Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions

Ioannis T. Kassios

Dept. of Computer Science, University of Toronto
BA5212, 40 St.George St. Toronto ON M5S 2E4 Canada
ykass@cs.toronto.edu

Abstract. This paper addresses the frame problem for programming theories that support both sharing and encapsulation through specification variables. The concept of dynamic frames is introduced. It is shown how a programming theory with dynamic frames supports both features, without the use of alias control or any other kind of restriction. In contrast, other approaches introduce a number of restrictions to the programs to ensure soundness.

Keywords: framing, object orientation.

1 Introduction

When specifying a piece of computation, we should also specify its *frame*, i.e. the part of the state that it operates upon. Without framing, our specification language is not very useful. For example, suppose that we want to specify that a computation C increments program variable x by 1. In a relational setting like [9], the specification would be

$$x' = x + 1 \tag{1}$$

where the primed identifier x' represents the final value of program variable x and the plain identifier x its initial value. The specification (1) says how C changes x but it says nothing about the effect of C on other program variables. A client which uses more program variables will have trouble using C.

In a non-modular setting like [9], we know all the program variables. We can use this knowledge to add framing requirements to (1). For example, if x, y, z are all the program variables, then the specification becomes

$$x' = x + 1 \ \wedge \ y' = y \ \wedge \ z' = z \tag{2}$$

Modular programming makes it impossible to write such assertions: we do not know all the variables of the program at the time that we specify a computation. In modular programming theories, it is standard to separate the framing specification from the functional specification like that:

$$\textbf{ensures } x' = x + 1 \quad \textbf{modifies } x \tag{3}$$

J. Misra, T. Nipkow, and E. Sekerinski (Eds.): FM 2006, LNCS 4085, pp. 268–283, 2006.

The above specification says that the value of x is increased by 1 and that the computation only modifies the program variable x. Its translation into a relational specification depends on the client. For example, if a client introduces variables y, z then (3) is translated to (2) for that client.

Now, let us add encapsulation to the picture: we want modules to support private program variables. We also want them to support public *specification variables*. Specification variables are abstract representations of the encapsulated state that are visible to the client. Their exact relation with the private program variables is known only to the implementer of the module.

One standard example of using specification variables is the specification and implementation of a module that formalizes sets of integers. The module provides an operation that inserts elements into the set and an operation that queries whether an element is in the set. The specification of the module uses a public specification variable S to represent the value of the set. This is the specification of the module as the client sees it:

module *ASpec*
 spec var $S \subseteq \mathbb{Z}$

 insert$(x \in \mathbb{Z})$ **ensures** $S' = S \cup \{x\}$
 find$(x \in \mathbb{Z})$ **ensures** $S' = S \ \wedge \ \mathit{return}' = (x \in S)$
end module

The client knows nothing about the internal representation of S and how it relates to its private variables.

The implementer's job is to refine the module *ASpec* using concrete program variables and concrete programs. A possibility is to use a private array L to hold all the elements of S. The exact representation of S is given in terms of the private program variable L. The refinement looks like this[1]:

module *AImpl*
 prog var $L \in \mathbb{Z}^*$
 spec var $S = \{x \in \mathbb{Z} \cdot \ \exists i \in \mathbb{N} \cdot \ x = L\ i\}$

 insert$(x \in \mathbb{Z})$ **ensures** $L' = [x]^\frown L$
 find$(x \in \mathbb{Z})$ **ensures** $L' = L \ \wedge \ \mathit{return}' = (\exists i \in \mathbb{N} \cdot \ L\ i = x)$
end module

Framing specifications in this new setting cannot mention the private program variables, which are unknown to the client. They must instead mention public specification variables, like S. For example, the framing specification of the method *insert* should be

modifies S (4)

[1] Of course, this is not yet an implementation, because the operations are not implemented. Further refinements will give an implementation, but this is not the point of this example.

which means that the computation is allowed to change S *and all specification and program variables on which* S *depends*. In our example, this means that the computation changes S and L. As in the case without specification variables, if a client introduces specification or program variables y, z, the specification (4) is translated to:

$$y' = y \ \wedge \ z' = z \tag{5}$$

since S is not known to depend on y, z. Thus the reasoning on the level of specifications expects that a computation that satisfies (4) preserves y, z.

Unfortunately, in the presence of pointers, the translation may be unsound. This is because, the representation of y may actually share heap locations with the representation of S. For example, y could be given by the following representation:

prog var $p \in \mathit{pointerTo}\ (\mathbb{Z}^*)$
$y = (\mathit{contentOf}\ p)0$

and the pointer p might happen to point to the private array L of module *AImpl*. When that happens, changes to L may change the value of y, contrary to what is predicted by the theory. In our example, our implementation of *insert* will change the value of y, unless the parameter x is equal to the initial value of y. This situation is called *abstract aliasing* [18].

