Ownership, Pointer Arithmetic and Memory Separation

Romain Bardou*

ENS Cachan, F-94230
& Univ Paris-Sud, CNRS, Orsay F-91405
& INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893

Abstract. Ownership systems provide a way to reason about data structures in a hierarchical fashion. We propose a small but extensible language featuring an ownership system and data invariants. It is then extended with pointer arithmetic, showing how to specify array invariants. We show how to express the global properties of the ownership system in the logic. This method can be used with a memory model featuring memory separation, and provides a practical way to use invariants in deductive verification of programs. We implemented the proposed system in the Why platform and applied it to C and Java programs.

1 Introduction

One way of proving computer program properties is to annotate programs with *specifications*, and then use a *verification condition generator* (VCG) to produce proof obligations. These obligations, once proven, ensure the adequation of programs with respect to their specification.

The specification language can feature *data invariants*. For instance, a balanced binary search tree has two invariants: it is sorted, and it is balanced. A program breaking the "balanced" invariant will not be as efficient; and a program breaking the "sorted" invariant will not even be sound. Thus, the VCG must produce proof obligations ensuring that data invariants are not broken. For instance, the soundness of a search function might rely on the "sorted" invariant. Handling data invariants in a sound way is not trivial, in particular for object-oriented programs [12].

A key issue is to choose an *invariant policy* saying when an invariant is supposed to hold. A *strong* invariant policy, where invariants hold permanently, is too constraining. For instance, when inserting a new element into a binary search tree, its invariants might temporarily be broken. In the Java Modeling Language (JML) [5], invariants are supposed to hold at method boundaries for all *accessible* objects. This policy is difficult to support, both for static and dynamic verification tools.

Ownership is a particular invariant policy where objects can own other objects. By preventing owned objects to be modified, the language can allow the invariant of an object to depend on its owned objects. Barnett *et al.* [2] use ownership in the Boogie methodology, which is used to prove C# programs using the Spec# language. Dietl and Müller use ownership in JML, using the Universe type system [6]. Boulmé and Potet use ownership to interpret invariant composition in the B method [4].

* This work is partly supported by the ARC INRIA "CeProMi" (`http://www.lri.fr/ceprom i/`) and by the "CAT" grant ANR-05-RNTL-00302

Ownership and invariant systems have global properties such as: "every object built by the program verifies its invariants", that are important to prove a program specification. It is tempting to provide these properties as axioms to the user, but the predicate "built by the program" is hard to express in the logic. The VCG cannot just enumerate objects in memory and add their invariants in hypotheses, as the number of these objects is unknown, and would be too big anyway.

In the Why platform [8], functions are specified using pre-conditions and post-conditions, as in Hoare logic [9]. The Why VCG, based on Dijkstra's weakest pre-condition calculus [7], computes proof obligations from these specifications. The Why platform contains an intermediate language called Jessie, whose memory model features pointer arithmetic and memory separation using the "component-as-array" model of Burstall and Bornat [3]. This model causes problems to implement an ownership system:

- because of pointer arithmetic, a field of a structure can represent several ownable objects;
- because of memory separation, the global invariants of the ownership system are harder to express.

Memory separation allows to reason about pointer modification without having to worry about pointer aliases, as memory is syntactically split into several *regions*. In particular, this simplifies modular reasoning about programs. Other works tackle this problem, such as separation logic [14] or systems where the user himself may define region variables as location sets [1]. Dynamic frames [11, 15], in particular, allow these variables to be modified during the execution. These systems allow finer memory separation than in Jessie, at the cost of verbosity.

This paper proposes a small formal language with ownership and invariants, compatible with the memory model of Jessie, *i.e.*, with pointer arithmetic and memory separation. We also show how to use invariants when proving the proof obligations of a program. This proposal generalizes the work of Barnett *et al.* in Spec# and was implemented in Jessie, allowing to test it on C and Java programs.

2 Invariants and Ownership

In this section, we give an intuitive description of our ownership system, which is mostly the same than the one used in Spec# [2].

In this paper, the term "object" does not necessarily refer to an instance of a class. In fact, an object is any reference (or pointer) to any data structure.

