Regions and Permissions for Verifying Data Invariants

Romain Bardou, Claude Marché

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Regions and Permissions for Verifying Data Invariants

Romain Bardou — Claude Marché

N° 7412

September 2010

ISSN 0249-6399 ISRN INRIA/RR--7412--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Regions and Permissions for Verifying Data Invariants

Romain Bardou[*] [†] , Claude Marché[†*]

Thème : Programmation, vérification et preuves
Équipes-Projets PROVAL

Rapport de recherche n° 7412 — September 2010 — 37 pages

Abstract: To formally verify behavioral properties of programs, stating complex first-order formulas as data invariants proves useful. In the context of pointer programs, such invariants are hard to maintain because of aliasing. We propose a type system based on memory regions and linear permissions which allows to reduce preservation of invariants to first-order verification conditions in a sound way. It further allows data abstraction and effect hiding. It thus provides an approach to modular verification of behavioral properties of pointer programs.

Key-words: Formal Specification, Deductive verification, Data invariants, Abstraction, Region-based type system

This work is partly supported by INRIA Collaborative Research Action (ARC) "CeProMi" (`http://www.lri.fr/ceprom i/`) and by the ESF COST action IC0701 "Formal Verification of Object-Oriented Software" (`http://www.cost-ic0701.org/`)

[*] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
[†] INRIA Saclay - Île-de-France, F-91893

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Vérification d'invariants de données à base de régions et de permissions

Résumé : Les invariants de données sont nécessaires pour établir des propriétés fonctionnelles avancées des programmes. Leur vérification par preuve demande de les exprimer dans un langage logique expressif comme les formules du premier ordre. Dans le cas des programmes avec pointeurs, la vérification de ces invariants est rendue encore plus complexe à cause du partage.

Nous proposons un système de typage statique basé sur des régions mémoire et des permissions d'accès linéaires, afin de réduire, de façon sure, la vérifaction de preservation des invariants à des obligations de preuve. Notre approche permet l'abstraction de données et le masquage des effets de bords internes aux modules de programmes. Ainsi, cette approche est une méthode de vérification modulaire de propriétés de programmes avec pointeurs et partage.

Mots-clés : Spécification formelle, Vérification par preuve, Invariants de données, Abstraction, Typage à base de régions

Contents

List of Figures

1 Introduction

Complex properties on functional behavior of programs can be expressed using the so-called *behavioral interface specification languages*. Examples of such specification languages are JML [8] for Java, Spec# [4] for C#, ACSL [5] and VCC [25] for C. These typically allow to express properties using first-order formulas over program states. Following the Hoare-logic concepts, those formulas are typically classified as preconditions and postconditions on program routines, invariants on loops, assertions on particular program points, and data invariants. The latter are also called class invariants in the context of object-oriented (OO) languages.

Verifying that a given program meets its given specification amounts to generate *verification conditions* (VC): first-order logic formulas which must be checked for validity. Although the techniques for generating proper VCs have been studied for a long time, generating sufficient VCs for preservation of data invariants remains a challenging issue [17].

Here is a simple example using invariants, in OO-style syntax. The first class below introduce a Sensor to record the rotation speed of a car's wheels.

```
class Sensor {
 double rpm; // wheels rpm
 // update the rpm
 void read() { ··· }
}
```

The class Car below is in charge of displaying the current speed on the board.

```
static void main (String argv) {
 ...
 Sensor s := new Sensor();
 Car c := new Car(s);
 s.read(); ···
}
```

A data invariant specifies that the displayed speed should be a given factor K of the wheel's rpm. When calling Car.update(), the invariant is violated by the call to mySensor.read() but re-established before the end of the method call. Nevertheless, it is not enough to check that all methods of Car preserve the invariant to guarantee it is preserved all the time. Indeed, in the main program below:

```
class Car {
 int displayedSpeed;
 Sensor mySensor;
 data invariant:
  displayedSpeed = round(K × mySensor.rpm);
 // update the displayed speed
 void update() {
  mySensor.read();
  displayedSpeed := K × mySensor.rpm;
 }
}
```

the invariant is violated, because the main program calls a sensor's method directly without notifying the change to the car. Such a program is correct enough to compile

and run, but is conceptually flawed: once the car is initialized with its sensor, no other part of the program should access the sensor directly.

There are several techniques proposed in the literature to avoid this kind of mistake. They share the common informal idea that the car should *own* its sensor, in the sense that nobody else should access the sensor's method directly. In the context of object-oriented paradigm, techniques are mostly based either on runtime checking of ownership properties, or on deductive techniques via VC generation. Only few approaches are based on advanced static typing techniques, e.g. *universe type systems* [12]. In the context of functional-style programs with side-effects, the same issue is tackled mostly by advanced type systems (e.g. involving *memory regions* and various notions of *access permissions* or *capabilities* [9]) but these do not consider behavioral properties expressed by general formulas. Our goal is to bridge the gap between those approaches by proposing a technique based first on advanced static checking involving regions and permissions, and generation of VCs only when verification goes beyond static typing.

Our contributions, which are developed on a core programming language that we introduce in Section 3, are the following:

- We define a type system in two parts: the first deals with memory regions and the second deals with permissions. A noticeable originality is that for read access no permission is required. The general ideas are presented in Section 2 and formalized in Section 3.
- Our first result (Theorem 3.1) shows that for well-typed programs, data invariants for pointers in a given region are guaranteed to be valid whenever the *closed permission* on that region is available. This is shown in Section 3.4.
- Our second result is motivated by the need for modular reasoning. Theorem 4.1 is a soundness property of effect hiding: we may hide some regions in the public interface of a module and side-effects on such regions can be safely ignored.

The core language has been implemented as a prototype called *Capucine*, available on web page `http://romain.bardou.fr/capucine/`. We describe briefly this implementation in Section 5 and illustrate it on a complete example. Other examples are given on the web page.

We compare with related work in Section 6.

2 Permission-based Invariant Preservation

This section describes informally the core ideas of our language. For simplicity, we adopt a classical, not OO-style, setting. Formalization is done in Section 3.

Types We require invariants to be associated to pointer types. This allows to *statically* know the invariant of a pointer by looking at its type. For instance, type $PosInt$ is the type of pointers on positive integers:

type $PosInt =$
 int
 inv$(this) = !this > 0$
end

where int is the type of values pointed by $PosInt$ pointers. Parameter $this$ of the invariant is a $PosInt$ pointer, which value is accessed using $!this$.

Regions As we cannot statically consider all pointers, we put each pointer in a *region*, which is a set of pointers. The type of $PosInt$ pointers of region ρ is denoted by $PosInt[\rho]$. Here is a function that adds two $PosInt$ pointers:

val $sum(x\text{: } PosInt[\rho_x],\ y\text{: } PosInt[\rho_y])$: int $= !x + !y$

Permissions We associate *permissions* to regions. Permissions are properties of regions and their pointers. We consider five kinds of permissions:

- $\rho^{\emptyset}$, which denotes that region ρ is empty;
- ρ°, which denotes that region ρ is a singleton;
- $\rho^{\times}$, which denotes that region ρ is a singleton and that its only pointer verifies its invariant;
- ρ^{G}, which denotes that every pointer of region ρ verifies its invariant ("^{G}" stands for "group");
- $\sigma \multimap \rho$, which denotes that $\sigma^{\times}$ can be given to obtain ρ^{G}.

Pointers belonging to a region ρ with permission ρ° are *open*, and pointers belonging to a region ρ with permission $\rho^{\times}$ or ρ^{G} are *closed*.

Permissions are *linear*. They can be consumed and produced by a function:

val $invert(x\text{: } PosInt[\rho_x])$: int **consumes** ${\rho_x}^{\times}$ **produces** ${\rho_x}^{\times} = 1\ /\ !x$

Function $invert$ can only be called if the caller provides ${\rho_x}^{\times}$. This ensures that x verifies its invariant and that no invertision by zero occurs. Permissions cannot be duplicated: if $invert$ did not produce ${\rho_x}^{\times}$, ownership of ρ_x would be transferred from the caller to $invert$.

Consider the following functions which each take a $PosInt$ pointer and add it to a container data structure:

val $add1(x\text{: } PosInt[\rho_x],\ c\text{: } Container[\rho_c])$: unit **consumes** $\emptyset$ **produces** $\emptyset$
val $add2(x\text{: } PosInt[\rho_x],\ c\text{: } Container[\rho_c])$: unit **consumes** ${\rho_x}^{\times}$ **produces** ${\rho_x}^{\times}$
val $add3(x\text{: } PosInt[\rho_x],\ c\text{: } Container[\rho_c])$: unit **consumes** ${\rho_x}^{\times}$ **produces** $\emptyset$

The first version does not require any permission on the region of x. This means that the caller does not have to own x to add it to c. The second version requires that the caller owns x and returns the permission so that the caller does not lose ownership. With the third version, the caller loses ownership of x.

Permission ρ° is required when assigning pointers. This prevents invariants from being broken. Consider functions $decrBad$ and $decrGood1$:

val $decrBad(x\text{: } PosInt[\rho_x])$: unit $= x := !x - 1$
val $decrGood1(x\text{: } PosInt[\rho_x])$: unit **consumes** ${\rho_x}^{\circ}$ **produces** ${\rho_x}^{\circ} = x := !x - 1$

Function $decrBad$ is rejected to prevent the invariant of x from being broken if $!x = 1$. One way to fix the function is to require ${\rho_x}^{\circ}$, which $decrGood1$ does. It also returns the permission so that the caller do not lose ownership.

region ρ_{ini} **in**	$\{\rho^G, {\rho_{ini}}^{\emptyset}\}$
let $x =$ **new** $PosInt[\rho_{ini}]$ **in**	$\{\rho^G, {\rho_{ini}}^{\circ}\}$
$x := 1;$	$\{\rho^G, {\rho_{ini}}^{\circ}\}$
pack $x;$	$\{\rho^G, {\rho_{ini}}^{\times}\}$
let $x' =$ (**adopt** x **in** ρ) **in**	$\{\rho^G\}$
...	
region σ **in**	$\{\rho^G, \sigma^{\emptyset}\}$
let $x'' =$ (**focus** x' **in** σ) **in**	$\{\sigma \multimap \rho, \sigma^{\times}\}$
unpack $x'';$	$\{\sigma \multimap \rho, \sigma^{\circ}\}$
$x'' := 5;$	$\{\sigma \multimap \rho, \sigma^{\circ}\}$
pack $x'';$	$\{\sigma \multimap \rho, \sigma^{\times}\}$
unfocus x'' **in** ρ	$\{\rho^G\}$

Figure 1: Life of a pointer

Packing and Unpacking Operations **pack** and **unpack** respectively close and open a pointer. Assume a pointer x in a region ρ. Unpacking x consumes ${\rho_x}^{\times}$ and produces ${\rho_x}^{\circ}$. Packing x does the opposite: it consumes ${\rho_x}^{\circ}$ and produces ${\rho_x}^{\times}$, but it also generates a proof obligation requiring that the invariant of x holds. This gives us another way to fix $decrBad$:

val $decrGood2(x\colon PosInt[\rho_x])$: unit **consumes** ${\rho_x}^{\times}$ **produces** ${\rho_x}^{\times}$ **pre** $!x > 1 =$
unpack x; $x := !x - 1$; **pack** x

The **pack** operation generates a proof obligation, requiring that the old value of x is greater than 1. We add a precondition to ensure this (permission ${\rho_x}^{\times}$ only gives $!x > 0$).

Choosing between $decrGood1$ and $decrGood2$ is an important design decision. Does the function need the invariant of its parameter? Does the function preserve this invariant? Should the function require the caller to handle the opening and closing of the parameter? Or, on the opposite, should the invariant be hidden and should the caller only manipulate closed values? Our type system handles all these cases and gives the choice to the user.

Focusing and Unfocusing To modify a pointer x of a group region ρ (i.e. with permission ρ^G), we need to extract it first. Indeed, we need to keep track of open pointers so that when we close them, only their invariant is verified. We extract pointers to singleton regions using the **focus** operation: if x is in ρ, **focus** x **in** σ consumes ρ^G and $\sigma^{\emptyset}$ and produces $\sigma \multimap \rho$ and $\sigma^{\times}$. This returns another pointer, which is the same as x but typed with region σ. This is reversed using **unfocus**.