To avoid the problem, existing solutions [18, 20, 16] impose a series of programming restrictions, which guarantee absence of abstract aliasing: if two variables are not known to be dependent then they can be assumed independent. Unfortunately, these solutions come at a price. One problem is formal complication: the theories either introduce new formalisms (universes in [20], packing and unpacking in [16] or a big collection of ad hoc rules [18]). Another problem is inflexibility: the restrictions imposed rule out several useful implementation patterns. These patterns have to do with objects that cross encapsulation boundaries and with sharing.

The contribution of this paper is a formal theory that supports specification variables and pointers without any programming restriction. The basic idea is to make the specification language strong enough to express the property "at the present state the values of x and y are independent", i.e. absence of abstract aliasing. Because this property is expressible as a state predicate, it can be asserted and assumed by the user of the theory at any point where it is needed. This means that it is not necessary for the programming theory to ensure that it is always true and thus to impose any restriction whatsoever. Furthermore, our approach is very simple in that it does not introduce any new formal concept: dynamic frames are a special case of specification variables and they are handled in exactly the same way by the user of the theory.

2 Theory of Dynamic Frames

2.1 Notation

Here we introduce some of the notation to be used in the rest of the paper.

Equality. The operator $=\!\!=$ has the same semantics as $=$ but lowest precedence. It is used to reduce the number of parentheses in expressions.

Sets and Set Notation. The set of booleans $\{\top, \bot\}$ is denoted $\mathbb{B}$. Set comprehension is denoted $\{x \in D \cdot\ P\}$ where D is a set and P is a boolean expression with free occurrences of variable x.

If i, j are integers, then the sets $\{i, ..j\}$ and $\{i, .., j\}$ are defined as follows:

$$\{i, ..j\} \ = \ \{x \in \mathbb{Z} \cdot\ i \leq x < j\}$$
$$\{i, .., j\} \ = \ \{x \in \mathbb{Z} \cdot\ i \leq x \leq j\}$$

Functions. Functions are introduced using syntax $\lambda x \in D \cdot B$ where D is the domain and B is the body of the function. Operator Dom extracts the domain of a function. Function application is denoted by juxtaposition. The domain restriction operator $\triangleright$ and the one-point update $\mapsto \mid$ operator are defined by:

$$f \triangleright D \ =\!\!= \ \lambda x \in D \cap \mathrm{Dom}\ f \cdot\ f\ x$$
$$y \mapsto z \mid f \ =\!\!= \ \lambda x \in \{y\} \cup \mathrm{Dom}\ f \cdot\ \textbf{if}\ x = y\ \textbf{then}\ z\ \textbf{else}\ f\ x$$

Lists. A *list* L is a function whose domain is $\{0, ..i\}$ for some natural number i called the *length* of L and denoted $\#L$. We can use syntax $[x; y; ...]$ to construct lists. The concatenation of lists L and M is denoted $L^\frown M$. Notation $L[i; ..j]$ extracts the part of the list between indices i (incl.) and j (excl.). The predicate *disjoint* takes a list of sets L and asserts that the sets in L are mutually disjoint. Formally:

$$\mathit{disjoint}\ L \ =\!\!= \ \forall i \in \{0, ..\#L\} \cdot \forall j \in \{0, ..\#L\} \cdot\ i = j \ \vee\ L\ i \cap L\ j = \emptyset$$

Open Expressions. In this paper, some identifiers *stand for* expressions that may contain free variables. We may say e.g. that E is an *expression* on variables $x, y, ...$ We call such identifiers "open expressions". Although use of open expressions is practiced in some influential formal theories, like for example [10, 1, 9], some people are not comfortable with them. Readers who do not like open expressions, may consider the occurrence of an expression E on variables $x, y, ...$ as a purely syntactical abbreviation of $\overline{E}\ x\ y\ ...$ where $\overline{E}$ is a function.

Let E, t be expressions and x a variable. Then $E(t/x)$ denotes expression E with all free occurrences of x substituted by t.

2.2 Basic Definitions

State and Variables. There is an infinite set of *locations Loc*. Any subset of *Loc* is called a *region*. A *state* σ is a finite mapping from locations to values. A location in Dom σ is *used* or *allocated* in σ. The *set of all states* is denoted Σ.