Objects Are Boxes If a box $\square$ is inside a box $\triangle$, then $\square$ is *owned* by $\triangle$. If $\triangle$ is itself inside another box $\diamond$, then $\square$ is also owned by $\diamond$, because $\square$ is also inside $\diamond$. This defines the *ownership* relation on objects.

The ownership relation is the transitive closure of the *direct ownership* relation. $\square$ is *directly owned* by $\triangle$ if $\square$ is owned by $\triangle$, and if for all boxes $\diamond$ owning $\square$, $\diamond = \triangle$ or $\diamond$ owns $\triangle$. In other words, there is no box between $\square$ and its direct owner.

The direct owner is unique. It doesn't have to exist, though: a box which is not inside any other box is not owned, nor directly owned.

Boxes Can Be Open or Closed To modify the content of a box, it must be *open*. And to open a box, it must be outside any other box, *i.e.*, it must not be owned. This means that to modify an owned object, one must open its owner first. Of course, this owner might be in another bigger box that we might need to open before.

Before closing a box, we must check that all the boxes it contains are already closed. In other words, an object can only own – directly or not – closed objects, and an open object cannot be owned.

Invariants on Closed Boxes The problem with an invariant is that it can be broken. Let's say that you have an integer x, and an invariant saying that $x \neq 0$. Nothing prevents the programmer from assigning 0 to x.

Except if the structure is closed. If we always check its invariant before closing x, we know that $x \neq 0$ always holds when x is closed, because x cannot have been modified since its invariant was checked. By transitivity, if a box is closed, every object it contains also verifies its invariants.

Invariants on Multiple Objects If an invariant depends on several objects, we have to check it each time one of these objects is closed, and we can only assume it if all the objects are closed. This constraint is too heavy.

In the ownership system of Spec#, invariants are associated to objects. The invariants of $\square$ can only depend on the objects that $\square$ owns. Thus, it is sufficient to check the invariant of $\square$ only when closing $\square$, because the other boxes on which the invariant depends are already closed, as they are owned by $\square$.

3 Core Language

In this section we define a small core language featuring pointers on values. These values are our boxes: they contain an invariant and may be open or closed. We show how to express and prove the soundness of the ownership system.

Due to space limitation, we omitted typing rules and the soundness proof of the ownership system. They can be found in the extended version of this article at `http://romain.bardou.fr/papers/jcownlong.pdf`.

Syntax The syntax of our core language is defined in Fig. 1. Values are expressions which cannot be reduced. They may be constants or pointers. Pointers are not directly used when programming; they are the result of allocation. They are annotated with their types. The language has basic expressions: let-binding, sequence, while loops and if-then-else tests.

A pointer p is allocated using **new** $\langle v; I; \boldsymbol{r} \rangle$. The value v can be accessed by dereferencing using $!p$ or modified using $p := e$. Pointers are the boxes of the ownership system: they have an invariant I which is given at allocation. I may depend on the contents of p and a set of pointers $\boldsymbol{r}$, which are also given at allocation. We assume given a syntax for sets of pointers, such as in `assignable` clauses of JML. These are the *reps* pointers of p (standing for "representation" pointers). Finally, pointers can be closed or

Expressions:

$e ::= v$		Values
	x	Variables
	let $x = e$ **in** e	Binding
	$e; e$	Seq
	while e **do** e	Turing-completion
	if e **then** e **else** e	Test
	new $\langle e; I; \boldsymbol{r} \rangle$	Allocation
	$!e$	Dereferencing
	$e := e$	Assignment
	pack e	Closing boxes
	unpack e	Opening boxes

Values:

$v ::= c$		Constants
	$p : \langle \tau \rangle$	Pointers

Types:

$\tau ::=$ unit $\mid$ bool $\mid \cdots$		Base types
	$\langle \tau \rangle$	Pointers

Environments:

$\Delta ::= \epsilon \mid \Delta, x : \tau$	Typing
$\Gamma ::= \epsilon \mid \Gamma, p = \langle e; I; \boldsymbol{r} \rangle^{\square}$	Memories
$\square ::= \circ \mid \times \mid \otimes$	Box state

Fig. 1. Core language syntax

opened using **pack** or **unpack**. As in Spec#, ownership transfer on a pointer p can be achieved to change the ownership hierarchy by unpacking the owner of p and packing another owner which has p as a rep.