Life of a Pointer A pointer is created using allocation: **new** $\mathcal{C}[\rho]$ returns a fresh pointer of type $\mathcal{C}[\rho]$. Region ρ must be empty before (permission $\rho^{\emptyset}$) and is singleton (ρ°) after.

The example of Figure 1 sums up the life of a pointer. We write available permissions after each line for convenience. A pointer x is first created in an empty region ρ_{ini}. It is initialized, packed, and put in an existing group region ρ using *region adoption*. Operation **region** σ **in** e binds a new empty region σ in e. Later, its value is modified. But before that the pointer is focused and unpacked.

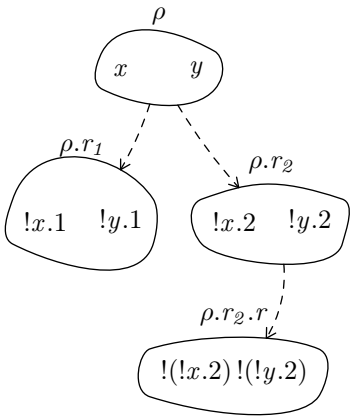

Figure 2: Region ownership tree

Many of these operations could be inferred. For instance, if the user modifies a pointer x in a region ρ with permission ρ^G, it is mandatory to focus and unpack x before. In this article though, we focus on the core ideas of our language and leave inference for future works.

Region Ownership Tree Consider a type $\langle \rho_1, \rho_2 \rangle\ Pair$, which is parameterized by regions ρ_1 and ρ_2. It uses two positive integers in respective regions ρ_1 and ρ_2, with an invariant stating that one integer is greater than the other. This invariant is easily broken, as after packing the pair pointers of ρ_1 and ρ_2 can still be assigned. To prevent this, the pair must *own* its components. We use the same syntactic condition which is used in ownership approaches [3]: the invariant of a pointer x cannot depend on a pointer which is not (transitively) owned by x. However, in our system it is the regions that own other regions. We add an **own** declaration in our type, which acts as a region binder:

type $Pair =$
 own r_1, r_2
 $(PosInt[r_1] \times PosInt[r_2])$
 inv$(this) = !(!this.2) > !(!this.1)$
end

Regions r_1 and r_2 are no longer region parameters, they are owned by the data structure. If a region ρ contains $Pair[\rho]$ pointers, we can access the owned regions of ρ using $\rho.r_1$ and $\rho.r_2$.

To prevent pointers of $\rho.r_1$ and $\rho.r_2$ from being modified when ρ is packed, the "package" ρ contains permissions ${\rho.r_1}^G$ and ${\rho.r_2}^G$. These permissions are not available as long as ρ is packed. Unpacking ρ produces ${\rho.r_1}^G$ and ${\rho.r_2}^G$, and packing ρ again consumes those permissions.

All pointers of ρ share the same regions $\rho.r_1$ and $\rho.r_2$ for all of their owned pointers. This is illustrated by Figure 2, with two $Pair[\rho]$ pointers.

Using Invariants in Proofs We know that whenever $\rho^\times$ or ρ^G is available, pointers of ρ verify their invariant. To actually use these invariants in proofs, we can provide an operation of *invariant assertion* such as: **assert invariant of** e. This operation

Type definition:
type $\mathcal{C} =$
 own $r,\cdots,r$
 τ
 inv$(x) = P$
end
With :
$\mathcal{C} ::= \langle r,\cdots,r\rangle\ (\tau,\cdots,\tau)\ C$

Values :

v	::=	c	Constant
		p	Address
		$(v,\cdots,v)$	Tuple

Types :

τ	::=	unit, int, bool, $\cdots$	Base type
		$\tau\times\cdots\times\tau$	Tuple
		$\mathcal{C}[\rho]$	Pointer
		α	Type variable

Regions :

ρ	::=	r	Region name or variable
		$\rho.r$	Local region

Permissions :

Σ	::=	$\rho^{\emptyset}$	Empty region
		ρ°	Open singleton region
		$\rho^{\times}$	Closed singleton region
		ρ^{G}	Group region
		$\sigma \multimap \rho$	Focus lock

Figure 3: Syntax of type declarations

would provide the invariant of e at the current point of the program, in the hypotheses of the proof obligations, if the right permission is available. Other approaches add these hypotheses automatically [2].

3 Formalization

3.1 Typing with Regions

This section details the first part of our type system: typing of regions in expressions. Typing of permissions will be tackled in Section 3.2. Our type system with regions is very similar to [26, 29, 15], themselves based on ML polymorphism.

Syntax of the language is as follows: first the syntax of type declarations is given in Figure 3, then syntax of logic formulas is presented in Figure 4 and finally syntax of programs is in Figure 5.

Region binders Regions are bound at the level of functions. All regions are implicitly quantified universally. Consider these two different sum functions:

Terms :

t	::=	$f(t,\cdots,t)$	Logic application (including equality)
		$t.i$	Projection
		x	Variable
		$!t$	Dereferencing

Predicates :

P	::=	$\top \mid \bot \mid \exists x,\, P \mid \forall x,\, P \mid P \vee P \mid P \wedge P \mid \neg P$	Connector
		t	Term

Figure 4: Syntax of logic annotations

Expressions :

e	::=	v	Value
		$(e,\cdots,e)$	Tuple
		$e.i$	Projection
		x	Variable
		let $x = e$ **in** e	Local variable
		$e;\, e$	Sequence
		$f(e,\cdots,e)$	Function call
		if e **then** e **else** e	Test
		while e **do** e	Loop
		$e := e$	Assignment
		$!e$	Dereferencing
		new $\mathcal{C}[\rho]$	Allocation
		pack e	Packing
		unpack e	Unpacking
		adopt e **in** ρ	Adoption
		focus e **in** σ	Focus
		unfocus e **in** ρ	Unfocus
		region r **in** e	Local region

Value or function:

val $f(x\colon \tau, \cdots, x\colon \tau)\colon \tau$
consumes $\{\Sigma,\cdots,\Sigma\}$, **pre** P
produces $\{\Sigma,\cdots,\Sigma\}$, **post** $P = e$

Figure 5: Syntax of programs

$$\frac{\Gamma \vdash e_1 : \mathcal{C}[\rho] \qquad \mathcal{C} : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 := e_2 : \text{unit}} \textsc{Assign}$$

$$\frac{\Gamma \vdash e : \mathcal{C}[\rho] \qquad \mathcal{C} : \tau_1}{\Gamma \vdash !e : \tau_1} \textsc{Deref}$$

$$\frac{f(\tau_1, \cdots, \tau_n) : \tau \qquad \Gamma \vdash e_1 : \tau_1\,\sigma \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n\,\sigma}{\Gamma \vdash f(e_1, \cdots, e_n) : \tau\,\sigma} \textsc{Call}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_2 : \tau} \textsc{If} \qquad \frac{}{\Gamma \vdash \textbf{new } \mathcal{C}[\rho] : \mathcal{C}[\rho]} \textsc{Alloc}$$

$$\frac{\Gamma \vdash e : \mathcal{C}[\rho]}{\Gamma \vdash \textbf{pack } e : \text{unit}} \textsc{Pack} \qquad \frac{\Gamma \vdash e : \mathcal{C}[\rho]}{\Gamma \vdash \textbf{unpack } e : \text{unit}} \textsc{Unpack}$$

$$\frac{\Gamma \vdash e : \mathcal{C}[\sigma]}{\Gamma \vdash \textbf{adopt } e \textbf{ in } \rho : \mathcal{C}[\rho]} \qquad \frac{\Gamma \vdash e : \mathcal{C}[\rho]}{\Gamma \vdash \textbf{focus } e \textbf{ in } \sigma : \mathcal{C}[\sigma]}$$

$$\frac{\Gamma \vdash e : \mathcal{C}[\sigma]}{\Gamma \vdash \textbf{unfocus } e \textbf{ in } \rho : \text{unit}}$$

Figure 6: Typing expressions

val $sum1(x\text{: } PosInt[\rho], y\text{: } PosInt[\rho])$: int $= !x + !y$
val $sum2(x\text{: } PosInt[\rho_x], y\text{: } PosInt[\rho_y])$: int $= !x + !y$

Parameters of $sum1$ must belong to the same region ρ, whereas no restriction is made by $sum2$ as ρ_x and ρ_y are two different regions.

Regions may also be bound at the level of types, either as owned regions or as type parameters. A local binder is available at the level of expressions: **region** ρ **in** e creates an empty region ρ which is bound in e.

As for ML type variables, region variables are unified when needed. If a value has type $PosInt[\rho]$ but is used with type $PosInt[\sigma]$, regions variables ρ and σ are unified. If they cannot be unified, a typing error is raised.

Typing rules are given in Figure 6.

Typing rules The focus operation takes a pointer of a group region and copies it temporarily into another empty region (which becomes a singleton region). The property of the regions being empty, singleton or group will be handled by permissions. Thus, the only thing the FOCUS rule does is to check that the pointer is indeed a pointer, and to ensure that the returned pointer is of the same type but in the new region.

Typing function calls is done by finding a substitution σ. This substitution both substitutes type variables (in case of polymorphism) and region variables. This substitution is applied to parameter types and to the return type. Rule CALL uses $f(\tau_1, \cdots, \tau_n)$: τ to denote the fact that f is declared with parameter types $\tau_1, \cdots, \tau_n$ and return type τ.

The other rules are quite straightforward. Allocating a pointer produces a pointer of the requested region, packing or unpacking an expression requires that the expression is a pointer, and so on.

3.2 Permissions

This section details the second part of our type system: typing of permissions. Rules are given in Figure 7.

Structural rules A *permission sequent* takes the following form:

$$\{\bar{\Sigma}_1\}\ e\ \{\bar{\Sigma}_2\}$$

where e is an expression, and $\bar{\Sigma}_1$ and $\bar{\Sigma}_2$ are lists of permissions. This reads: "e consumes $\bar{\Sigma}_1$ and produces $\bar{\Sigma}_2$". Informally, permissions $\bar{\Sigma}_1$ are required for e to reduce, and once e reduces to a value, we are left with permissions $\bar{\Sigma}_2$. This mechanism is best illustrated by rule CSEQ: if e_2 consumes $\bar{\Sigma}_2$, then e_1 must produce $\bar{\Sigma}_2$ for the sequence $e_1; e_2$ to be typed.

These sequents allow us to define a *linear* typing of permissions. Permissions cannot be duplicated: $\rho^\emptyset$ is *not* equivalent to $\rho^\emptyset, \rho^\emptyset$. Thus, $\bar{\Sigma}$ denotes a *bag* of permissions, or a multi-set, but not a set. The order does not matter, however.

Rule CWEAK1 allows to drop permissions. If an expression produces Σ, we might as well use it in a context where it did not.

Permissions $\rho^\emptyset$ and $\rho^\times$ can be weakened to ρ^G using rules CWEAK2 and CWEAK3 respectively. Indeed, ρ^G denotes that all pointers of region ρ verify their invariants. In particular, it is the case if ρ is empty (denoted by $\rho^\emptyset$) or if ρ is a singleton whose only pointer verify its invariant (denoted by $\rho^\times$).

An expression that reduces by consuming some permissions can always reduce by consuming and re-producing more permissions. This is done using rule CWEAK4, which is similar to a framing rule.

General expressions Values (rule CVALUE) are already fully reduced, and thus do not consume nor produce any permission.

Reducing an if-then-else test (rule CIF) or a while loop (rule CWHILE) does not consume nor produce any permission in itself. Note, however, that these rules assume some evaluation order between the sub-expressions. Other rules with multiple sub-expressions such as CASSIGN also assume such order.

Calling a function (rule CCALL) consumes the permissions consumed by the first parameter, then produces the permissions produced by this first parameter, and so on until all parameters are reduced and we are left with permissions $\bar{\Sigma}$. Then we remove from $\bar{\Sigma}$ the permissions consumed by the function, and add the permissions produced by the function, to obtain the final permissions $\bar{\Sigma}'$.

Pointer expressions Dereferencing (rule CDEREF) does not require any permission. In other type systems with permissions, it usually requires some access right on the pointer. In our system, as dereferencement does not necessarily assume the invariant of the pointer being read, no permission is required.

