DOI 10.1007/s00165-010-0152-5
BCS © 2010
Formal Aspects of Computing (2011) 23: 267–288

Formal Aspects of Computing

The dynamic frames theory

I. T. Kassios

Chair of Programming Methodolgy RZ F 1, ETH Zurich, Clausiusstrasse 59, 8092 Zurich, Switzerland.
E-mail: ioannis.kassios@inf.ethz.ch

Abstract. The theory of Dynamic Frames has been invented to deal with the frame problem in the presence of encapsulation and pointers. It has proved more flexible and conceptually simpler than previous approaches that tackled the problem. It is now being actively used both for theoretical and for practical purposes related to the formal verification of program correctness. This paper presents the full theory of Dynamic Frames, together with its reasoning laws and exemplifies the use of these laws in proving correct several common design patterns. It also discusses the ongoing research on the topic.

Keywords: Frame problem, Specification languages

1. Introduction

A significant aspect of the behavior of any operation is what parts of the world it leaves unchanged. Usually, a specification for an operation is split into a *functional requirement*, which describes what changes the specified operation brings about and a *framing requirement*, which describes the *frame* of the operation, i.e. the part of the world which the operation has license to change. Anything outside the frame is left unchanged. The *frame problem*, i.e. the problem of how to formalize framing requirements, is a central issue in formal logic, first studied in the context of artificial intelligence [MH69]. In the context of software specification, the "operations" are computations (procedures, methods, etc.) and the "world" is the state.

For example, suppose that we want to specify that a computation C increments program variable x by 1. In a relational theory, like [Heh93], 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. The program below cannot ensure that y ends up being 0:

$$y := 0\,;\,C$$

Correspondence and offprint requests to: I. T. Kassios, E-mail: ioannis.kassios@inf.ethz.ch

In a non-modular setting like [Heh93], 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:

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

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.

Adding encapsulation complicates matters even further: now the client does not know the exact variables that constitute the implementation of the operation under use. Still, the specification must speak in a language that the client can understand. It must somehow refer indirectly to the hidden state of the implementer, modelling it with an abstract state that is known to the client. This is achieved either by specification variables/attributes (also known as model variables/fields) or by pure functions/methods. We are using specification variables and attributes here, but this choice is a matter of taste; other formalisms are also compatible with the rest of the paper. A *specification variable* is essentially a function on the state that represents the relation between the abstract state that the client imagines and the concrete state of the implementation.