We do not specify the logic used to write invariants. We only need to be able to know if an invariant holds, given the state of the memory. We assume that modifying a pointer which is not a rep of a pointer p cannot break the invariant of p.

As an example, the following expression E has one free variable: x which is an integer pointer. It returns a new pointer which has an invariant: it is greater than x. The invariant is a function which takes its future associated pointer as an argument (here p).

$$E = \textbf{let } y = \textbf{new } \langle 0; \lambda p.\ (!p > !x); x \rangle \textbf{ in } y := !x + 1; \textbf{pack } y; y$$

Note that the invariant does not necessarily hold initially, as x might be stricly greater than 0. This is allowed because the pointer is initially open.

This example also shows that in our core language, invariants may be associated with any pointer and may depend on any pointers. This is more general than the ownership system of Spec# where the reps of an object are defined in its type and are restricted to its fields.

Semantics We define a small-step semantics for our core language. $\rightarrow$ is a relation on states, and states are couples of a memory and an expression. The following:

$$\Gamma_1; e_1 \rightarrow \Gamma_2; e_2$$

should be read: "e_1 in memory Γ_1 reduces to e_2 in memory Γ_2".

The $\rightarrow$ relation is the smallest fixpoint of the rules given in Fig. 2, plus some context rules which define the evaluation order (for example, the test cannot be reduced if its condition has not been reduced already). We suppose that the substitution used when reducing let-bindings does not capture variables.

Rules for let-binding (1), sequence (2), loop (3) and test (4, 5) are easy to read. The interesting rules are the ones for allocation (6), pointer dereferencing (7) or assignment (8), packing (9) and unpacking (10), as they can read and write in memory. The

$$\Gamma; \textbf{let } x = v \textbf{ in } e \to \Gamma; e[v/x] \tag{1}$$

$$\Gamma; (\text{unit}; e) \to \Gamma; e \tag{2}$$

$$\Gamma; \textbf{while } e_1 \textbf{ do } e_2 \to \Gamma; \textbf{if } e_1 \textbf{ then } (e_2; \textbf{while } e_1 \textbf{ do } e_2) \textbf{ else } \text{unit} \tag{3}$$

$$\Gamma; \textbf{if } \text{true} \textbf{ then } e_1 \textbf{ else } e_2 \to \Gamma; e_1 \tag{4}$$

$$\Gamma; \textbf{if } \text{false} \textbf{ then } e_1 \textbf{ else } e_2 \to \Gamma; e_2 \tag{5}$$

$$\Gamma; \textbf{new } R \to \Gamma, p = R^\circ; p \ \text{ (where } p \text{ is fresh in } \Gamma) \tag{6}$$

$$\Gamma, p = \langle v; I; \boldsymbol{r} \rangle^\square; !p \to \Gamma, p = \langle v; I; \boldsymbol{r} \rangle^\square; v \tag{7}$$

$$\Gamma, p = \langle v_1; I; \boldsymbol{r} \rangle^\circ; p := v_2 \to \Gamma, p = \langle v_2; I; \boldsymbol{r} \rangle^\circ; \text{unit} \tag{8}$$

$$\Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\circ \\ p_1 = R_1^\times, \cdots, p_n = R_n^\times \end{pmatrix}; \textbf{pack } p \to \Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\times \\ p_1 = R_1^\otimes, \cdots, p_n = R_n^\otimes \end{pmatrix}; \text{unit} \quad (\text{If } I(\Gamma) \text{ holds}) \tag{9}$$

$$\Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\times \\ p_1 = R_1^\otimes, \cdots, p_n = R_n^\otimes \end{pmatrix}; \textbf{unpack } p \to \Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\circ \\ p_1 = R_1^\times, \cdots, p_n = R_n^\times \end{pmatrix}; \text{unit} \tag{10}$$

We omitted rules concerning evaluation order.