Assignment (rule CASSIGN) must not break any invariant. Invariants that might be broken are the invariant of the pointer being modified and those of its transitive owners. Thus we require the pointer to be open: it must be in a region ρ with permission ρ°.

$$\frac{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \Sigma\}}{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2\}}\ \textsc{CWeak1} \qquad \frac{\{\bar\Sigma_1\}\ e_1\ \{\bar\Sigma_2\} \qquad \{\bar\Sigma_2\}\ e_2\ \{\bar\Sigma_2\}}{\{\bar\Sigma_1\}\ \textbf{while}\ e_1\ \textbf{do}\ e_2\ \{\bar\Sigma_2\}}\ \textsc{CWhile}$$

$$\frac{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \rho^{\emptyset}\}}{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \rho^{G}\}}\ \textsc{CWeak2} \qquad \frac{\{\bar\Sigma_1,\ r^{\emptyset}\}\ e\ \{\bar\Sigma_2\}}{\{\bar\Sigma_1\}\ \textbf{region}\ r\ \textbf{in}\ e\ \{\bar\Sigma_2 - r\}}\ \textsc{CRegion}$$

$$\frac{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \rho^{\times}\}}{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \rho^{G}\}}\ \textsc{CWeak3} \qquad \frac{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2\}}{\{\bar\Sigma_1\}\ !e\ \{\bar\Sigma_2\}}\ \textsc{CDeref} \qquad \frac{}{\{\bar\Sigma\}\ v\ \{\bar\Sigma\}}\ \textsc{CValue}$$

$$\frac{\{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2\}}{\{\bar\Sigma,\ \bar\Sigma_1\}\ e\ \{\bar\Sigma,\ \bar\Sigma_2\}}\ \textsc{CWeak4} \qquad \frac{\{\bar\Sigma_1\}\ e_1\ \{\bar\Sigma_2\} \qquad \{\bar\Sigma_2\}\ e_2\ \{\bar\Sigma_3\}}{\{\bar\Sigma_1\}\ e_1;\ e_2\ \{\bar\Sigma_3\}}\ \textsc{CSeq}$$

$$\frac{\begin{array}{c}\bar\Sigma\ =\ \bar\Sigma''\ \uplus\ \textbf{consumes}(f)\ \sigma \qquad \bar\Sigma'\ =\ \bar\Sigma''\ \uplus\ \textbf{produces}(f)\ \sigma \\ \{\bar\Sigma_1\}\ e_1\ \{\bar\Sigma_2\} \qquad \cdots \qquad \{\bar\Sigma_n\}\ e_n\ \{\bar\Sigma\}\end{array}}{\{\bar\Sigma_1\}\ f(e_1, \cdots, e_n)\ \{\bar\Sigma'\}}\ \textsc{CCall}$$

$$\frac{\{\bar\Sigma_1\}\ e_1\ \{\bar\Sigma_2\} \qquad \{\bar\Sigma_2\}\ e_2\ \{\bar\Sigma_3\} \qquad \{\bar\Sigma_2\}\ e_3\ \{\bar\Sigma_3\}}{\{\bar\Sigma_1\}\ \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \{\bar\Sigma_3\}}\ \textsc{CIf}$$

$$\frac{e_1 :\ \rho \qquad \{\bar\Sigma_1\}\ e_1\ \{\bar\Sigma_2\} \qquad \{\bar\Sigma_2\}\ e_2\ \{\bar\Sigma_3,\ \rho^{\circ}\}}{\{\bar\Sigma_1\}\ e_1 := e_2\ \{\bar\Sigma_3,\ \rho^{\circ}\}}\ \textsc{CAssign}$$

$$\frac{}{\{\bar\Sigma,\ \rho^{\emptyset}\}\ \textbf{new}\ \mathcal{C}[\rho]\ \{\bar\Sigma,\ \rho^{\circ},\ \textbf{own}(\rho)^{\emptyset}\}}$$

$$\frac{e :\ \mathcal{C}[\rho] \qquad \{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \rho^{\circ},\ \textbf{own}(\rho)^{G}\}}{\{\bar\Sigma_1\}\ \textbf{pack}\ e\ \{\bar\Sigma_2,\ \rho^{\times}\}}$$

$$\frac{e :\ \mathcal{C}[\rho] \qquad \{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \rho^{\times}\}}{\{\bar\Sigma_1\}\ \textbf{unpack}\ e\ \{\bar\Sigma_2,\ \rho^{\circ},\ \textbf{own}(\rho)^{G}\}} \qquad \frac{e :\ \mathcal{C}[\sigma] \qquad \{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \sigma^{\times},\ \rho^{G}\}}{\{\bar\Sigma_1\}\ \textbf{adopt}\ e\ \textbf{in}\ \rho\ \{\bar\Sigma_2,\ \rho^{G}\}}$$

$$\frac{e :\ \mathcal{C}[\rho] \qquad \{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \sigma^{\emptyset},\ \rho^{G}\}}{\{\bar\Sigma_1\}\ \textbf{focus}\ e\ \textbf{in}\ \sigma\ \{\bar\Sigma_2,\ \sigma^{\times},\ \sigma \multimap \rho\}}$$

$$\frac{e :\ \mathcal{C}[\sigma] \qquad \{\bar\Sigma_1\}\ e\ \{\bar\Sigma_2,\ \sigma^{\times},\ \sigma \multimap \rho\}}{\{\bar\Sigma_1\}\ \textbf{unfocus}\ e\ \textbf{in}\ \rho\ \{\bar\Sigma_2,\ \rho^{G}\}}$$

Figure 7: Typing permissions

This implies that its owners are also open. In practise, CASSIGN requires that after e_1 and e_2 are executed, permission ρ° is available to actually run the assignment $e_1 := e_2$. The permission itself is unaffected.

Allocation (rule CALLOC) returns a fresh, uninitialized pointer in a region ρ. This new pointer might not verify its invariant. Thus, we cannot allow allocation in a group

region (ρ^G) as group regions assume the invariants of all of their pointers. We cannot allocate in a singleton region, as the region would contain two pointers, become a group region, and we would face the same problem. Thus, we only allow allocation in empty regions. The operation consumes $\rho^\emptyset$ and produces ρ°, as the region is now singleton. Allocation also produces permissions denoting the fact that all regions owned by ρ are empty ($\mathbf{own}(\rho)^\emptyset$), so the user is able to allocate or adopt pointers in the regions owned by ρ.

Packing (rule CPACK) a pointer in a region ρ requires that ρ is singleton and not already packed (ρ°). All regions owned by ρ must be closed: permissions $\mathbf{own}(\rho)^G$ are also consumed, to be "stored" in $\rho^\times$. This prevents the user from opening a pointer which is owned by a closed pointer. Packing generates a proof obligation: the invariant of the pointer being closed must hold.

Unpacking (rule CUNPACK) is the opposite of packing: it consumes $\rho^\times$ and produces ρ° and permissions $\mathbf{own}(\rho)^G$. Unpacking cannot directly be done on pointers in group regions, it requires focusing first.

Adoption (rule CADOPT) allows a group region ρ to absorb a singleton region σ. The pointer x of this singleton region is no longer usable as a pointer of σ, as permission $\sigma^\times$ is consumed, but adoption returns a copy of the pointer in region ρ, which can be used instead. This operation is not reversible.

Focusing (rule CFOCUS) allows to *temporarily* extract a pointer from a group region ρ to an empty region σ. It is not the opposite of adoption. As long as the pointer is extracted, the group region is no longer usable: permission ρ^G is replaced by $\sigma \multimap \rho$, which can be seen as a lock on region ρ. The key to this lock is $\sigma^\times$, which is also produced by the focus operation. Indeed, region σ is no longer empty: it contains the extracted pointer.

Unfocusing (rule CUNFOCUS) takes a lock $\sigma \multimap \rho$, its key $\sigma^\times$, and unlocks ρ. The key is destroyed in the process: region σ is no longer usable.

Creation of a new empty region r (rule CREGION) produces permission $r^\emptyset$. This region cannot get out of its scope: after e is computed, all permissions on r are ignored (we denote this operation $\bar{\Sigma} - r$).

3.3 Memory Model and Operational Semantics

To express the soundness of our type system, we first give a semantics for it. We use a big-step operational semantics. This section details the memory model used, how expressions reduce to values and how these reductions modify memory. The proof of soundness itself is done in Section 3.4.

Heap Model We model the heap using: a function H from pointer addresses to their values, a function R from region names to the set of their pointers, and a function F from region names to region names. We denote $\mathcal{H} = H, R, F$. Function H represents the traditional heap. In this sense, it is the only part of $\mathcal{H}$ that is actually needed to run the program. Function R is used to reason about the contents of regions. It can be modified by allocations, adoptions, and so on.

Function F is used to reason about focusing, which temporarily extracts a pointer from a group region ρ to a singleton region σ. However, the pointer of σ is still a pointer of ρ in our model: ρ includes σ. This is denoted by the fact that $F(\sigma) = \rho$. Function F is used to extend function R: if $F(\sigma) = \rho$, we define, for every region r owned by ρ, $R(\sigma.r) = R(\rho.r)$.

We define the *fusion* operation $R[\sigma \rightsquigarrow \rho]$. It is used by adoption, to insert the region tree of σ into ρ. Regions σ and ρ must contain pointers of the same types, with the same region trees. Thus, the operation simply consists in replacing each node of the tree of ρ by the union of this node and the corresponding node in σ.

We use notation $f[v \mapsto r]$ to denote function f where value returned for v is r instead of the previous binding, if any. In particular, we use notations $H[p \mapsto v]$, $R[\rho \mapsto s]$ and $F[\sigma \mapsto \rho]$. We also may use $\mathcal{H}$ instead of H, R or F, if it is not ambiguous. For instance, and $\mathcal{H}[p \mapsto v]$ is $\mathcal{H}$ where H is $H[p \mapsto v]$. Although $\mathcal{H}(\rho)$ could be both $R(\rho)$ or $F(\rho)$, we will only use it to denote $R(\rho)$.

Heap Coherence Intuitively, a heap is coherent if invariants that should hold do hold, regions that should be singleton are singleton, and so on. But first we need some auxiliary definitions. We define term evaluation $[\![t]\!]_{\mathcal{H}}$ in the usual fashion. In particular, $[\![!t]\!]_{\mathcal{H}} = \mathcal{H}([\![t]\!])$. Then we define the satisfaction of a predicate P in $\mathcal{H}$, and we denote it $\mathcal{H} \models P$, in the usual fashion by using $[\![t]\!]_{\mathcal{H}}$.

Definition 1 (Satisfaction of a Pointer Invariant) *A pointer p satisfies its invariant in heap $\mathcal{H}$, and we denote it $\mathcal{H} \models_{inv} p$, if and only if: if I is the invariant predicate of p, then $\mathcal{H} \models I(H(p))$ and if p' is in a region owned by the region of p then $\mathcal{H} \models_{inv} p'$ inductively.*

Definition 2 (Pointer Commitment) *A pointer p is* committed *for heap $\mathcal{H}$ and permissions $\bar{\Sigma}$ if modifying p could break another invariant. We denote it $committed(p, \mathcal{H}, \bar{\Sigma})$. Formally: $committed(p, \mathcal{H}, \bar{\Sigma})$ if, and only if there is a value v, a region ρ such that $\rho^{\times} \in \bar{\Sigma}$ or $\rho^{G} \in \bar{\Sigma}$, and a pointer $q \in \mathcal{H}(\rho)$ such that $p \neq q$ and $\mathcal{H}[p \mapsto v] \not\models_{inv} q$.*

Definition 3 (Heap Coherence) *Heap $\mathcal{H}$ is* coherent *for permissions $\bar{\Sigma}$, and we denote it $coh(\mathcal{H}, \bar{\Sigma})$, if:*

1. *for all region ρ, there is at most one permission of $\bar{\Sigma}$ where ρ appears positively (i.e. not at the left of a lollipop $\multimap$);*
2. *for all region ρ, if ρ appears positively in a permission of $\bar{\Sigma}$, then for all pointer p of $\mathcal{H}(\rho)$, we do not have $committed(p, \mathcal{H}, \bar{\Sigma})$;*
3. *for all $\rho^{\emptyset} \in \bar{\Sigma}$, we have $\mathcal{H}(\rho) = \emptyset$ and no permission on a region owned by ρ appear in $\bar{\Sigma}$;*
4. *for all $\rho^{\circ} \in \bar{\Sigma}$, there is p such that $\mathcal{H}(\rho) = \{p\}$;*
5. *for all $\rho^{\times} \in \bar{\Sigma}$, there is p such that $\mathcal{H}(\rho) = \{p\}$ and $\mathcal{H} \models_{inv} p$;*
6. *for all $\rho^{G} \in \bar{\Sigma}$, for all $p \in \mathcal{H}(\rho)$, we have $\mathcal{H} \models_{inv} p$;*
7. *for all $\sigma \multimap \rho \in \bar{\Sigma}$, there is $p \in \mathcal{H}(\rho)$ such that $\mathcal{H}(\sigma) = \{p\}$, and for all $q \in \mathcal{H}(\rho)$, if $q \neq p$ then $\mathcal{H} \models_{inv} q$.*

Item 1 ensures in particular that we cannot have both $\sigma^{\times}$ and σ°, which would lead to errors. However, we can have both $\sigma^{\times}$ and $\sigma \multimap \rho$.