A *specification variable* is an expression that depends on the state (i.e. with free occurrences of variable $\sigma \in \Sigma$). Two important specification variables are the set of all allocated locations Used and the set of all unallocated locations Unused, defined as follows:

$$\mathrm{Used} = \mathrm{Dom}\ \sigma \qquad\qquad \mathrm{Unused} = \mathit{Loc} \setminus \mathrm{Used}$$

For any specification variable v, the expression v' is defined by:

$$v' = v(\sigma'/\sigma)$$

The expression v' is called the *final value of* v.

A *program variable* x is a special case of specification variable whose value is the content of the state at a constant location $addr\_x$, called the *address* of x:

$$x = \sigma(addr\_x)$$

Imperative Specifications. An *imperative specification* is a boolean expression on the state-valued variables $\sigma \in \Sigma$ and $\sigma' \in \Sigma$. The state σ is called the *pre-state* and the state σ' is called the *post-state*. Programming constructs are defined as imperative specifications. The program *ok* leaves the state unchanged:

$$ok \;=\; \sigma' = \sigma$$

If x is a program variable and E is an expression on σ, then the program $x := E$, called *concrete assignment*, is defined by:

$$x := E \;=\; \sigma' = addr\_x \mapsto E \mid \sigma$$

If l is a location-valued expression on σ and E is an expression on σ, then the program $*l := E$, called *pointer assignment*, is defined by:

$$*l := E \;=\; \sigma' = l \mapsto E \mid \sigma$$

If P and Q are imperative specifications, then the specification $P;Q$, called the *sequential composition* of P and Q is defined by:

$$P;Q \;=\; \exists \sigma'' \cdot P(\sigma''/\sigma') \wedge Q(\sigma''/\sigma)$$

If P is an imperative specification, then the specification **var** $x \cdot P$, called *local program variable introduction*, is defined by:

$$\textbf{var}\;\; x \cdot P \;=\; \exists addr\_x \in \text{Unused} \cdot \; P$$

In P, occurrences of the identifier x are abbreviations of expression $\sigma(addr\_x)$. More programming constructs can be introduced; here we present only those used in this paper.

Modules. A *module* is a collection of name declarations and axioms. We introduce a module using syntax **module** N, where N is the name of the module and we conclude its definition using syntax **end module** . Keywords **spec var** and **prog var** declare specification variables and program variables respectively. Syntax **import** M is used to import all names and axioms of module M into the module in which it appears. A module M *refines* (or *implements*) a module N if its axioms imply the axioms of N.

2.3 Dynamic Frames and Framing Specifications

A *dynamic frame* f is a specification variable whose value is a set of allocated locations, i.e. $f \subseteq \text{Used}$. For any dynamic frame f, we define three new imperative specifications. The *preservation* Ξf is satisfied by every computation that does not touch region f. The *modification* Δf is satisfied by every computation that only touches region f or at most allocates new memory. Finally, the *swinging pivots requirement* Λf does not allow f to increase in any way other than allocation of new memory. The formal definitions are:

$$\Xi f = \sigma' \triangleright f = \sigma \triangleright f \qquad \Delta f = \Xi(\text{Used} \setminus f) \qquad \Lambda f = f' \subseteq f \cup \text{Unused}$$

Let f be a dynamic frame. Let v be a specification variable. The state condition f **frames** v is defined as follows:

$$f \textbf{ frames } v \;=\; \forall \sigma' \cdot \; \Xi\, f \;\Rightarrow\; v' = v$$

In a state σ in which this condition is true, we say that f *frames* v or that f *is a frame for* v. When that happens, v depends only on locations in f, i.e. leaving those locations untouched preserves the value of v. There can be more than one variable to the right of **frames** :

$$f \textbf{ frames } (x, y, ...) \;=\; f \textbf{ frames } x \;\wedge\; f \textbf{ frames } y \;\wedge\; ...$$

Framing properties are usually introduced as axioms in a module. The implementer of the module is then obliged to provide a definition for the specification variables and their frames such that the framing property is always true. For example, in the following definitions, Module *CImpl* refines Module C.

module C	**module** *CImpl*
spec var $x \in \mathbb{Z}$, $f \subseteq \text{Used}$	**prog var** $y \in \mathbb{Z}$, $z \in \mathbb{Z}$
f **frames** x	**spec var** $x = y + z$, $f = \{addr\_y, addr\_z\}$
end module	**end module**