Fig. 2. Core language semantics

memory is a set of allocated pointers with their value, their invariant and their reps (the pointers their invariant may depend on). Syntax for memories Γ is given in Fig. 1; they are maps from pointers to their values, invariant and reps. Each pointer can be open ($\circ$), closed ($\times$), or owned ($\otimes$); this is also stored in memory Γ.

To allocate a new pointer, one first needs a fresh pointer p. This pointer is entered in memory with its state R which contains its value, its invariant and the pointers it depends on. Initially, the pointer is open.

Pointer access and modification are just reading or modifying the value associated to the pointer in memory. Access can be done whatever the state of the pointer is, but assignment can only be done on open pointers.

Closing a pointer p with **pack** does not only change the state of p from $\circ$ (open) to $\times$ (closed); it also changes the state of its reps from $\times$ to $\otimes$ (owned). This prevents them from being opened and modified. Opening a pointer with **unpack** is similar, except that the pointer state goes from $\times$ to $\circ$ and the state of its reps goes from $\otimes$ to $\times$. Packing can only be done if the invariant holds.

Our example expression E reduces as follows, in a memory where pointer x has value 42 (omitting some trivial reductions):

$$\begin{aligned}
x = \langle 42; \text{true}; \emptyset \rangle^\times & \quad \textbf{let } y = \textbf{new } \langle 0; \lambda p.\ !p > !x; x \rangle \textbf{ in } \cdots \\
x = \langle 42; \text{true}; \emptyset \rangle^\times, p = \langle 0; !p > !x; x \rangle^\circ & \quad \textbf{let } y = p \textbf{ in } y := !x + 1; \textbf{pack } y; y \\
x = \langle 42; \text{true}; \emptyset \rangle^\times, p = \langle 0; !p > !x; x \rangle^\circ & \quad p := !x + 1; \textbf{pack } p; p \\
x = \langle 42; \text{true}; \emptyset \rangle^\times, p = \langle 43; !p > !x; x \rangle^\circ & \quad \textbf{pack } p; p \\
x = \langle 42; \text{true}; \emptyset \rangle^\otimes, p = \langle 43; !p > !x; x \rangle^\times & \quad p
\end{aligned}$$

Safety A memory is consistent if all closed pointers verify their invariants, all reps pointers of all closed pointers are owned, and all owned pointers have a unique owner.

Definition 1 (Memory Consistency and Pointer State). *A memory Γ is* valid*, written $Valid(\Gamma)$, when:*

- *for all $\langle v; I; p_1 \cdots p_n \rangle^{\square}$ in Γ where $\square \in \{\times, \otimes\}$, I holds and for all i, the state of p_i in Γ is $\otimes$;*
- *for all $p = R^{\otimes}$ in Γ, there is a unique $\langle v; I; \boldsymbol{r} \rangle^{\square}$ in Γ such that $\square \in \{\times, \otimes\}$ and $p \in \boldsymbol{r}$.*

The safety of the ownership system is given by the following property: the memory stays consistent when executing the program.

Theorem 1 (Ownership Safety). *If $Valid(\Gamma_1)$ and $\Gamma_1; e_1 \rightarrow^{\star} \Gamma_2; e_2$ then $Valid(\Gamma_2)$*

The proof is basically the same as the one of Barnett *et al.* [2] but adapted to our simpler core language.

4 Pointer Arithmetic

This section shows how to extend our core language with pointer arithmetic. This can also be used to model arrays.

Pointer Shifting and Difference We assume an operation $\oplus$, called *shift*, which takes a pointer and an integer *offset* and returns a pointer. This operation should verify the following properties:

$$p \oplus i = p \iff i = 0$$
$$(p \oplus i) \oplus j = p \oplus (i + j)$$

We also assume an operation $\ominus$ which takes two pointers and return an integer offset. This operation should verify the following property:

$$p' \ominus p = i \iff p' = p \oplus i$$

Note that the axiomatisation of $\oplus$ does not necessarily mean that all pointers are related through an offset shift. This means that $\ominus$ does not have to be defined for all pairs of pointers. For instance, $\ominus$ allows to model pointer difference in C programs. The ANSI semantics of C specifies that the result is undetermined if the two pointers are not in the same block.[1] This property will be useful when defining arrays (see Sect. 4).