Item 2 ensures that if we have a permission on region ρ, all its transitive owners have been opened correctly. This allows to modify a pointer of ρ without breaking any other invariant than the invariant of this pointer.

Items 3, 4, 5, 6 define how permissions control whether a region is empty, singleton or group, and whether their pointers must verify their invariant. Point 7 describe the particular case of the focus operation.

Operational Semantics Now that we have described precisely the possible states of a program, we can define how this state is modified by each operation. We use the following relation: $e, \mathcal{H} \Longrightarrow v, \mathcal{H}'$ to denote that expression e reduces to value v, while heap $\mathcal{H}$ becomes $\mathcal{H}'$.

The complete set of reduction rules is given in Figure 8. They define the $\Longrightarrow$ relation inductively, the base cases being values and allocation.

Allocation (rule SALLOC) returns a new pointer address p. Notation $\mathcal{H}[p \mapsto ?]$ means that this pointer is uninitialized (its actual value does not matter). It is allocated in empty region ρ: after the allocation, ρ is a singleton region containing only p. Regions owned by ρ (denoted $\mathbf{own}(\rho)$) do not contain any pointer yet.

Adoption (rule SADOPT) takes a pointer of a region σ to put it in another region ρ. This implies a fusion of the region trees, and R becomes $R[\sigma \rightsquigarrow \rho]$.

Focusing (rule SFOCUS) extracts a pointer p from a region ρ to an empty region σ, which becomes singleton. Thus, R becomes $R[\sigma \mapsto \{p\}]$. The region σ is linked to ρ by modifying F, as p belongs both to σ and ρ.

Unfocusing (rule SUNFOCUS) removes the singleton region σ from the heap, so that its pointer may be extracted again in another region. Otherwise, σ could break coherence even if σ is not used anymore. The link $F(\sigma) = \rho$ is removed.

3.4 Soundness Theorem

Now that we have defined our language and its semantics, we prove that a well-typed program is sound. Our main result is that if an invariant is supposed to hold, then it holds. To prove this global invariant, however, we need to prove a stronger property, which is heap coherence (Definition 3, Section 3.3).

Theorem 3.1 (Soundness) *If an expression is well-typed, executing it does not break coherence of the heap. Formally:*

$$\left.\begin{array}{r} \Gamma \vdash e\colon \tau \text{ (a)} \\ \{\bar{\Sigma}\}\ e\ \{\bar{\Sigma}'\} \text{ (b)} \\ coh(\mathcal{H}, \bar{\Sigma}) \text{ (c)} \\ e, \mathcal{H} \Longrightarrow v, \mathcal{H}' \text{ (d)} \end{array}\right\} \quad \textit{implies } coh(\mathcal{H}', \bar{\Sigma}')$$

Moreover, any well-typed expression either reduces to a value or do not terminate. To prove this result, we would need to define a small-step semantics to handle non-terminating expressions. As this is not the focus of this article, we simply remark that if we hide region annotations and typing of permissions, the language we obtain is a classical language which already verifies the property. Our heap $\mathcal{H}$ is expanded with F and R, but they do not prevent the application of any semantic rule.

Proof. by induction on the reduction of e.

1. $!e, \mathcal{H} \Longrightarrow v, \mathcal{H}'$:

   Assume $coh(\mathcal{H}, \bar{\Sigma})$. Rule DEREF gives $\Gamma \vdash e\colon \mathcal{C}[\rho]$. Rule CDEREF gives $\{\bar{\Sigma}\}\ e\ \{\bar{\Sigma}'\}$. Rule SDEREF gives $e, \mathcal{H} \Longrightarrow p, \mathcal{H}'$. By induction, we thus have $coh(\mathcal{H}', \bar{\Sigma}_2)$.

$$\frac{e_1,\ \mathcal{H}_1 \implies v_1,\ \mathcal{H}_2 \qquad e_2[v_1/x],\ \mathcal{H}_2 \implies v_2,\ \mathcal{H}_3}{\textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2,\ \mathcal{H}_1 \implies v_2,\ \mathcal{H}_3}\ \textsc{SLet}$$

$$\frac{e_1,\ \mathcal{H}_1 \implies \mathit{true},\ \mathcal{H}_2 \qquad e_2,\ \mathcal{H}_2 \implies v,\ \mathcal{H}_3}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3,\ \mathcal{H}_1 \implies v,\ \mathcal{H}_3}\ \textsc{SIf1}$$

$$\frac{e_1,\ \mathcal{H}_1 \implies \mathit{false},\ \mathcal{H}_2 \qquad e_3,\ \mathcal{H}_2 \implies v,\ \mathcal{H}_3}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3,\ \mathcal{H}_1 \implies v,\ \mathcal{H}_3}\ \textsc{SIf2}$$

$$\frac{\begin{array}{c}e_1,\ \mathcal{H}_1 \implies \mathit{true},\ \mathcal{H}_2\\ e_2,\ \mathcal{H}_2 \implies (),\ \mathcal{H}_3 \qquad \textbf{while}\ e_1\ \textbf{do}\ e_2,\ \mathcal{H}_3 \implies (),\ \mathcal{H}_4\end{array}}{\textbf{while}\ e_1\ \textbf{do}\ e_2,\ \mathcal{H}_1 \implies (),\ \mathcal{H}_4}\ \textsc{SWhile1}$$

$$\frac{e_1,\ \mathcal{H}_1 \implies \mathit{false},\ \mathcal{H}_2}{\textbf{while}\ e_1\ \textbf{do}\ e_2,\ \mathcal{H}_1 \implies (),\ \mathcal{H}_2}\ \textsc{SWhile2} \qquad \frac{}{v,\ \mathcal{H} \implies v,\ \mathcal{H}}\ \textsc{SValue}$$

$$\frac{\begin{array}{c}e_1,\ \mathcal{H}_1 \implies v_1,\ \mathcal{H}_2\\ \cdots \qquad e_n,\ \mathcal{H}_n \implies v_n,\ \mathcal{H} \qquad f\ \mathit{defined\ by}\ f(x_1,\ \cdots,\ x_n)\ =\ e\\ e[v_1/x_1,\ \cdots,\ v_n/x_n],\ \mathcal{H} \implies v,\ \mathcal{H}'\end{array}}{f(e_1,\ \cdots,\ e_n),\ \mathcal{H}_1 \implies v,\ \mathcal{H}'}\ \textsc{SCall}$$

$$\frac{e,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2 \qquad \mathcal{H}_2(p)\ =\ v}{!e,\ \mathcal{H}_1 \implies v,\ \mathcal{H}_2}\ \textsc{SDeref}$$

$$\frac{e,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2}{\textbf{pack}\ e,\ \mathcal{H}_1 \implies (),\ \mathcal{H}_2}\ \textsc{SPack} \qquad \frac{e,\ \mathcal{H}_1[r \mapsto \emptyset] \implies v,\ \mathcal{H}_2}{\textbf{region}\ r\ \textbf{in}\ e,\ \mathcal{H}_1 \implies v,\ \mathcal{H}_2}\ \textsc{SRegion}$$

$$\frac{e,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2}{\textbf{unpack}\ e,\ \mathcal{H}_1 \implies (),\ \mathcal{H}_2}\ \textsc{SUnpack}$$

$$\frac{e_1,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2 \qquad e_2,\ \mathcal{H}_2 \implies v,\ \mathcal{H}_3}{e_1 := e_2,\ \mathcal{H}_1 \implies (),\ \mathcal{H}_3[p \mapsto v]}\ \textsc{SAssign}$$

$$\frac{p\ \mathit{is\ fresh}}{\textbf{new}\ \mathcal{C}[\rho],\ \mathcal{H} \implies p,\ \mathcal{H}[p \mapsto ?][\rho \mapsto \{p\}][\textbf{own}(\rho) \mapsto \emptyset]}\ \textsc{SAlloc}$$

$$\frac{e\ :\ \mathcal{C}[\sigma] \qquad e,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2}{\textbf{adopt}\ e\ \textbf{in}\ \rho,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2[\sigma \rightsquigarrow \rho]}\ \textsc{SAdopt}$$

$$\frac{e\ :\ \mathcal{C}[\rho] \qquad e,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2}{\textbf{focus}\ e\ \textbf{in}\ \sigma,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2[\sigma \mapsto \{p\}][\sigma \mapsto \rho]}\ \textsc{SFocus}$$

$$\frac{e\ :\ \mathcal{C}[\sigma] \qquad e,\ \mathcal{H}_1 \implies p,\ \mathcal{H}_2}{\textbf{unfocus}\ e\ \textbf{in}\ \rho,\ \mathcal{H}_1 \implies (),\ \mathcal{H}_2[\sigma \mapsto \emptyset][\sigma \mapsto \sigma]}\ \textsc{SUnfocus}$$

Figure 8: Operational semantics

2. $e_1 := e_2, \mathcal{H} \Longrightarrow (), \mathcal{H}'$:

   By induction and typing, we have $\{\bar{\Sigma}\}\ e_1\ \{\bar{\Sigma}_1\}$ and $e_1, \mathcal{H} \Longrightarrow p, \mathcal{H}_1$ with $coh(\mathcal{H}_1, \bar{\Sigma}_1)$. We also have $\{\bar{\Sigma}_1\}\ e_2\ \{\bar{\Sigma}_2\}$ and $e_2, \mathcal{H}_1 \Longrightarrow v, \mathcal{H}_2$ with $coh(\mathcal{H}_2, \bar{\Sigma}_2)$ and $\rho^\circ \in \bar{\Sigma}_2$ where ρ° is the region of e_1. We obtain $coh(\mathcal{H}_2[p \mapsto v], \bar{\Sigma}_2)$ as p is not committed.

3. **new** $\mathcal{C}[\rho], \mathcal{H} \Longrightarrow p, \mathcal{H}'$:

   We have $\mathcal{H}' = \mathcal{H}[p \mapsto ?][\rho \mapsto \{p\}][\mathbf{own}(\rho)^\emptyset \mapsto \emptyset]$. We have $\bar{\Sigma} = \bar{\Sigma}_1, \rho^\emptyset$ and $\bar{\Sigma}' = \bar{\Sigma}_1, \rho^\circ, \mathbf{own}(\rho)^\emptyset$, where ρ does not appear positively in $\bar{\Sigma}_1$. It is enough to show the property of permissions ρ° and $\mathbf{own}(\rho)^\emptyset$. We do have p such that $\mathcal{H}'(\rho) = \{p\}$. Moreover, as p is fresh, no invariant may depend on its value except the invariant of p itself. Permission ρ° implies that modifying p cannot break any invariant. Thus the property of ρ° do hold. Properties of permissions $\mathbf{own}(\rho)^\emptyset$ hold by definition of $\mathcal{H}[\mathbf{own}(\rho)^\emptyset \mapsto \emptyset]$, and because no permission on a region of the form $\rho.r.r'$ is produced, and such a permission could not exist before as we had permission $\rho^\emptyset$.

4. **pack** $e, \mathcal{H} \Longrightarrow (), \mathcal{H}'$:

   By induction we have $e, \mathcal{H} \Longrightarrow p, \mathcal{H}'$. We have $e\colon \mathcal{C}[\rho]$. We have $\bar{\Sigma}_1 = \bar{\Sigma}_1, \rho^\circ, \mathbf{own}(\rho)^G$ and $coh(\mathcal{H}', \bar{\Sigma}_1)$ by induction. We have $\bar{\Sigma}' = \bar{\Sigma}_1, \rho^\times$. For all pointer q of a region $\rho.r$ of $\mathbf{own}(\rho)$, we have permission $\rho.r^G$, so we have $\mathcal{H}' \models_{\mathbf{inv}} q$. Heap $\mathcal{H}'$ satisfies the invariant of p by proof obligation. Thus, $\mathcal{H}' \models_{\mathbf{inv}} p$ and we conclude with $coh(\mathcal{H}', \bar{\Sigma}')$.