Independence of two variables (absence of abstract aliasing) is expressible as disjointness of dynamic frames. In particular, if f is the frame of x and g is the frame of y and the f, g are disjoint, then the specification variables x and y are independent. If we want to change only variable x, then we frame on f, which guarantees preservation of y (and all other known and unknown specification variables that are independent of x):

$$f \textbf{ frames } x \;\wedge\; g \textbf{ frames } y \;\wedge\; \mathit{disjoint}[f; g] \;\wedge\; \Delta f \;\;\Rightarrow\;\; y' = y$$

Disjointness of frames is an important property and therefore one we want to preserve. To do that, dynamic frames need to be framed too. We usually axiomatize a dynamic frame to frame itself, i.e. f **frames** $(f, x, ...)$. Given self-framing dynamic frames, a way to preserve disjointness is the conjunction of framing on f with the *swinging pivots requirement* on f. Suppose that g is a self-framing frame disjoint from f. Then $\Delta f \;\wedge\; \Lambda f$ preserves the disjointness:

$$\mathit{disjoint}[f; g] \;\wedge\; \Delta f \;\wedge\; \Lambda f \;\;\Rightarrow\;\; \mathit{disjoint}[f'; g'] \tag{6}$$

Intuitively, the reason is that Δf preserves g while Λf ensures that f only grows with previously unallocated locations, i.e. with locations that are not in g. The formal proof is found in [14]. Notice that the implementer of $\Delta f \ \wedge \ \Lambda f$ does not even have to know g, which makes the property (6) very useful for modular reasoning.

The combination of Δ and Λ is very useful. It is a good idea to give it its own notation. Suppose that:

$$f \textbf{ frames } (f, x, y, z, ...)$$

is given as an axiom. Then, we define *abstract assignment* to specification variable x (and similarly for the other specification variables $y, z, ...$) as follows:

$$x := E \ = \ \Delta f \ \wedge \ \Lambda f \ \wedge \ x' = E \ \wedge \ y' = y \ \wedge \ z' = z \ \wedge \ ...$$

2.4 Objects

Basics. The theory of dynamic frames has already been exposed and it is orthogonal to object oriented programming. However, the examples that we use are based on object orientation so we need some formal support for objects. This section is by no means a complete formalization of object orientation.

There is a set $\mathcal{O}$. The elements of $\mathcal{O}$ are called *object references*. The special value *null* denotes the null reference. It is not included in $\mathcal{O}$.

A *specification attribute* is an expression with free occurrences of the identifiers $\sigma \in \Sigma$ and $self \in \mathcal{O}$. A *program attribute* x is a special case of specification attribute such that

$$x = \sigma(addr\_x)$$

for some location $addr\_x$ that depends on *self* but not on σ. The location $addr\_x$ is called the *address* of x. The keyword **spec attr** introduces specification attributes. The keyword **prog attr** introduces program attributes. The definitions for concrete assignment and abstract assignment are valid for program and specification attributes as well

The following abbreviation is introduced to facilitate the access of attributes of object references other than *self*:

$$p.E \ = \ E(p/self)$$

for object reference p and any expression E that depends on *self*. The notation $(.)$ can be generalized to apply many times: (for any $k \in \mathbb{N}$)

$$[E]^0 = self \qquad\qquad [E]^{k+1} = [E]^k.E$$

We use three specification attributes, the initialization constraint *init*, the invariant *inv* and the representation region *rep*. These specification attributes obey the following axioms for all object references and states:

$$init \in \mathbb{B} \qquad inv \in \mathbb{B} \qquad init \ \Rightarrow \ inv \qquad inv \ \Rightarrow \ rep \subseteq \text{Used} \qquad (7)$$

For our convenience we specify that the representation region of the null reference is empty:

$$null.rep = \emptyset$$

If o is an object reference, l is an identifier and $x, y, ...$ are values, then $o.l(x; y; ...)$ is an imperative specification called *method invocation* of l on o with parameters $x; y; ....$

Class Specifications. A *class* is a set of object references. The specification of class C is a collection of axioms that begins with **class** C and ends with keyword **end class** . In each axiom, the identifier *self* is implicitly universally quantified over C, and the identifiers σ, σ' are implicitly universally quantified over Σ. Within the specification of C, the identifier *self* represents the *current* object reference. There are usually two kinds of axioms in a class specification: the *attribute specifications* and the *method specifications.*

Attribute Specifications. The attribute specifications axiomatize the specification and program attributes of a class. In a class implementation, the attribute specifications have the form $a = E$, where a is a specification attribute and E is an expression.

Framing properties are attribute specifications. Frequently we assert that the representation region frames itself, the invariant and other specification attributes, i.e.:

$$inv \;\Rightarrow\; rep \;\textbf{frames}\; (rep\;,\; inv\;, ...)$$

There are cases, like *IteratorSpec* of Sect. 3.3, where we do not use such framing.