The syntax of expressions is extended to handle the $\oplus$ and $\ominus$ operations. The semantics of the syntax constructions $\oplus$ and $\ominus$ is simply to reduce into the value returned by the respective operations, without changing the memory.

[1] In the Why platform, pointer difference generates a proof obligation requesting that the two pointers are in the same block.

Allocation Pointer shifting can be used to build new pointers, but nothing ensures that these pointers have been allocated. If they are not in memory, the access reduction rule cannot be applied.

We extend allocation by adding an expression n of type int in brackets. This integer is the size of the allocated block. The invariant is now parameterized by the offset of the pointer. The semantics of **new** is defined by the following reduction rule:

$$\Gamma; \textbf{new}\ R[n] \rightarrow \Gamma, p = R(0), p \oplus 1 = R(1), \cdots, p \oplus n - 1 = R(n-1); p$$

where R denotes $\lambda o.\ \langle v; I(o); \boldsymbol{r}(o)\rangle$ and p is a fresh pointer such that $p\oplus 1 \cdots p\oplus(n-1)$ are also fresh in Γ.

This extended **new** can be used to allocate several pointers at the same time. These pointers are all accessible by shifting the pointer returned by the allocation, and their invariants can be different.

Arrays The pointer arithmetic extension can be used to build arrays. For example, the following expression allocates a new array of positive integers:

$$\textbf{new}\ \lambda o.\ \langle 0; \lambda p.\ !p \geq 0; \emptyset\rangle[n]$$

However, the invariant of this array is split into each cell, which is handled separately. A better solution would be to have one single invariant for the whole array:

$$\begin{array}{l}\textbf{let}\ p = \textbf{new}\ \langle 0; \text{true}; \emptyset\rangle[n]\ \textbf{in}\\ \textbf{new}\ \langle p; (\lambda p'.\ !p' \geq 0); p \oplus [0..(n-1)]\rangle\end{array}$$

Note that the size of the rep pointer set depends on n. Another solution is to use the set constructor $p[\star]$ meaning: "all valid shifts of p".

This supposes that the cells of an array are exactly the valid shifts of its first cell. This is possible in our extension thanks to the unconstrained operator $\oplus$ which doesn't have to link all pointers together; otherwise there could be only one array in the whole program.

For pratical purposes, it is also handy to add packing and unpacking operations on blocks $p[\star]$ of pointers. Without them, the user has to write a loop everytime an array is opened or closed.

5 Using Invariants in Proofs

We assume that we can statically determine the invariant of a pointer. This strong restriction can be obtained by typing: the user defines a finite set of invariants and the invariant of a pointer is added to its type. In Spec# or in Jessie, all objects of the same class or structure have the same invariant.

Problem Let's assume that example E in section Sect. 3 has $!p > !x$ as a post-condition P. The generated proof obligation looks like (universal quantifications are omitted):

$$\begin{array}{rcl} \Gamma_1^x = (x = \langle 42; \text{true}; \emptyset \rangle) & \Rightarrow & S_1^x = (x = \times) \Rightarrow \\ \Gamma_1^p = (p = \langle 0; !p > !x; x \rangle) & \Rightarrow & S_1^p = (p = \circ) \Rightarrow \\ \Gamma_2^p = store(\Gamma_1^p, p, 43) & \Rightarrow & S_2^x = store(S_1^x, x, \otimes) \Rightarrow \\ S_2^p = store(S_1^p, p, \times) & \Rightarrow & select(\Gamma_2^p, p) > select(\Gamma_1^x, x) \end{array}$$

where $store$ and $select$ are logic functions to, respectively, change and read the value of a pointer in a memory map. For example, $!p$ becomes $select(\Gamma, p)$ in the logic, where Γ is the current memory. Note how memory separation such as the "component-as-array" model [3] or even finer separation using regions [10] allow to split Γ into several maps. Here, we separated pointers x and p. We also separated values and states of pointers using two maps: Γ and S respectively.