5. **unpack** $e, \mathcal{H} \Longrightarrow (), \mathcal{H}'$:

   By induction we have $e, \mathcal{H} \Longrightarrow p, \mathcal{H}'$. We have $e\colon \mathcal{C}[\rho]$. We have $\bar{\Sigma}_1 = \bar{\Sigma}_1, \rho^\times$ and $coh(\mathcal{H}', \bar{\Sigma}_1)$ by induction. We have $\bar{\Sigma}' = \bar{\Sigma}_1, \rho^\circ, \mathbf{own}(\rho)^G$. We must prove that pointers of owned regions are not committed. By syntaxic restriction, the only invariants which depend on a pointer of a region owned by ρ are the invariants of pointers of ρ, or other regions owning ρ transitively which are thus open, so we do not assume these invariants anymore. Thus we have $coh(\mathcal{H}', \bar{\Sigma}')$.

6. **adopt** e **in** $\rho, \mathcal{H} \Longrightarrow p, \mathcal{H}'$:

   We have $e\colon \mathcal{C}[\sigma]$ and $\{\bar{\Sigma}\}\ e\ \{\bar{\Sigma}_1, \sigma^\times, \rho^G\}$. By induction we have $e, \mathcal{H} \Longrightarrow p, \mathcal{H}_1$ and $coh(\mathcal{H}_1, \{\bar{\Sigma}_1, \sigma^\times, \rho^G\})$. We have $\mathcal{H}' = \mathcal{H}_1[\sigma \rightsquigarrow \rho]$ and $\bar{\Sigma}' = \bar{\Sigma}_1, \rho^G$. Removing $\sigma^\times$ cannot break coherence. We fusion σ in ρ; we must prove that with this new value of $\mathcal{H}(\rho)$, coherence is preserved. The only pointer added to ρ is p, whose invariant holds. Fusion of its region tree in ρ ensures that the pointers it owns and on which its invariant may depend are still in its tree.

7. **focus** e **in** $\sigma, \mathcal{H} \Longrightarrow p, \mathcal{H}'$:

   We have $e\colon \mathcal{C}[\rho]$ and $\{\bar{\Sigma}\}\ e\ \{\bar{\Sigma}_1, \sigma^\emptyset, \rho^G\}$. By induction we have $e, \mathcal{H} \Longrightarrow p, \mathcal{H}_1$ and $coh(\mathcal{H}_1, \{\bar{\Sigma}_1, \sigma^\emptyset, \rho^G\}$. We have $\mathcal{H}' = \mathcal{H}_1[\sigma \mapsto \{p\}][\sigma \mapsto \rho]$ and $\bar{\Sigma}' = \bar{\Sigma}_1, \sigma^\times, \sigma \multimap \rho$. We lose permissions $\sigma^\emptyset$ and ρ^G, which cannot break coherence. We gain permission $\sigma^\times$. Region σ is indeed singleton in $\mathcal{H}'$. Moreover, ρ^G gives $\mathcal{H}_1 \models_{\mathbf{inv}} p$, and thus $\mathcal{H}' \models_{\mathbf{inv}} p$ as pointer values have not changed and the tree of p is still accessible thanks to the link $\sigma \mapsto \rho$. Thus the property of $\sigma^\times$ holds. Finally, as the property of ρ^G holds in $\mathcal{H}_1$, it also holds in $\mathcal{H}'$, and thus the property of $\sigma \multimap \rho$ holds.

8. **unfocus** e **in** $\rho, \mathcal{H} \Longrightarrow (), \mathcal{H}'$:

   We have $e\colon \mathcal{C}[\sigma]$ and $\{\bar{\Sigma}\}\ e\ \{\bar{\Sigma}_1, \sigma^\times, \sigma \multimap \rho\}$. By induction we have $e, \mathcal{H} \Longrightarrow p, \mathcal{H}'$ and $coh(\mathcal{H}', \{\bar{\Sigma}_1, \sigma^\times, \sigma \multimap \rho\})$. We have $\bar{\Sigma}' = \bar{\Sigma}_1, \rho^G$. The property of ρ^G holds, as $\sigma \multimap \rho$ gives the invariant of every pointer but the one in σ, whose invariant is given by $\sigma^\times$.

9. **region** r **in** $e, \mathcal{H} \Longrightarrow v, \mathcal{H}'$:

   We have $\{\bar{\Sigma}, r^\emptyset\}\ e\ \{\bar{\Sigma}_1\}$. We have $coh(\mathcal{H}, \bar{\Sigma})$, and thus $coh(\mathcal{H}[r \mapsto \emptyset], \{\bar{\Sigma}, r^\emptyset\})$. By induction we thus have $e, \mathcal{H}[r \mapsto \emptyset] \Longrightarrow v, \mathcal{H}'$ and $coh(\mathcal{H}', \bar{\Sigma}_1)$. We have $\bar{\Sigma}' = \bar{\Sigma}_1 - r$. Removing permissions mentioning r does not break coherence.

10. $f(e_1, \cdots, e_n), \mathcal{H} \Longrightarrow v, \mathcal{H}'$: By induction, $e_1, \cdots, e_n$ reduce to $v_1, \cdots, v_n$ respectively, and the heap we obtain is coherent. Coherence of $\mathcal{H}'$ is given by the fact that f is well-typed, and by applying CWEAK4 to use f in the current context.

This is not enough, however, as weakening rules may be applied at any time. We now prove their soundness as well.

Rule CWEAK1 is sound as removing a permission cannot break coherence. Coherence only gets stronger as permission are added. Rule CWEAK2 is sound as a group region can in particular be empty. Rule CWEAK3 is sound as a group region can in particular be singleton as long as its pointer verifies its invariant, which is given by permission $\rho^\times$.

We now prove that CWEAK4 preserves coherence. We suppose $coh(\mathcal{H}, \{\bar{\Sigma}, \bar{\Sigma}_1\})$ and we prove $coh(\mathcal{H}', \{\bar{\Sigma}, \bar{\Sigma}_2\})$ for each operation e which modifies $\mathcal{H}$ to $\mathcal{H}'$ by consuming $\bar{\Sigma}_1$ and producing $\bar{\Sigma}_2$.

Items 3, 4, 5, 6, 7 are easy to prove by just remarking that the proof we have done before for each operation can also be done by assuming a stronger coherence property with more permissions, and that the properties of these new permissions are preserved by the operation.

The only operations that may break item 1 of coherence are allocation and unpacking, which creates new permissions for regions owned by ρ, from $\rho^\emptyset$ and $\rho^\times$ respectively. This implies that ρ is in $\bar{\Sigma}_1$, so if there is a double permission on the same region it has to be on one of the regions $\rho.r$. This would imply that $\rho.r$ was in $\bar{\Sigma}$ or $\bar{\Sigma}_1$, which is impossible because of $coh(\mathcal{H}, \{\bar{\Sigma}, \bar{\Sigma}_1\})$.

Item 2 can be broken by a *bad couple* of regions (ρ, σ) where σ is transitively owned by ρ and we have permissions $\rho^\times$ or ρ^G and any permission where σ appears positively. Operations that may produce such a bad couple are allocation, packing and unpacking. Allocation and unpacking require region ρ in $\bar{\Sigma}_1$ and produce a permission on $\rho.r$ in $\bar{\Sigma}_2$ which may only produce a bad couple if ρ was already part of a bad couple before the operation. Packing produces a permission $\rho^\times$ by consuming all permissions $\mathbf{own}(\rho)^G$, which means that there was no regions of the form $\rho.r.r'$ used in $\bar{\Sigma}$ or $\bar{\Sigma}_1$, and no bad couple is produced either. □

3.5 Example: Linked Lists

We illustrate our setting on the classical example of linked list data structure. It is made of a sequence of Node structures all in the same region. To allow node updates we typically need more general sequences where the first node is in a different region

val $cons(x\colon \alpha,\ l\colon (\alpha)\langle\rho\rangle Node[\rho])$: unit
 consumes ρ^G **produces** ρ^G
 post $\forall y,\ mem(y, \textbf{result}) \iff$
  $y = x \vee \textbf{old}(mem(y,\ l)) =$
  region ρ_n **in** $\{\rho^G,\ {\rho_n}^{\emptyset}\}$
  let $n = \textbf{new}\ (\alpha)\langle\rho\rangle Node[\rho_n]$ **in**
  $n := !l;$ $\{\rho^G,\ {\rho_n}^{\circ}\}$
  pack $n;$ $\{\rho^G,\ {\rho_n}^{\times}\}$
  let $n = \textbf{adopt}\ n$ **in** ρ **in** $\{\rho^G\}$
  region σ **in** $\{\rho^G,\ \sigma^{\emptyset}\}$
  let $l = \textbf{focus}\ l$ **in** σ **in** $\{\sigma^{\times},\ \sigma \multimap \rho\}$
  unpack $l;$ $\{\sigma^{\circ},\ \sigma \multimap \rho\}$
  $l := Cons(x,\ n);$ $\{\sigma^{\circ},\ \sigma \multimap \rho\}$
  pack $l;$ $\{\sigma^{\times},\ \sigma \multimap \rho\}$
  unfocus l **in** ρ $\{\rho^G\}$

val $nil()$: $(\alpha)\langle\rho\rangle Node[\rho]$
 consumes $\rho^{\emptyset}$ **produces** $\rho^{\times}$
 post $\forall y,\ \neg mem(y, \textbf{result}) =$
  let $n = \textbf{new}\ (\alpha)\langle\rho\rangle Node[\rho]$ **in** $\{\rho^{\circ}\}$
  $n := Nil;$ $\{\rho^{\circ}\}$
  pack $n;\ n$ $\{\rho^{\times}\}$

Figure 9: List functions

from the rest. Thus type Node is parameterized both by the type α of elements and the region ρ of the other nodes:

type $(\alpha)\langle\rho\rangle Node[\rho_n] = Nil \mid Cons\ of\ \alpha \times (\alpha)\langle\rho\rangle Node[\rho]$ **end**

To annotate the list functions with interesting behavioral properties, we introduce a predicate for list membership, specified inductively by two clauses:

inductive $mem : \alpha, (\alpha)\langle\rho\rangle Node[\rho] \to Prop\ :=$
$\mid\ case_{head}\colon \forall x\ n\ n',\ !n = Cons(x,n') \Rightarrow mem(x,n)$
$\mid\ case_{tail}\colon \forall x\ y\ n\ n',\ !n = Cons(y,n') \wedge mem(x,n') \Rightarrow mem(x,n)$

The nil() function of Fig. 9 creates an empty list in a given empty region ρ. The **new** statement consumes $\rho^{\emptyset}$ and produces ρ°, the latter is consumed by **pack** producing $\rho^{\times}$. A typical use of this function is **region** σ **in let** $l = nil()$ **in** $\cdots$ to build a empty list l.

The cons() function in Fig. 9 adds an element at the head of a list. It operates "in-place" modifying the first node to point to a newly allocated node. We do not need any permission on x: lists do not own the data they contain.

3.6 Example: Observer Pattern

A typical example in the literature [23] which illustrates issues with data invariants is the *Observer pattern*, a classical design pattern in OO paradigm.

Observers are objects that can register to a given *subject*, so that they are notified by any changes in the subject state. For simplicity we assume that the internal state is an integer. The Subject type comes with two public methods register and update

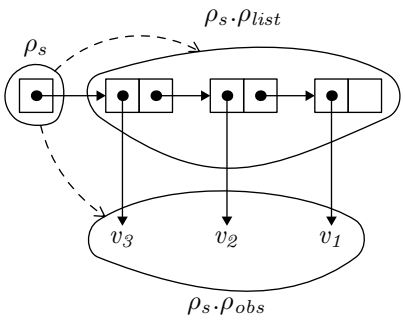

Figure 10: Regions for Observer

which respectively registers observer o to this subject, and updates the internal state with value v and notify the observers. The type Observer comes with a method notify to notify this observer that the new Subject's state is v.

We first decide to put observers in the same region. The invariant we seek relates the subject and the collection of observers, so we declare it in the subject type and we require that the region of observers is owned by the subject.