Method Specifications. Method specifications have the form:

$$\forall x \cdot \forall y \cdot \;\; ... \; self.l(x; y; ...) \quad \Rightarrow \quad S \tag{8}$$

where l is an identifier, $x, y, ...$ are data-valued identifiers and S is an imperative specification, called the *body of method* l. The expression (8) is abbreviated by

$$\textbf{method}\; l(x; y; ...) \cdot S$$

In a class implementation, S must be a program.

Object Creation. To create a new object of class C, we allocate fresh memory for its representation region and we ensure that its initialization condition is met. This is all done by the specification $x :=$ **new** C defined as follows:

$$\begin{aligned} & x := \textbf{new}\;\; C \\ = \; & \Delta\{addr\_x\} \;\wedge\; x' \in C \;\wedge\; (x.init)' \;\wedge\; (x.rep)' \subseteq \text{Unused} \setminus \{addr\_x\} \end{aligned}$$

where x is a program variable.

3 Examples

In this section, we present some examples of specification and implementation in our theory. Proofs of correctness are omitted for lack of space; the reader is instead referred to [13] and [14]. Also, for the sake of brevity, we have omitted all queries from the class specifications, because they add nothing to the examples.

3.1 Lists

This example concerns the specification and implementation of a class *List* that formalizes lists of integers. The specification comes in a module named *ListSpec*. It introduces the class *List* and a specification attribute L whose value is the represented list. The frame *rep* frames itself, the invariant and L. The initial value of L is the empty list.

module *ListSpec*
class *List*
spec attr L
$inv \;\Rightarrow\; L \in \mathbb{Z}^* \;\wedge\; rep$ **frames** (rep, inv, L)
$init \;\Rightarrow\; L = []$

The method *insert* inserts an item at the beginning of the list:

method $insert(x) \cdot \; inv \;\wedge\; x \in \mathbb{Z} \;\Rightarrow\; (L := [x]^\frown L)$

The method *cut* takes two parameters, an address l and an integer *pos*. It breaks the list in two (at the point where *pos* is pointing). The first part of the old list is returned as a result (the address l serves as returning address). The second part is the new value of the current list. The specification of *cut* allows this method to be implemented by pointer operations: in particular, it allows the representation region of the returned list to contain memory that used to belong to the representation region of *self*. The final representation regions of the two lists are disjoint:

method $cut(l; pos)\cdot$
$inv \;\wedge\; l \in Loc \setminus rep \;\wedge\; pos \in \{0, .., \#L\}$
$\Rightarrow \quad \Delta(\{l\} \cup rep) \;\wedge\; L' = L[pos; ..\#L] \;\wedge\; inv' \;\wedge\; \Lambda rep$
$\wedge\, \sigma' l \in List \;\wedge\; (\sigma l.L)' = L[0; ..pos] \;\wedge\; (\sigma l.inv)'$
$\wedge\, (\sigma l.rep)' \subseteq rep \cup \mathrm{Unused} \;\wedge\; disjoint[rep \;;\; \sigma l.rep \;;\; \{l\}]$

Finally, the method *paste* concatenates a list to the beginning of the current list. The initial representation regions of the two lists must be disjoint. The specification says that the representation region of the parameter may be "swallowed" by the representation region of the current list object. This allows implementation with pointer operations:

method *paste*(*p*)·
$inv \ \wedge \ p \in List \ \wedge \ p.inv \ \wedge \ rep \cap p.rep = \emptyset$
$\Rightarrow \quad \Delta(rep \cup p.rep) \ \wedge \ L' = p.L \frown L \ \wedge \ inv'$
$\wedge \ rep' \subseteq rep \cup \text{Unused} \cup p.rep$
end class
end module

To implement *ListSpec*, we define a new module *ListImpl*. We use a standard linked list implementation. The nodes are object references with program attributes *val* and *next*, where *val* stores a list item and *next* refers to the next node in list (or is equal to *null* if there is no next node). The list object has a reference *head* to the first node.