Example 1.1 One 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 $S' = S \ \wedge \ return' = (x \in S)$
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:

```
module $ARef$
 prog var $L \in \mathbb{Z}^*$
 spec var $S = \{x \in \mathbb{Z} \mid \exists i \in \mathbb{N} \cdot \ i < \#L \ \wedge \ x = L\, i\}$

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

In the above module, the new syntax used should be fairly straightforward: X^* denotes the set of finite lists of items in X, $^\frown$ denotes list concatenation, $[x]$ denotes a one-item list, #L denotes the size of list L and juxtaposition denotes zero-based list indexing. Module *ARef* is a *refinement* of Module *ASpec* but not yet an implementation, as the operations have not been implemented yet. Further refinements will give an implementation, but this is not the point of this example.

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

$$\textbf{modifies } S \tag{4}$$

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:

$$y' = y \ \wedge \ z' = z$$

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. When that happens, changes to L may change the value of y, contrary to what is predicted by the theory. This situation is called *abstract aliasing* [LN02].

Various approaches are proposed in the literature to deal with this problem. Perhaps the most flexible and general solution is the *theory of Dynamic Frames* [Kas06a, Kas06b]. Unlike other systems which seek to avoid the possibility of abstract aliasing by imposing various restrictions to the programmer, the key idea behind the theory of Dynamic Frames is to make abstract aliasing directly expressible in the specification language. This way, the theory succeeds to solve the framing problem without restricting the programmer and without introducing any new special formal constructs (Dynamic Frames are simply set-valued specification variables).

The use of the theory was first presented in [Kas06a]. In that paper, only specification/implementation examples were given, but no example of formal proofs was made and the techniques and theorems behind Dynamic Frames formal proofs were not shown. The present paper attempts a more comprehensive coverage of the theory, which includes the proof laws, the proof examples and yet one more example which were omitted from [Kas06a]. The paper also revises of work that was based on the theory since its invention, as well as future research directions.

2. Preliminaries

Here we introduce the basic mathematical notation, as well as the basic notions of specifications, implementations, modules etc. to be used in the rest of the paper. Further notation and definitions will be introduced as we proceed in the presentation of theory.

2.1. Basic mathematical notation

We will be using standard logical, arithmetic and set-theoretic notation, with which we hope most readers are very familiar. Besides that, we will be using the notation which is introduced in this subsection.

Large operators. The operators $=\Longleftarrow\Longrightarrow$ have the same semantics as $=\Leftarrow\Rightarrow$ but lowest precedence. They are used to reduce the number of parentheses in expressions.

Sets and Set Notation. The set of booleans $\{\top, \bot\}$ is denoted $\mathbb{B}$. The symbol $\top$ stands for "true". The set of rational numbers is denoted $\mathbb{Q}$. 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:

$$\begin{aligned}\{i, ..j\} &= \{x \in \mathbb{Z} \cdot \ i \le x < j\}\\ \{i, ..., j\} &= \{x \in \mathbb{Z} \cdot \ i \le x \le j\}\end{aligned}$$

Notice that $\{i, ..j\}$ does not include 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 the 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, identifiers may *stand for* expressions that may contain free variables. We may say e.g. that E is an *expression* on variables $x, y, \ldots$. We call such identifiers "open expressions".

The reason open expressions are used is because some quantities of interest depend on the same small set of variables (for example, the *program state* σ), which we therefore rather keep implicit to reduce the notational clutter. Readers who are uncomfortable with open expressions, may consider the occurrence of an open expression E on variables $x, y, \ldots$ 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*. There is also an infinite set of *values Val* that contains at least the booleans and integers.

Definition 2.1 (Region) Any subset of *Loc* is called a *region*.

Definition 2.2 (State and allocation) 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.

Before introducing object orientation, our basic state abstraction mechanism will be *specification variables*. A specification variable is used as a visible abstraction of part of the state that is supposed to be hidden by the client. In our theory, it is just a state-dependent expression:

Definition 2.3 (Spec. variable) A *specification variable* is an open expression that depends only on the state σ (i.e. with free occurrences of the variable $\sigma \in \Sigma$).

We will see later examples of the use of specification variables as abstractions. For now, we introduce two important specification variables: the set of all allocated locations *Used* and the set of all unallocated locations *Unused*:

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

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

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

As we will be using σ to represent the initial state of a computation and σ' to represent the final state of the computation, the expression v' is called the *final value of* v.

The actual variables of imperative computations, here called program variables, can be represented as special cases of specification variables:

Definition 2.4 (Program variable) 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:

$$x = \sigma(\mathit{addr\_x})$$

Imperative specifications. Imperative specifications are modelled using the relational style:

Definition 2.5 (Imperative spec.) An *imperative specification* is a boolean expression on the state-valued variables σ, σ'. 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:

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

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. We now provide a module construct that will hold all our specifications and implementations:

Definition 2.6 (Module) 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.

The correct implementation of a specification is capture by the notion of *module refinement*:

Definition 2.7 (Refinement) A module M *refines* (or *implements*) a module N if the axioms of M imply the axioms of N and the names declared in M are included in the names declared in N.

Procedures. In a module, we may introduce *procedures*, using the keyword **proc** followed by the name of the procedure, its parameters in parentheses and an imperative specification:

$$\textbf{proc}\ p(x_1, \ldots, x_n) \cdot\ S \tag{5}$$

The axiom (5) promises that invoking the procedure will satisfy the imperative specification S. A procedure invocation $p(x_1, \ldots, x_n)$ is an imperative specification.

Formally (5) where p and the x_i are identifiers and S is an imperative specification, is an abbreviation for the following axiom:

$$\forall x_1 \in Val, \ldots, x_n \in Val \cdot \sigma \in \Sigma, \sigma' \in \Sigma \cdot\ p(x_1, \ldots, x_n) \Rightarrow\ S$$