To represent the collection of observers, we reuse the linked list structure. The Subject and the Observer types are defined below, where owned regions are illustrated in Fig. 10. The subject owns both the region of the linked list and the region of the observer.

type $\langle\rho_s\rangle Observer[\rho_o] = Subject[\rho_s] \times$ int **end**

type $Subject[\rho_s] =$
 own ρ_{list}, ρ_{obs}
 $((\langle\rho_s\rangle Observer[\rho_{obs}])\ \langle\rho_{list}\rangle\ Node[\rho_{list}]) \times$ int
 inv$(this) = \forall o\colon \langle\rho_s\rangle Observer[\rho_{obs}];\ mem(o, !this.1) \Rightarrow !o.2 = !this.2$
end

The functions for registration and updates are as follows.

val $register(this\colon Subject[\rho_s],\ o\colon \langle\rho_s\rangle\ Observer[\rho_o])$: unit
 consumes ${\rho_s}^{\times}, {\rho_o}^{\times}$ **produces** ${\rho_s}^{\times} =$

unpack $this$;	$\{{\rho_s}^{\circ}, {\rho_o}^{\times}, \rho_s.{\rho_{list}}^{G}, \rho_s.{\rho_{obs}}^{G}\}$
let $o =$ **adopt** o **in** $\rho_s.\rho_{obs}$ **in**	$\{{\rho_s}^{\circ}, \rho_s.{\rho_{list}}^{G}, \rho_s.{\rho_{obs}}^{G}\}$
$cons(o, !this.1)$;	$\{{\rho_s}^{\circ}, \rho_s.{\rho_{list}}^{G}, \rho_s.{\rho_{obs}}^{G}\}$
$notify(o, !this.2)$;	$\{{\rho_s}^{\circ}, \rho_s.{\rho_{list}}^{G}, \rho_s.{\rho_{obs}}^{G}\}$
pack $this$	$\{{\rho_s}^{\times}\}$

val $notify(this\colon \langle\rho_s\rangle\ Observer[\rho_o],\ v$:int): unit **consumes** ${\rho_o}^{G}$ **produces** ${\rho_o}^{G} =$

region σ **in**	$\{{\rho_o}^{G}, \sigma^{\emptyset}\}$
let $o =$ **focus** $this$ **in** σ **in**	$\{\sigma^{\times}, \sigma \multimap \rho_o\}$
unpack o;	$\{\sigma^{\circ}, \sigma \multimap \rho_o\}$
$o := (!o.1, v)$; **pack** o;	$\{\sigma^{\times}, \sigma \multimap \rho_o\}$
unfocus o **in** ρ_o	$\{{\rho_o}^{G}\}$

val $update(this\colon Subject[\rho_s],\ n\colon \text{int})$: unit **consumes** ${\rho_s}^\times$ **produces** ${\rho_s}^\times =$

unpack $this$;	$\{{\rho_s}^\circ, {\rho_s.\rho_{list}}^G, {\rho_s.\rho_{obs}}^G\}$
$this := (!this.1,\ n)$;	$\{{\rho_s}^\circ, {\rho_s.\rho_{list}}^G, {\rho_s.\rho_{obs}}^G\}$
foreach o **in** $!this.1$ **do**	
$notify(o,\ n)$;	$\{{\rho_s}^\circ, {\rho_s.\rho_{list}}^G, {\rho_s.\rho_{obs}}^G\}$
pack $this$	$\{{\rho_s}^\times\}$

An important remark is that $\rho_\sigma.\rho_{obs}$ is owned by ρ_σ but not by $\rho_\sigma.\rho_{list}$, whereas pointers of $\rho_s.\rho_{obs}$ lie in $\rho_\sigma.\rho_{list}$. In other words, the ownership tree is not based on the pointer structure.

Finally, note that in an actual implementation, the subject should not own the observer but only the part which appears in the invariant. In this example, the subject would no longer own ρ_{obs}; instead, it would own a region ρ, and the int value of the observer would be replaced by a pointer of ρ.

3.7 Example: Hash Tables

Assume a type Key and a function $hash(x\colon Key[\rho])$: int. A hash table indexed by Key is an implementation of finite maps from Key to some type α. Data are arranged in an array according to the hash value of their key, modulo the array size. Elements of same hash are stored in small linked lists in each array cell.

Key being a mutable type, it would be a major mistake to allow the keys to be mutated after they are added in the table: their hash value would probably change and the value would then be stored in the wrong place. Mutation of keys can be forbidden using permissions. First, we decide that the regions of keys is owned by the hash table. It also owns the region of the small linked lists.

type $(\alpha)Hashtbl[\rho] =$
own $\rho_{keys}, \rho_{lists}$
$((Key[\rho_{keys}] \times \alpha)\ \langle\rho_{lists}\rangle\ Node[\rho_{lists}])$ **array**
end

Second, the $add()$ function consumes the given permission on the key without returning it, which forbids future mutation of it.

val $add(h\colon (\alpha)\ Hashtbl[\rho],\ k\colon Key[\rho_k],\ d\colon \alpha)$: unit
consumes ${\rho}^\times, {\rho_k}^\times$ **produces** ${\rho}^\times =$

unpack h;	$\{\rho^\circ, {\rho.\rho_{keys}}^G, {\rho.\rho_{lists}}^G, {\rho_k}^\times\}$
let $k =$ **adopt** k **in** $\rho.\rho_{keys}$ **in**	$\{\rho^\circ, {\rho.\rho_{keys}}^G, {\rho.\rho_{lists}}^G\}$
let $n = hash(k)$ **mod** $len(!this)$	$\{\rho^\circ, {\rho.\rho_{keys}}^G, {\rho.\rho_{lists}}^G\}$
in $cons(!this.(n),\ (k,\ d))$;	$\{\rho^\circ, {\rho.\rho_{keys}}^G, {\rho.\rho_{lists}}^G\}$
pack h	$\{{\rho}^\times\}$

Remark that another implementation would be to make a full deep copy of the data structure of the given key k, which would then make it possible to produce the permission on it again. The important point here is that permissions allow to use the more efficient implementation without copy in a safe way, regardless of the complexity of the key data type and its invariant. As soon as the permission is consumed by adding the key into the table, it is known that its internal invariants cannot be violated: since permission has been given up, the rest of the program cannot mutate it anymore.

4 Data Abstraction

We now consider the classical approach to modular programming: a *module* is a set of type declarations and function definitions, whereas an *interface* for such a module is a set of type declarations and function profiles. Any type declaration can be made *abstract* by giving only the name but hiding the owned regions, the pointed type value and the invariant. Function profiles give the type of parameters and returned value, permissions consumed and produced, and also possible pre- and postconditions.

4.1 Abstraction Theorem

Let's assume a module M with interface I.

Definition 4 *A function of M implemented by an expression E soundly implements its profile in I:*

val $f(x_1, \cdots, x_k)$ **consumes** $\bar{\Sigma}$ **produces** $\bar{\Sigma}'$ **pre** P **post** P'

if (1) P and P' refer only to data visible in I, (2) for any concrete execution:

$$E[v_1/x_1, \cdots, v_k/x_k], \mathcal{H} \Longrightarrow v, \mathcal{H}'$$

if $\mathcal{H} \models \mathbf{pre}(f)[v_1/x_1, \cdots, v_k/x_k]$ *then:*

$$\mathcal{H}, \mathcal{H}', v \models \mathbf{post}(f)[v_1/x_1, \cdots, v_k/x_k, v/\mathbf{result}]$$

A module M soundly implements an interface I if each function f of I is soundly implemented in M.

Following this notion of abstraction, it is possible to extend our operational semantics to allow calls to abstract functions. We add a new semantic rule:

$$\frac{\begin{array}{c} e_1, \mathcal{H}_1 \Longrightarrow v_1, \mathcal{H}_2 \quad \cdots \quad e_k, \mathcal{H}_k \Longrightarrow v_k, \mathcal{H} \\ \mathcal{H} \models \mathbf{pre}(f)[v_1/x_1, \ldots, v_k/x_k] \\ \mathcal{H}, \mathcal{H}', v \models \mathbf{post}(f)[v_1/x_1, \ldots, v_k/x_k, v/\mathbf{result}] \end{array}}{f(e_1, \ldots, e_k), \mathcal{H}_1 \Longrightarrow v, \mathcal{H}'}$$

Theorem 4.1 *If a program P is proved to satisfy its specification with respect to an abstract interface I, then it is correct with respect to any module M which soundly implements interface I.*

Proof. When we reason on program P w.r.t interface I, a reasoning step on a function call to some f in I is performed via the following deduction rule: if $f(v_1, \ldots, v_k), \mathcal{H} \Longrightarrow v, \mathcal{H}'$ and $\mathcal{H} \models \mathbf{pre}(f)$ then $\mathcal{H}, \mathcal{H}', v \models \mathbf{post}(f)$. If we consider an implementation M of I, we have to show that this deduction is correct.

Any concrete execution of this function call has the form $b[v_1/x_1, \ldots, v_k/x_k], \bar{\mathcal{H}} \Longrightarrow v, \bar{\mathcal{H}}'$ where $\bar{\mathcal{H}}$ (resp. $\bar{\mathcal{H}}'$) denote the heap $\mathcal{H}$ (resp. $\mathcal{H}'$) augmented with private data of M.

Assuming $\mathcal{H} \models \mathbf{pre}(f)$ we immediately have $\bar{\mathcal{H}} \models \mathbf{pre}(f)$ since $\mathcal{H} \subseteq \bar{\mathcal{H}}$. From the concrete execution step $b[v_1/x_1, \ldots, v_k/x_k], \bar{\mathcal{H}} \Longrightarrow v, \bar{\mathcal{H}}'$ and the hypothesis that M soundly implements I, we get $\bar{\mathcal{H}}, \bar{\mathcal{H}}', v \models \mathbf{post}(f)$. But since $\bar{\mathcal{H}} = \mathcal{H} \cup P$ and $\bar{\mathcal{H}}' = \mathcal{H}' \cup P'$ where P and P' give the values of private data in M, and those do not appear in $\mathbf{post}(f)$, we get $\mathcal{H}, \mathcal{H}', v \models \mathbf{post}(f)$. □

Corollary 4.1 *The side-effects occurring in regions of a module that are invisible in the interface can be safely ignored from the outside of the module: they cannot violate any external invariant, and external code cannot violate invariants in such a region.*

We now illustrate these results on a few examples.

4.2 Example: Counter

We illustrate our abstraction theorem on a very simple example. Type $Counter$ provides some arbitrary function f, and additionally counts how many times that function is called. The concrete program is:

type $Integer[\rho] =$ int **end**

type $Counter[\rho]$
 own ρ_c
 $Integer[\rho_c]$
 inv$(this) = !(!this) \geq 0$
end

val $createCounter() : Counter[\rho]$
 consumes $\rho^\emptyset$ **produces** $\rho^\times =$
 region ρ_i **in** $\{ \rho^\emptyset, {\rho_i}^\emptyset \}$
 let $i =$ **new** $Integer[\rho]$ **in** $i := 0;$ $\{ \rho^\emptyset, {\rho_i}^\circ \}$
 pack $i;$ $\{ \rho^\emptyset, {\rho_i}^\times \}$
 let $c =$ **new** $Counter[\rho]$ **in** $c := i;$ $\{ \rho^\circ, {\rho_i}^\times \}$
 pack c $\{ \rho^\times \}$

val $f(c\text{: } Counter[\rho], <extra\ args>)$: $<return$ **type**$>$
 consumes $\rho^\times$ **produces** $\rho^\times =$
 unpack $c;$ $\{ \rho^\circ, \rho.{\rho_c}^G \}$
 region σ **in** $\{ \rho^\circ, \sigma^\emptyset, \rho.{\rho_c}^G \}$
 let $i =$ **focus** $!c$ **in** σ **in** $\{ \rho^\circ, \sigma^\times, \sigma \multimap \rho.\rho_c \}$
 unpack $i;$ $\{ \rho^\circ, \sigma^\circ, \sigma \multimap \rho.\rho_c \}$
 $i := !i{+}1;$ $\{ \rho^\circ, \sigma^\circ, \sigma \multimap \rho.\rho_c \}$
 pack $i;$ $\{ \rho^\circ, \sigma^\times, \sigma \multimap \rho.\rho_c \}$
 unfocus i **in** $\rho.\rho_c;$ $\{ \rho^\circ, \rho.{\rho_c}^G \}$
 pack $c;$ $\{ \rho^\times \}$
 $<remaining\ computation\ of\ f>$

val $nbCalls(c\text{: } Counter[\rho])$: int $= !(!c)$

We abstract this module by:

type $Counter[\rho]$
val $f(c\text{: } Counter[\rho], <extra\ args>)$: $<return$ **type**$>$
 consumes $\rho^\times$ **produces** $\rho^\times$
val $nbCalls(c\text{: } Counter[\rho])$: int

Note that $\rho.\rho_c$ is hidden and thus not useable by the rest of the program. From our theorem, it is statically known that on the one hand the invariant of c is maintained; and on the other hand, the side-effect on c when calling f can be safely hidden as part

of the modifications to ρ: no other invariant in the program can be violated by such a call.