module *ListImpl*
class *Node*
prog attr *val* , *next*
$init \ = \ next = null \ \wedge \ val \in \mathbb{Z}$
$rep = \{addr\_val, addr\_next\}$
end class

class *List*
prog attr *head*

The specification attributes and the methods for linked lists are implemented as follows:

spec attr $len \ = \ min\{i \in \mathbb{N} \cdot \ head.[next]^i = null\}$
spec attr $L = \lambda i \in \{0, ..len\} \cdot head.[next]^i.val$
$rep \ = \ \{addr\_head\} \cup \bigcup i \in \{0, ..len\} \cdot head.[next]^i.rep$
$inv \ = \ (\forall i \in \{0, ..len\} \cdot head.[next]^i.val \in \mathbb{Z})$
$\wedge \ disjoint(\ [\{addr\_head\}]$
$\frown \ \lambda i \in \{0, ..len\} \cdot head.[next]^i.rep \ )$
$init \ = \ head = null$

method *insert*(*x*) · **var** *n*·
$n :=$ **new** *Node* ; $n.val := x$; $n.next := head$; $head := n$
method *cut*(*l*; *pos*)·
$*l :=$ **new** *List*
; **if** $pos = 0$ **then** *ok*
else (**var** $q \cdot \ \sigma l.head := head$; $q := head.[next]^{pos-1}$
; $head := q.next$; $q.next := null$)
method *paste*(*p*)·
if $p.head = null$ **then** *ok*
else (**var** *q*·
$q := p.head.[next]^{p.len-1}$; $q.next := head$; $head := p.head$)
end class
end module

3.2 Sets

This example presents the specification *SetSpec* of a class *Set* that formalizes sets of integers. The class supports an insertion method *insert* and a method *paste* that performs the union of the current set to its parameter. Like its *List* counterpart, the method *paste* allows the current set object to "swallow" part of the representation region of the parameter:

module *SetSpec*
 class *Set*
  spec attr S
  $inv \;\Rightarrow\; S \subseteq \mathbb{Z} \;\wedge\; rep$ **frames** (S, rep, inv)
  $init \;\Rightarrow\; S = \emptyset$

  method $insert(x) \cdot \; inv \;\wedge\; x \in \mathbb{Z} \;\Rightarrow\; (S := S \cup \{x\})$
  method $paste(p) \cdot$
   $inv \;\wedge\; p \in Set \;\wedge\; p.inv \;\wedge\; rep \cap p.rep = \emptyset$
  $\Rightarrow$ $\Delta(rep \cup p.rep) \;\wedge\; S' = p.S \cup S \;\wedge\; inv'$
   $\wedge\; rep' \;\subseteq\; rep \cup \text{Unused} \cup p.rep$
 end class
end module

We can implement the class by using an internal list object:

module *SetImpl*
 import *ListSpec*

 class *Set*
  spec attr S
  prog attr *contents*
  $inv \;=\; contents \in List \;\wedge\; contents.inv$
   $\wedge\; addr\_contents \notin contents.rep$
  $init \;=\; inv \;\wedge\; contents.init$
  $rep \;=\; \{addr\_contents\} \cup contents.rep$
  $S = \{x \in \mathbb{Z} \cdot \exists i \in \{0, ..\#(contents.L)\} \cdot contents.L\ i = x\}$

  method $insert(x) \cdot contents.insert(x)$
  method $paste(p) \cdot contents.paste(p.contents)$
 end class
end module

3.3 Iterators

This example shows how the theory handles sharing and friend classes. We specify iterators in a module *IteratorSpec* which imports the *ListSpec* module. An iterator has a list attached to it, given by the value of the program attribute *attl*. It also points to an item in the list, or perhaps to the end of the list. The index

of the pointed item is given by the value of the specification attribute *pos*. The representation region of an iterator is disjoint from that of the attached list.

module *IteratorSpec*
import *ListSpec*

class *Iterator*
prog attr *attl*
spec attr *pos*

$$
\begin{array}{l}
\quad inv \\
\Rightarrow \quad (attl = null \ \vee \ (attl \in List \ \wedge \ attl.inv)) \\
\quad \wedge \ disjoint[rep; attl.rep] \ \wedge \ rep \ \textbf{frames} \ (attl, rep) \\
\quad \wedge \ (rep \cup attl.rep) \ \textbf{frames} \ inv \\
\\
\quad inv \ \wedge \ attl \neq null \\
\Rightarrow pos \in \{0, .., attl.(\#L)\} \ \wedge \ (rep \cup attl.rep) \ \textbf{frames} \ pos \\
\\
init \quad \Rightarrow \quad attl = null
\end{array}
$$

The class of iterators supports methods for attachment and traversal:

method $attach(l)\cdot$
$inv \ \wedge \ l \in List \ \wedge \ l.inv$
$\Rightarrow \Delta rep \ \wedge \ inv' \ \wedge \ pos' = 0 \ \wedge \ attl' = l \ \wedge \ \Lambda rep$
method $next()\cdot$
$inv \ \wedge \ pos < attl.(\#L)$
$\Rightarrow \Delta rep \ \wedge \ inv' \ \wedge \ pos' = pos + 1 \ \wedge \ attl' = attl \ \wedge \ \Lambda rep$
end class
end module

