Walker, Greg Morrisett. *Alias types.* (ESOP 2000)

[Summers09] Alexander J. Summers, Sophia Drossopoulou, Peter Müller. *The Need for Flexible Object Invariants.* (IWACO 2009)

[Summers10] Alexander J. Summers, Sophia Drossopoulou. *Considerate Reasoning and the Composite Design Pattern.* Lecture Notes in Computer Science, vol. 5944, 2010, pp. 328–344. (VMCAI 2010)

[Tofte94] Mads Tofte, Jean-Pierre Talpin. *Implementation of the typed call-by-value λ-calculus using a stack of regions.* (POPL 1994)

[Tofte97] Mads Tofte, Jean-Pierre Talpin. *Region-based memory management.* Information and Computation, vol. 132, no. 2, 1997, pp. 107–176.

[Walker00] David Walker, Greg Morrisett. *Alias types for recursive data structures.* (TIC 2000)

[Z3] Leonardo de Moura, Nikolaj Bjørner. *Z3, An Efficient SMT Solver.* `http://research.microsoft.com/projects/z3/`

[Zhao08] Yang Zhao, John Tang Boyland. *A Fundamental Permission Interpretation for Ownership Types.* (2nd IEEE International Symposium on Theoretical Aspects of Software Engineering 2008)