In other words, our permission-based approach allows a modular reasoning on data invariants, with a true separation of the internal side-effects of a module and the external ones.

4.3 Example: Memoization

Memoization is a general technique used to improve the time complexity of a recursive function f. A private table records the pairs $(x, f(x))$ for later reuse.

Let f be the classical Fibonacci function computed by the recursive formula $f(n) = f(n-1) + f(n-2)$[1]. The following data structure Fib computes f using memoization. It reuses the hash tables of Section 3.7 for keys of type $Integer$.

type $Fib[\rho] =$
 own ρ_{hash}, ρ_{data}
 $(Integer[\rho_{data}])\ Hashtbl[\rho_{hash}]$
 inv$(this) = \forall x, y; mem((!x, !y), !this) \Rightarrow y = fib(x)$
end

Predicate mem is similar to the one used by lists, and *fib* is a definition of the Fibonacci function in the logic. The function computing Fibonacci is:

val *fibo*(x: $Fib[\rho]$, n: int): int	
consumes $\{\rho^\times\}$ **produces** $\{\rho^\times\}$	
pre $(n \geq 0)$ **post** $(\textbf{result} = fib(n)) =$	
if $n \leq 1$ **then** 1 **else**	
try $find(!x, n)$ **with** $notFound \Rightarrow$	
let $y = fibo(x, n-1)$ **in**	
let $z = fibo(x, n-2)$ **in**	
region σ **in**	$\{\rho^\times, \sigma^\emptyset\}$
let $k =$ **new** $Integer[\sigma]$ **in** $k := n$;	$\{\rho^\times, \sigma^\circ\}$
pack k;	$\{\rho^\times, \sigma^\times\}$
region σ' **in**	$\{\rho^\times, \sigma^\times, \sigma'^\emptyset\}$
let $i =$ **new** $Integer[\sigma']$ **in**	$\{\rho^\times, \sigma^\times, \sigma'^\circ\}$
$i := y + z$;	$\{\rho^\times, \sigma^\times, \sigma'^\circ\}$
pack i;	$\{\rho^\times, \sigma^\times, \sigma'^\times\}$
unpack x;	$\{\rho^\circ, \rho.\rho_{hash}{}^G, \rho.\rho_{data}{}^G, \sigma^\times, \sigma'^\times\}$
let $i =$ **adopt** i **in** $\rho.\rho_{data}$ **in**	$\{\rho^\circ, \rho.\rho_{hash}{}^G, \rho.\rho_{data}{}^G, \sigma^\times\}$
region σ_x **in**	$\{\rho^\circ, \rho.\rho_{hash}{}^G, \rho.\rho_{data}{}^G, \sigma^\times, \sigma_x{}^\emptyset\}$
let $y =$ **focus** $!x$ **in** σ_x **in**	$\{\rho^\circ, \sigma_x \multimap \rho.\rho_{hash}, \rho.\rho_{data}{}^G, \sigma^\times, \sigma_x{}^\times\}$
$add(y, n, i)$;	$\{\rho^\circ, \sigma_x \multimap \rho.\rho_{hash}, \rho.\rho_{data}{}^G, \sigma_x{}^\times\}$
unfocus y **in** $\rho.\rho_{hash}$;	$\{\rho^\circ, \rho.\rho_{hash}{}^G, \rho.\rho_{data}{}^G\}$
pack x; $!i$	$\{\rho^\times\}$

An abstract interface for the module is:

type $Fibo[\rho]$
val *fibo*(x: $Fib[\rho]$, n:int): int
 consumes $\{\rho^\times\}$ **produces** $\{\rho^\times\}$ **pre** $n \geq 0$ **post result** $= fib(n)$

The region of the $Hashtbl$ is hidden: we can safely ignore its modifications.

[1] There are more efficient implementations, but it illustrates our point.

5 Prototype and Experimentations

The language described in this paper is now implemented in a stand-alone prototype called *Capucine*. It is freely available on the web page `http://romain.bardou.fr/capucine`, together with concrete examples. Please look at this web page for up-to-date experimental data. In this section we quickly describe the implementation, and present a detailed experiment.

5.1 The Capucine prototype

First, it is worth noticing that the first job of Capucine is to typecheck a given code with respect to regions and permissions. But Capucine also uses an inference algorithm to relieve the user from having to annotate the code with pack, unpack, focus and such statements. In practice, most of the expected annotations are correctly inferred, as shown by the examples that follow. But of course not everything is inferred: the user still must declare the permissions consumed and produced by functions, and *focus* statements must be given.

The third job is not to execute the code but is to generate verification conditions to check that data invariants are preserved, and also any other user-defined properties given as assertions in the code, or pre- and post-conditions to functions. The generation of verification conditions proceeds by first generating intermediate code in the input language of the Why VC generator [14], and then calling the Why tool to proceed with VC generation and calls to several external provers.

In addition to providing the core language described before, Capucine offers these functionalities:

- in the input language, modeling of data-types is permitted via algebraic-style declarations, in the same flavor as Why, or other specification languages such as ACSL [5];
- syntactic sugar constructs are given, such as declarations of field names as alternatives to notations `.1`, `.2`, etc (declaration **`selector`** below);
- inference of pack, unpack, adopt and unfocus when needed, as described above.

The prototype is available on the web page `http://romain.bardou.fr/capucine`, together with a few concrete examples. Below we illustrate the prototype on an example given on the so-called VACID-0 benchmarks [18].

5.2 Example: Constant-Time Sparse Arrays

The following example of *constant-time sparse arrays* is inspired from the VACID-0 benchmarks [18] (`http://vacid.codeplex.com/`). The pseudo-java code for it is as follows:

```
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); // (1)
      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);
  }
}
```

The SparseArray class implements the data structure displayed as follows. Assuming three elements x y z were added, in this order, at indexes a, b, and c respectively, then the three arrays look like this:

```
              b     a      c
val   +-----+-+---+-+----+-+----+
      |     |y|   |x|    |z|    |
      +-----+-+---+-+----+-+----+

idx   +-----+-+---+-+----+-+----+
      |     |1|   |0|    |2|    |
      +-----+-+---+-+----+-+----+

       0 1 2  n=3
back  +-+-+-+------------------+
      |a|b|c|                  |
      +-+-+-+------------------+
```

Thus, the first element x was given the index 0 in idx, and its real index a is stored in cell 0 of array back. With this data structure, one can access a previously inserted element at index d as follows: (1) get the internal index i stored in idx[d], (2) if not $0 \leq i < n$ then surely no element of index d was inserted, hence we can return the

default value, (3) if $0 \leq i < n$ then look at index d' = back[i], (4) if d' = d then there is indeed an element inserted with index d, hence return val[d], otherwise return the default. What is clever in this implementation is that arrays do not need to be initialized: this data-strucutre implements arrays of constant-time operations set and get, but also constant-time creation.

The verification tasks are given by the assertions in the code above. The method `sparseArrayTestHarness()` allows to test the implementation against simple data. These tasks may appear easy at first, but they are not so easy because we want to reason modularly on the code. This means we need to give specifications to our methods that would apply to any instance of use. In particular, a tricky point is that the main program calls the `create()` method twice, and we need to specify that the resulting data structure is fresh. This means that one needs a methodology that support some kind of seperation: for example the client program should know that the `create()` method does produce an object with fresh internal arrays. It is thus a interesting example for our methodology based on regions and permissions.

5.2.1 Capucine arrays

The Capucine concrete code starts as follows. First, we need to defined the structure of arrays: this is defined by a pair of its length, together with a pure logic infinite array:

```
logic type array (a)
logic function store (array (a), int, a): array (a)
logic function select (array (a), int): a

axiom select_eq: forall a: array (a). forall i: int.
  forall v: a. [select(store(a, i, v), i)] = [v]

axiom select_neq: forall a: array (a). forall i: int.
  forall j: int. forall v: a.
  [i] <> [j] ==> [select(store(a, i, v), j)] = [select(a, j)]

(* Array Reference *)
selector (length, cell)
class Array (a) =
  (int * array (a))
  invariant(this) = [0] <= [this.length]
end

val array_create(size:int): Array (a) [R]
  consumes R^e (* empty region *)
  produces R^c (* closed singleton region *)
  requires [0] <= [size]
  ensures [!result.length] = [size]
  =
    let tmp = new Array (a) in
    tmp := (size, !tmp.cell); tmp

val array_get(this: Array (a) [R], i:int) : a
  requires [0] <= [i] and [i] < [!this.length]
  ensures [result] = [select(!this.cell,i)]
  =
    if 0 <= i and i < !this.length then select(!this.cell,i)
```

```
    else ((while true do ());select(!this.cell,0))

val array_set(this: Array (a) [R], i:int, v:a) : unit
  consumes R^c (* closed singleton region *)
  produces R^c (* closed singleton region *)
  requires [0] <= [i] and [i] < [!this.length]
  ensures [!this.length] = [old(!this.length)] and
          [!this.cell] = [store(old(!this.cell),i,v)]
  =
    if 0 <= i and i < !this.length then
       (this := (!this.length, store(!this.cell,i,v)))
    else (while true do ())
```

Notice that since Capucine supports parametric polymorphism, we can declare polymorphic array data structure. Functions get and set are properly equipped with preconditions so as to check the absence of out-of-bounds array accesses. The occurences of (while (true) do ()) are just tricks for encoding an "assert false" statement for unreachable branches.

5.2.2 Capucine sparse array structure

The Capucine concrete code for sparse arrays continues as follows.

```
predicate interval(a:int, x:int, b:int) =
     [a] <= [x] and [x] < [b]

selector (value, idx, back, n, default, size)
class Sparse (a) =
 own Rval, Ridx, Rback;

 (Array (a) [Rval] * (* 1: value *)
  Array (int) [Ridx] * (* 2: idx *)
  Array (int) [Rback] * (* 3: back *)
  int * (* 4: n *)
  a * (* 5: default *)
  int (* 6: size *))

 invariant (x) =
  [0] <= [x.n] and [x.n] <= [x.size] and
  [!(x.value).length] = [x.size] and
  [!(x.idx).length] = [x.size] and
  [!(x.back).length] = [x.size] and
  forall i: int. interval([0],[i],[x.n]) ==>
   interval([0],[select (!(x.back).cell, i)],[x.size]) and
   [select (!(x.idx).cell, select (!(x.back).cell, i))] = [i]
end

predicate is_elt(a: Sparse (a) [R], i: int) =
 [0] <= [select (!(!a.idx).cell, i)] and
 [select (!(!a.idx).cell, i)] < [!a.n] and
 [select (!(!a.back).cell, select (!(!a.idx).cell, i))] = [i]

logic function model (Sparse (a) [R], int): a
```

```
axiom model_in:
 forall a: Sparse (a) [R]. forall i: int. is_elt([a], [i])
   ==> [model(a, i)] = [select(!(!a.value).cell, i)]

axiom model_out:
 forall a: Sparse (a) [R]. forall i: int. not is_elt([a], [i])
   ==> [model(a, i)] = [!a.default]
```

The invariant formalizes the picture given at the beginning of this section. The predicate `is_elt`(a, i) tells whether the index i is defined in array a, as was explained before.

Finally, the logic function `model` provides the logic model of the sparse array, that is `model`(a, i) returns the element currently associated with index i in array a. It is defined axiomatically because Capucine does yet support direct definitions of logic functions.