One way of proving P is to read the values of x and p using the hypotheses about Γ_1^x and Γ_2^p. In this example, this is trivial; but usually it is not so simple. A much easier solution would be to use the last hypothesis (p is closed) to apply the invariant of p. To do so we need to apply $Valid$. But one cannot just add an axiom such as:

$$\forall \Gamma,\ Valid(\Gamma) \tag{11}$$

This axiom is inconsistent; the quantification on Γ should be restricted to memories that are actually produced by the program. In Spec# this is done using a predicate called $IsHeap$. The VCG adds instances of this predicate in the proof-obligation hypotheses and (11) becomes:

$$\forall \Gamma,\ IsHeap(\Gamma) \Rightarrow Valid(\Gamma) \tag{12}$$

With memory separation, we cannot instanciate $IsHeap$ on all memory parts, otherwise one could prove inconsistent instances of $Valid$ such as $Valid(\Gamma_1^x, \Gamma_1^p, S_2^x, S_2^p)$ which implies $0 > 42$.

Solution In Jessie, we choose to bypass the use of $IsHeap$: the $Valid$ predicate is instantiated everytime the user might need it, and this instantiation is added as an assumption in the hypotheses of the obligations the user has to prove. We add the following hypotheses to the proof obligation for example E:

$$\begin{array}{cc} Valid(\Gamma_1^x, S_1^x) & Valid(\Gamma_1^x, S_1^x, \Gamma_1^p, S_1^p) \\ Valid(\Gamma_1^x, S_1^x, \Gamma_2^p, S_1^p) & Valid(\Gamma_1^x, S_2^x, \Gamma_2^p, S_2^p) \end{array}$$

In theory, the user might need the $Valid$ predicate to be instantiated everytime the memory is modified, but this would pollute proof obligations with too many hypotheses. In practice, we only instantiate $Valid$ at the beginning of function bodies and loops, and when the memory is modified. It is instanciated on the needed memory parts only.

Another possibility is to instantiate $Valid$ only at the beginning of functions; the user can then deduce the other instances of $Valid$. However, proving $Valid$ can sometimes be quite difficult. Another drawback would be that we would lose some separation properties. Thanks to memory separation, memory can be split into (Γ, S) where

Γ contains pointer values, and S contains pointer states ($\circ$, $\times$ or $\otimes$). For example:

$$\textbf{let } x = \textbf{new } \langle 1; \lambda p.\ !p > 0; \emptyset \rangle \textbf{ in } e$$

A theorem prover can easily deduce that $!x > 0$ if it knows that e returns x packed, but only if Valid has been added as an assumption after e.

Well-foundedness We are using Valid to prove some proof obligations, but Valid only holds if the proof obligations have been proven. In this section, we show that this is well-founded.

The proof obligations ensure, among other things, that an expression e_1 which is not a value reduces without errors:

$$\text{If } \mathit{Valid}(\Gamma_1) \text{ then there exist } \Gamma_2, e_2 \text{ such that } \Gamma_1; e_1 \rightarrow \Gamma_2; e_2 \tag{13}$$

In particular, this means that invariants hold before packing, assigned pointers are open, and so on.

We apply Theorem 1 (which does not depend on proof obligations) to extend (13):

$$\text{If } \mathit{Valid}(\Gamma_1) \text{ then there exist } \Gamma_2, e_2 \text{ such that } \Gamma_1; e_1 \rightarrow \Gamma_2; e_2 \text{ and } \mathit{Valid}(\Gamma_2) \tag{14}$$

By applying (14) inductively, we show our final theorem:

Theorem 2. *All instances of Valid introduced as assumptions are correct.*

6 Example

This example is inspired by some Java code due to Müller [13]. We suppose an integer array t of size $n+1$. The following Java code counts the number of positive integers in the array, and then copies them into a new array u:

```
int i, j, count = 0;
for (i=0; i < t.length; i++)
  if (t[i] > 0) count++;
int u[] = new int[count];
for (i=0, j=0; i < t.length; i++)
  if (t[i] > 0) u[j++] = t[i];
```