The implementation of iterators imports *ListImpl*. This means that the implementer of the *Iterator* class has access to the implementation of the *List* class. This makes *Iterator* a *friend* of *List*. Compare that to the implementation of the *Set* class which imports *ListSpec* and therefore does not have access to the implementation of *List*: the class *Set* is not a friend of *List*. Iterators are implemented as pointers to list nodes:

module *IteratorImpl*
import *ListImpl*

class *Iterator*
prog attr *attl* , *currentNode*
spec attr *pos*

$$
\begin{array}{lll}
inv & = & (attl = null \ \vee \ (attl \in List \ \wedge \ attl.inv)) \\
& & \wedge \ (attl \neq null \ \Rightarrow \ pos \in \{0, .., attl.(\#L)\}) \ \wedge \ rep \subseteq \text{Used} \\
& & \wedge \ disjoint[rep \ ; \ attl.rep] \\
pos & = & min\{i \in \mathbb{N} \cdot \ attl.head.[next]^i = currentNode\}
\end{array}
$$

$rep = \{addr\_attl\ ,\ addr\_currentNode\}$
$init\ \ =\ \ attl = currentNode = null$

method $attach(l) \cdot\ \ attl := l$ **;** $currentNode := l.head$
method $next() \cdot\ \ currentNode := currentNode.next$
end class
end module

4 Discussion

The theory of Dynamic Frames is part of the more general theory of object oriented refinement that appears in [14]. It is an application of the design principles of *decoupling* and *unification* as advertised in [11, 12]: it decouples the feature of alias control from other formal constructs, like the class, the module or even the object and it unifies frame specifications with functional specifications.

The two important merits of the theory are simplicity and generality. It is formally simple because it solves the problem without introducing any new concept, formalism or axiomatization (dynamic frames are a special case of specification variables). It is general because, unlike competing theories, to guarantee its soundness we do not need to enforce any programming restrictions.

One objection to the theory of Dynamic Frames is that specifications in it may become too verbose. This is always a danger when designing a more flexible system: the extra generality provides more options to the user; thus more things to say. However, there are good ways to deal with this problem. For example, common specification cases may be given their own notation and reasoning laws. Such specifications include the swinging pivots requirement and the abstract assignment. Further notational and reasoning conveniences are found in [14].

4.1 Related Work

Older Approaches. Leino and Nelson's work [18] is a big collection of rules that deal with some of the most frequent cases of the problem. The approach has considerable complexity and it does not address all cases uniformly. Its most drastic restriction is that it forces each method to obey the swinging pivots requirement. This, even in its less restrictive version [5], rules out the implementation for *paste* in Sect. 3.1. In a variant [19], the authors use *data groups* [15] instead of variables in frame specifications. However, absence of abstract aliasing is still not expressible in the specification language and thus the swinging pivots requirement together with other restrictions similar to those in [18] are enforced.

The *Universes* type system [20] is a much simpler and more uniform approach to the problem, also adopted by the JML language [21, 6]. It too imposes restrictions that have to do with objects travelling through encapsulation boundaries. Our implementation for *List* is possible in [20], although somewhat awkwardly, by declaring the node objects "peers" to their containing list object. Our implementation of the *paste* method for *Set* is impossible, because for the peer solution to work, *Set* and *List* should be declared in the same module.

Boogie. A less restrictive variant of Universes is the Boogie methodology [2, 16, 4, 17] used in Spec# [3]. Its most important improvement over Universes is that it allows objects to cross encapsulation boundaries. However, the Boogie methodology has the same visibility restriction concerning "peer" objects as the Universes type system: a class of shareable objects must be aware of all its sharing clients. This causes a modularity problem: the creation of a new sharing client of a class C means that the specification of C must be revised. Moreover, if C happens to be a library class whose specification and implementation cannot be modified, the creation of new sharing clients is not even possible [16].

The Dynamic Frames theory imposes no such restriction and therefore it is more flexible than Boogie. The Iterator example of Sect. 3.3 shows an example of sharing. In this particular example, the class *Iterator* happens to be a friend of the class *List*. This is a coincidence. An example of sharing without friendship appears in the treatment of the Observer pattern in [14].