5.2.3 Capucine sparse array code and test harness

We are now ready to provide the Capucine code corresponding to the pseudo-Java code, as follows.

```
val create(sz:int, def: a): Sparse (a) [R]
  consumes R^e
  produces R^c
  requires
    [0] <= [sz]
  ensures
    [!result.size] = [sz] and
    forall i: int. [model (result, i)] = [def]
  =
    let arr = new Sparse (a) in
    arr := (array_create (sz), array_create (sz),
            array_create (sz), 0, def, sz);
    arr

val get(a: Sparse (a) [R], i: int): a
  consumes R^c
  produces R^c
  requires [0] <= [i] and [i] < [!a.size]
  ensures [result] = [model(a, i)]
  =
    let index = array_get(!a.idx, i) in
    if 0 <= index and index < !a.n
       and array_get(!a.back, index) = i
    then
       array_get(!a.value, i))
    else
       !a.default

val set(a: Sparse (a) [R], i: int, v: a): unit
  consumes R^c
  produces R^c
  requires [0] <= [i] and [i] < [!a.size]
  ensures
```

```
    [!a.size] = [old(!a.size)] and
    (forall j: int. [j] <> [i] ==>
       [model(a, j)] = [old(model(a, j))]) and
    [model(a, i)] = [v]
  =
    array_set((focus !a.value), i, v);
    let index = array_get(!a.idx, i) in
    if not (0 <= index and index < !a.n and
            array_get(!a.back, index) = i)
    then (
      assert [!a.n] < [!a.size]; (* (1) *)
      array_set((focus !a.idx), i, !a.n);
      array_set((focus !a.back), !a.n, i);
      a := (!a.value, !a.idx, !a.back,
            !a.n + 1, !a.default, !a.size);
    )

val main(): unit =
  region Ra: Sparse (int) in
  region Rb: Sparse (int) in
  let default = 0 in
  let a = (create(10,default): Sparse (int) [Ra]) in
  let b = (create(20,default): Sparse (int) [Rb]) in
  let x = get(a, 5) in
  let y = get(b, 7) in
  assert ([x] = [default] and [y] = [default]);
  set(a, 5, 1);
  set(b, 7, 2);
  let x = get(a, 5) in
  let y = get(b, 7) in
  assert ([x] = [1] and [y] = [2]);
  let x = get(a, 7) in
  let y = get(b, 5) in
  assert ([x] = [default] and [y] = [default]);
  let x = get(a, 0) in
  let y = get(b, 0) in
  assert ([x] = [default] and [y] = [default]);
  let x = get(a, 9) in
  let y = get(b, 9) in
  assert ([x] = [default] and [y] = [default]);
  assert false (* poor man's check for inconsistency *)
```

Notice that the get function needs permission on its argument because the invariant is required to hold to prove the pre-condition of the call to `array_get()`. Such a requirement could be relaxed if we allowed *fractional permissions* [7] such as they are used in concurrent separation logic [6] or as they are implemented in Chalice [19].

5.2.4 Running VC generation and proof

Running the capucine tool on this file generates a few VCs which are displayed on figure 11.

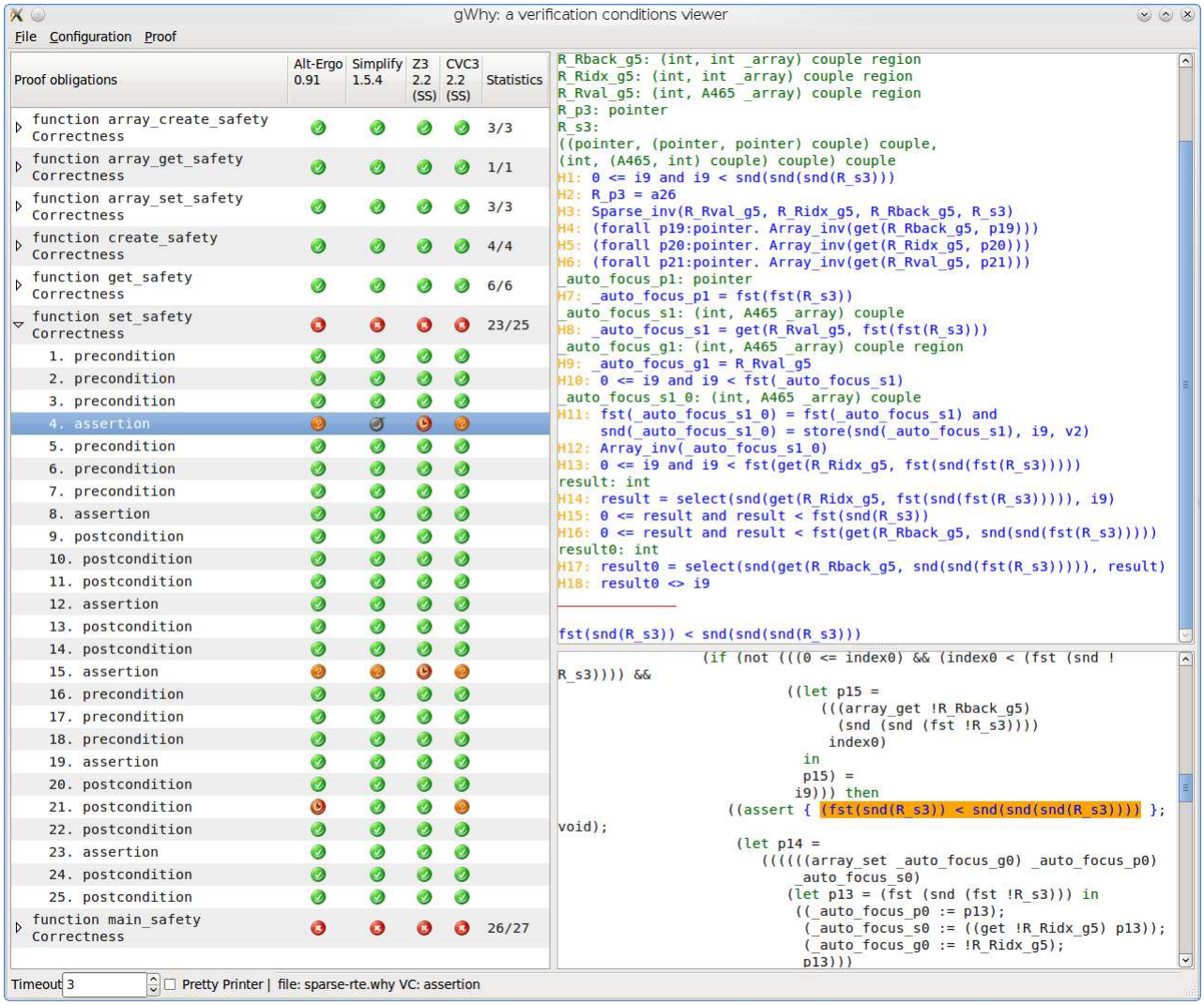

Figure 11: VCs for sparse arrays in Why GUI

Three VCs are not discharged, but the other ones are proved by all the provers Alt-Ergo, Simplify, Z3 and CVC3, except one proved only by Simplify and Z3. This one corresponds to the post-condition of function set related to the model.

The reasons for the three unproved VCs are as follows.

- The first two correspond to the assertion (1). It appears twice because of the if condition having different ways of being true (this duplication is a typical effect of the interpretation of lazy conjunction in Why).

  Assertion (1) is a tricky one because proving it amounts to apply the pigeon-hole principle: if n becomes equal to $size$, then we know that idx and $back$ provide a permutation of $[0..size-1]$, so we know that each index was filled, a contradic-

tion. These reasoning is unfortunately out-of-reach of automated provers so we need to assume this assertion correct.

- The last unproved VC is indeed fortunately not proved, because it corresponds to the last assertion of the test harness: we assert false as a way to test for possible inconsistency, which might be introduced by the axioms we stated. This is not of course a guaranty of consistency, but it is quite reassuring.

6 Related Works

This is a continuation of our previous work [2]. Main improvements are the addition of regions and permissions, and static association of invariants to pointers.

The existing related works are essentially of two kinds. A first kind of work aims at building advanced type systems to control aliasing or access to resources, to provide a static ownership policy, etc. Regions in type systems were introduced by Jouvelot and Talpin in 1991 [28] and used for memory management [29]. The notion of ownership is due to Clarke, Potter and Noble [10] with the goal to control aliasing. The notions of permissions or *capabilities* were introduced by Crary, Walker and Morrisett [11] and operations of adoption, focus and unfocus by Fahndrich and Deline [13]. These provide very advanced static typing for memory management, but no attempt was proposed yet to apply such approaches to deductive verification [9].

The second kind of work is, on the other hand, aimed at verifying behavioral properties by theorem proving. Separation Logic [24] is a famous approach which builds in the logic concepts of non-aliasing and of access capability. Yet, everything is pushed into the logic hence delegated to a prover, no advanced static typing is involved. Ownership for deductive verification was proposed by Barnett et al. [3] in 2004, providing for the first time a sound methodology for preserving invariants. Their ownership notion resides also in the logic, not in a type system. In that approach, the ownership relation is defined by declaring which class fields are owned. By using regions, in our setting we allow ownerships links that do not follow pointer links, as in the Observer example. *Universe types* [12, 22] add static typing to ownership but still disallows ownership links different from pointer links. *Regional logic* [1] allows general ownership structures as we do, but everything is handled dynamically: regions are *ghost* fields that the programmer must update. *Dynamic frames* provides a similar approach by declaring regions as specification variables denoting memory footprints, on which it is possible to reason with on the theorem proving side. As ownership, dynamic frames do not provide advanced static typing, but has advantages close to what can be done in separation logic: the logic formulas contain static information about memory structure.

Our work aims at bridging the gap between the two kinds of work, by pushing as much information as possible into static typing. Our type systems are not as complex as say [9], for example we do not support *strong update* which allow type to change during execution. However, our approach does not require any permission for read access, which was needed to allow pointer dereferencing in specifications. Notice finally that the separation logic approach is clearly interesting to integrate in our setting, for example to reason locally on a given region like in our Hashtbl example where the small linked list all lie in the same region [16].

7 Conclusion and Future Works

We tackle data invariant preservation by combining static typing and theorem proving. Static typing is used as much as possible to specify memory regions and permissions to modify them. Future works include the following directions.

- Adapt the technique to object-oriented programs, possibly by using our language as an intermediate language [20]. The support for dynamic calls must be studied, and properties like behavioral subtyping [8] should be needed.
- Our abstraction result should be extended to *refinement*, where concrete private fields could be modeled by abstract, pure datatype in interfaces [21, 27].

Acknowledgments

We thank François Pottier, Arthur Charguéraud and Sylvain Boulmé for fruitful discussion about capabilities, and all the other members of the CeProMi project for discussions on modular verification of pointer programs in general. We also thank Jean-Christophe Filliâtre and Andrei Paskevich for providing an initial set of annotations for the sparse arrays example.

References

[1] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, Paphos, Cyprus, July 2008.

[2] R. Bardou. Ownership, pointer arithmetic and memory separation. In *Formal Techniques for Java-like Programs (FTfJP'08)*, Paphos, Cyprus, July 2008.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[5] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.

[6] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL 2005*, pages 259–270, New York, NY, USA, 2005. ACM.

[7] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[8] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

[9] A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 213–224, Sept. 2008.

[10] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM Press, 1998.

[11] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 262–275. ACM Press, 1999.

[12] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[13] M. Fahndrich and R. Deline. Adoption and focus: practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, volume 37, pages 13–24, May 2002.

[14] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[15] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming language design and implementation (PLDI)*, pages 282–293, 2002.

[16] N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *Types in Language Design and Implementation (TLDI)*, pages 105–116. ACM, 2009.

[17] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.

[18] K. R. M. Leino and M. Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.

[19] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[20] C. Marché. Jessie: an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM.

[21] C. Marché. Towards modular algebraic specifications for pointer programs: a case study. In *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 235–258. Springer, 2007.

[22] P. Müller and A. Rudich. Ownership transfer in universe types. In *ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA)*, pages 461–478. ACM, 2007.

[23] M. Parkinson. Class invariants: The end of the road? In T. Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO), in conjunction with ECOOP 2007*, Berlin, Germany, July 2007. `http://www.cs.purdue.edu/homes/wrigstad/iwaco/`.

[24] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[25] W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. `http://www.cs.ru.nl/~tews/cv07/cv07-smans.pdf`.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[27] A. Tafat, S. Boulmé, and C. Marché. A refinement methodology for object-oriented programs. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software, Papers Presented at the International Conference*, Karlsruhe Reports in Informatics, pages 143–159, Paris, France, June 2010. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083`.

[28] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[29] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. Academic Press.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399