We can encode this in our core language:

let $i =$ **new** $\langle 0; \text{true}; \emptyset \rangle$ **in**
let $j =$ **new** $\langle 0; \text{true}; \emptyset \rangle$ **in**
let $\mathit{count} =$ **new** $\langle 0; (\lambda p.\ !p = \mathit{Card}\{k \mid 0 \le k \le n\ \wedge\ !(t \oplus k) > 0\}); t \oplus [0..n] \rangle$ **in**
while $!i \le n$ **do**
  (**if** $!(t \oplus !i) > 0$ **then** $\mathit{count} := !\mathit{count} + 1$;
  $i := !i + 1$);
pack count;
let $u =$ **new** $\langle 0; \text{true}; \emptyset \rangle [!\mathit{count}]$ **in**
$i := 0$;
while $!i \le n$ **do**
  (**if** $!(t \oplus !i) > 0$ **then** $(u \oplus !j := !(t \oplus !i);\ \ j := !j + 1)$;
  $i := !i + 1$)

Pointer $count$ has an invariant saying that its content is the number of strictly positive integers in t. We prove it when packing $count$ using a loop invariant on the first loop.

This illustrates several features of our core language: invariants on any pointer and not just objects, pointer arithmetic, and memory separation support. Memory separation is key to prove the safety of the accesses to u in the second loop. Indeed, $j < count$ is a loop invariant; but to show it we need to know that t is not modified by updates to u. Memory separation gives this for free by separating t and u [10].

To show that $j < count$ we also need the invariant of $count$, given by instanciating the following predicate:

$$\begin{array}{l} \mathit{Valid}(\Gamma^t, \Gamma^{count}, S^t, S^{count}) \equiv \\ \quad \forall count,\ \mathit{select}(S^{count}, count) \in \{\times, \otimes\}\ \Rightarrow \\ \quad \mathit{select}(\Gamma^{count}, count) = \mathit{Card}\{k \mid 0 \leq k \leq n\ \wedge\ \mathit{select}(\Gamma^t, (t \oplus k)) > 0\} \end{array}$$

This predicate is instanciated using the memory parts corresponding to the last time $count$ was modified, *i.e.*, when it was packed. We only show the part of the predicate needed to deduce the invariant of $count$.

7 Conclusion

We introduced a small language with pointers, an ownership system and invariants. It was shown to be safe, and then extended with pointer arithmetic. As far as we know, this work is the first attempt to formalize an ownership system with pointer arithmetic, and thus applicable to the C language, although the VCC tool [16] implements a solution. This extension offers multiple ways to specify array invariants: they can be split in each cell, or a pointer on the array can be allocated with a global invariant for the array.

Our core language could be extended with several other features. In particular, pointers could contain extensible records, which would model objects. As for pointer arithmetic, this is mostly independent from the ownership system itself. Our language is an attempt to generalize the Spec# methodology: invariants are not limited to object fields and may depend on any pointer. This simplifies formal reasoning, and can be used in languages without objects, or with invariants which are not necessarily defined for a whole class.

We also showed how the safety properties of the ownership system can be instantiated to be used when proving the proof obligations of the program. This was already done with simple memory models, although to the best of our knowledge, formalizing this method and proving its soundness is new. Moreover, our solution can be used when the memory model features memory separation, as in the Jessie language in which is was implemented.

Another lead for research could be the separation properties of the ownership system shown in Sect 5. Invariant properties on the whole memory can be deduced from a small part of the memory.

Acknowledgements Thanks to Claude Marché, Yannick Moy, to all the members of the ProVal team, and to all anonymous reviewers for their invaluable remarks and advice.

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, Paphos, Cyprus, July 2008.
2. 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.
3. R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
4. S. Boulmé and M.-L. Potet. Interpreting invariant composition in the B method using the Spec# ownership relation: a way to explain and relax B restrictions. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2007.
5. 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.
6. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
7. E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.
8. 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*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580 and 583, Oct. 1969.
10. T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, Mar. 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`.
11. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Hamilton, Canada, 2006.
12. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.
13. P. Müller. Specification and verification challenges. Exploratory Workshop: Challenges in Java Program Verification, Nijmegen, The Netherlands, Sept. 2006. `http://www.cs.ru.nl/~woj/esfws06/slides/Peter.pdf`.
14. 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.
15. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering (FASE'08)*, Budapest, Hungary, Apr. 2008.
16. J. S. Wolfram Schulte, Songtao Xia and F. Piessens. A glimpse of a verifying c compiler.