Separation Logic. The development of separation logic [7, 23] attacks the framing problem from a different more low-level perspective. The idea is to extend the condition language of Hoare logic with a *separating conjunction* operator $\star$, with the following intuitive semantics: condition $P \star Q$ is true if and only if P and Q hold for disjoint parts of the heap. Framing is handled by the following *frame rule*:

$$\frac{\{P\}C\{Q\}}{\{P \star R\}C\{Q \star R\}}$$

where R is a condition that has none of the variables modified by C. The idea is that the implementer of a program C proves the local property $P\{C\}Q$ and the client uses the frame rule to prove the wider property $\{P \star R\}C\{Q \star R\}$ that the client needs. Separation logic handles well many intricate low-level examples with pointers, even with pointer arithmetic, but until recently it has not been considered in the presence of information hiding.

O'Hearn et al.'s work [8] is a first attempt to deal with information hiding in separation logic. The solution does not scale to dynamic modularity, i.e. it deals only with single instances of a hidden data structure [22]. Thus, it is not suitable for the dynamic modularity of object orientation in which the solution must usually be applied to arbitrarily many objects.

Parkinson and Bierman [22] provide a much more complete treatment based on their introduction of *abstract predicates* (very similar to our notion of invariant). However, this work is heavily based on the Frame Rule, which insists on complete heap-separation of the client predicate R from the implementer's predicates P, Q. This is inappropriate in the case of sharing, like the example of Sect. 3.3. A client of the *IteratorSpec* module may hold two iterators attached to the same list object. The representation of their *pos* specification attribute depends on their representation regions as well as the representation region of the shared list object. Thus, the representations of these two specification attributes are not heap-separated. The dynamic frames theory can show that invoking *next* on

one of them preserves the value of the other. It is unclear how to do that using the frame rule of separation logic.

5 Conclusion

This paper has introduced *Dynamic Frames*, a simple and flexible solution to the frame problem for programming theories that support both specification variables and pointers. The solution is simple in that it uses the already existing and well-understood formalism of specification variables. It is more flexible than other approaches because it does not introduce any methodological restrictions for the programmer. *Dynamic Frames* is part of the object oriented theory of [14]. The reader is referred to [14] for further notational and methodological conventions, metatheorems and examples.

Acknowledgments. I would like to thank Eric Hehner, Gary Leavens, Rustan Leino, and Peter Müller, the members of the IFIP WG2.3 and the members of the Formal Methods Group of the University of Toronto for valuable feedback concerning the theory, the presentation and the related work.

References

[1] R. Back and J. vonWright. *Refinement Calculus. A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[2] M. Barnett, R. DeLine, M. Fahndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 2004.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# specification language: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2004.

[4] M. Barnett and D. Naumann. Friends need a bit more: maintaining invariants over shared state. In D. Kozen, editor, *Proceedings of MPC'04: Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag, 2004.

[5] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep-exposure. Technical Report 156, DEC-SRC, 1998.

[6] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 2005. To appear. Available on-line at `http://sct.inf.ethz.ch/publications/index.html` .

[7] P. O' Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01: Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.

[8] P. O' Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *Proceedings of POPL'04: Principles of Programming Languages*, pages 268–280, 2004.

[9] E. C. R. Hehner. *A Practical Theory of Programming*. Current edition, 2004. Available on-line: `http://www.cs.toronto.edu/~hehner/aPToP/` First edition was published by Springer-Verlag in 1993.

[10] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[11] I. T. Kassios. Object orientation in predicative programming, unification and decoupling in object orientation. Technical Report 500, Computer Systems Research Group, University of Toronto, 2004. Available on-line: `http://www.cs.toronto.edu/~ykass/work/oopp.ps.gz`.

[12] I. T. Kassios. Decoupling in object orientation. In *FM'05 World Congress on Formal Methods*, Lecture Notes in Computer Science, pages 43–58, 2005.

[13] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. Technical Report 528, Computer Systems Research Group, University of Toronto, 2005. Current version available on-line: `http://www.cs.toronto.edu/~ykass/work/DFcv.ps`.

[14] I. T. Kassios. *A theory of object oriented refinement*. PhD thesis, Dept. of Computer Science, University of Toronto, 2006.

[15] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference on Object Oriented Programming Systems Languages and Applications*, pages 144–153. ACM, 1998.

[16] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *Proceedings of ECOOP'04: European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[17] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *Proceedings of ESOP'06: European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2006.

[18] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *TOPLAS*, 24(5), 2002.

[19] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 246–257. ACM Press, 2002.

[20] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science. Springer-Verlag, 2002.

[21] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency Computation Practice and Experience*, 15:117–154, 2003.

[22] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of POPL'05: Principles of Programming Languages*, pages 247–258, 2005.

[23] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.