If a procedure is written using only programming constructs, then it is a program and its implementation is finished.

Example 2.1 (Rational number) As an example of the use of modules, module refinement, specification and program variables, we will show their use in the implementation of a rational number module. Classes have not been introduced yet, so we will have to do with a single rational number.

The specification for this is given in module *Rat*:

module *Rat*
 spec var $rat\_inv \in \mathbb{B}$, rat
 $rat\_inv \;\Rightarrow\; rat \in \mathbb{Q}$
 proc $double() \cdot\; rat\_inv \Rightarrow rat' = 2 \times rat \;\wedge\; rat\_inv'$
end module

The module introduces specification variables *rat_inv* and *rat*. The boolean specification variable *rat_inv* is abstracting the *module invariant*, a condition that must hold for the operations of the module to behave appropriately. The specification attribute *rat* is the abstract rational value that the client "sees". The client cannot see the actual representation of these specification variables in program variables. An axiom promises that as long as the invariant holds, what the client will see is a rational number.

The imperative specification of the procedure *double* promises to double the represented rational number, as long as the invariant initially holds (the latter requirement is typically separated as a *pre-condition* in many specification languages). It also promises to preserve the invariant.

Of course this specification is practically useless, since it provides no observer procedures. However, this is not an important part of the specification as far as we are concerned. Defining observer procedures would only clutter the example. We can imagine that our module also contains specifications for some observer procedures that we do not show here. The same policy will be followed in later examples.

To implement this specification, we must write a module *RatImpl* providing definitions of all the specification variables and procedures. Here is such an implementation:

module *RatImpl*
 prog var nom, $denom$
 spec var $rat\_inv \;=\; (nom \in \mathbb{Z} \;\wedge\; denom \in \mathbb{N}\backslash\{0\})$
 spec var $rat \;=\; nom/denom$
 proc $double() \cdot\; nom := 2 \times nom$
end module

This is a valid implementation, as it is very easy to show that the axioms of *RatImpl* imply those of *Rat*. It is not a very surprising implementation, neither in terms of the definition of its specification attributes nor in terms of the implementation of procedure *double*.

3. The Dynamic Frames theory

Example 2.1 has no support for framing, so the module *Rat* is not very useful for a client. For example, suppose that we have a client of *Rat* with its own variable x. The client wishes to invoke *double* but at the same time preserve the value of x. Formally, the client wishes to prove:

$$double() \;\Rightarrow\; x' = x$$

It is impossible to prove that using the specification of *double* which makes no promise about any part of the state other than the representation of the rational number itself. In this section, we introduce the basics of our framing specification methodology which will solve this problem.

3.1. Framing with regions

In the Dynamic Frames theory, all properties related to framing are formed using regions:

Definition 3.1 (Region) A *region* is a set of locations.

Regions are used to write imperative framing requirements, which specify the parts of the state that a method is allowed to change. There are two different ways to write imperative framing requirements: the "preserves" operator Ξ and the "modifies" operator Δ. Preservation specifies the part of the state that should be left untouched. Modification specifies the part of the state that we have license to change.

Definition 3.2 (Imperative framing requirements) Let f be a region. The *preservation* of f is an imperative specification Ξf defined as follows:

$$\Xi f \;=\; \sigma' \triangleright f = \sigma \triangleright f$$

The *modification* of f is an imperative specification Δf defined as follows:

$$\Delta f = \Xi(Used \backslash f)$$

Notice that Δf allows the allocation of new memory.

Regions are also used to *frame* expressions that vary with state, such as specification variables. When a region f *frames* a state-dependent expression E, that means that E depends only on the locations in f.

Definition 3.3 (Expression framing) Let f be a region and E an expression on σ. We say that f *frames* E in state σ if the following holds:

$$\forall \sigma' \in \Sigma \cdot \ \Xi f \ \Rightarrow \ E' = E \tag{6}$$

Notice that expression framing is a state condition: f may frame E in some states but not in others. State condition (6) is abbreviated by

$$f \ \mathbf{frames} \ E$$

More than one expression can be written to the right of **frames** separated by commas:

$$f \ \mathbf{frames} \ (x, y, \ldots) \ = \ f \ \mathbf{frames} \ x \wedge f \ \mathbf{frames} \ y \wedge \ldots$$

Two expressions E, D on σ are known to be *independent* in state σ if they are known to have disjoint frames in that state, i.e. for some regions f, g:

$$f \ \mathbf{frames} \ E \wedge g \ \mathbf{frames} \ D \wedge \mathit{disjoint}[f;\ g]$$

When that happens, the modification of f guarantees the preservation of the value of D (and similarly, the modification of g guarantees the preservation of the value of E):

Theorem 3.1 (Value preservation) Let f, g be regions and D be an expression on σ. For any states σ, σ', the following holds:

$$\Delta f \ \wedge \ g \ \mathbf{frames} \ D \ \wedge \ \mathit{disjoint}[f;\ g] \ \Rightarrow \ D' = D \tag{7}$$

Proof. The formal proof is as follows:

$$\begin{array}{ll} & \Delta f \ \wedge \ g \ \mathbf{frames} \ D \ \wedge \ \mathit{disjoint}[f;\ j] \\ = & \Xi(Used \backslash f) \ \wedge \ g \ \mathbf{frames} \ D \ \wedge \ \mathit{disjoint}[f;\ j] \\ \Rightarrow & \Xi g \ \wedge \ g \ \mathbf{frames} \ D \\ \Rightarrow & D' = D \end{array}$$

□

3.2. Dynamic Frames and variable framing

The methodology of Dynamic Frames is based on Theorem. 3.1. To make use of it, we define specification variables that play the roles of f and g in (7). They are called dynamic frames:

Definition 3.4 (Dynamic Frame) A specification variable f is a *dynamic frame* (or simply *frame*) at state σ, if

$$f \subseteq Used$$

Notice that being a dynamic frame is a state condition.

For every specification variable v that we use in a theory, we introduce a frame f to frame it. Frames do not have to be unique; more than one specification attribute may share the same frame. The fact that f is a frame is usually made part of an invariant:

$$inv \ \Rightarrow \ f \subseteq Used$$

The specification variable-framing property for v is also usually asserted as part of the invariant:

$$inv \ \Rightarrow \ f \ \mathbf{frames} \ v$$

In most cases, it is enough to introduce one frame for all the specification variables of interest. We use the name *rep* for that frame. We call the value of *rep* the *representation region*. We do not need to introduce frames for program variables: the frame of program variable m is the singleton that contains its address $\{addr\_m\}$.

Example 3.1 (Rational number with frames) To address the frame problem in Example 2.1, we introduce a frame *rat_rep* and assert that it frames both *rat* and *rat_inv*:

$$rat\_inv \;\Rightarrow\; rat\_rep \subseteq Used \;\wedge\; rat\_rep \;\textbf{frames}\; (rat\_inv, rat)$$

The full specification now becomes:

```
module $Rat$
 spec var $rat\_inv \in \mathbb{B}, rat, rep$
 $rat\_inv \;\Rightarrow\; rat \in \mathbb{Q}$
 $rat\_inv \;\Rightarrow\; rat\_rep \subseteq Used \;\wedge\; rat\_rep$ frames $(rat\_inv, rat)$
 proc $double() \cdot\; rat\_inv \Rightarrow rat' = 2 \times rat \;\wedge\; rat\_inv'$
end module
```

Consider now a client that introduces a new variable x:

```
module $Client$
 import $Rat$
 spec var $c\_inv \in \mathbb{B}, x, c\_rep$
 $c\_inv \;\Rightarrow\; rat\_inv \wedge c\_rep \subseteq Used \;\wedge x \in \mathbb{Z}$
 $c\_inv \;\Rightarrow\; c\_rep$ frames $(c\_inv, x) \wedge disjoint[rat\_rep, c\_rep]$
...
end module
```

It is straightforward for the client to prove, using Theorem 3.1 that invocation of *double( )* will not touch x:

$$double(\,) \;\wedge\; c\_inv \;\Rightarrow\; x' = x$$

It is the implementer's obligation to find implementations for the specification attributes (including the frames) that satisfy the requirements of the theory. The module *RatImpl* already implements everything, except for *rat_rep*. In this simple case, it is very easy and the frame happens to be constant:

$$rat\_rep = \{addr\_nom, addr\_denom\}$$

So, the whole module *RatImpl* becomes:

```
module $RatImpl$
 prog var $nom, denom$
 spec var $rat\_inv \;=\; (nom \in \mathbb{Z} \;\wedge\; denom \in \mathbb{N}\backslash\{0\})$
 spec var $rat \;=\; nom/denom$
 spec var $rat\_rep \;=\; \{addr\_nom, addr\_denom\}$
 proc $double() \cdot\; nom := 2 \times nom$
end module
```

The example demonstrates not only the use, but also the modularity of the Dynamic Frames approach. In Theorem 3.1, it is the implementer's responsibility to ensure Δf, without knowing about g and D. These belong to the client, whose responsibility is to ensure $disjoint[f;\; g]$ as well as g **frames** D.

In this simple example, the frame is constant. In more interesting examples later on, frames vary with state, which makes their handling more challenging.

3.3. Objects

To show the full-fledged theory of Dynamic Frames, apart from encapsulation, we need references and dynamic allocation. For that reason, we introduce a small mathematical model of objects. The model is far from being a complete formalization of object orientation; we only introduce the features that we need.

References.

Definition 3.5 (References) 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 $\mathcal{O}$.

Attributes and methods. Specification and program attributes and methods are introduced in complete analogy with specification and program variables and procedures: they only depend on one more mathematical variable *self* of type $\mathcal{O}$. The variable *self* is called the *current* object.

Definition 3.6 (Specification attribute) A *specification attribute* is an open expression with free occurrences of the identifiers $\sigma \in \Sigma$ and $self \in \mathcal{O}$ only. A *program attribute* x is a special case of specification attribute such that

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

for some open expression *addr_x* that depends on *self* only. The location *addr_x* is called the *address* of x.

Specification attributes are also called *model fields*. Elsewhere *pure methods* are employed for basically the same abstraction mechanism on objects.

The keyword **spec attr** introduces specification attributes. The keyword **prog attr** introduces program attributes. The definitions for concrete assignments 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 $k \in \mathbb{N}$)

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

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

$$init \in \mathbb{B} \ \wedge \ inv \in \mathbb{B} \ \wedge \ (init \Rightarrow inv) \ \wedge \ (inv \Rightarrow rep \subseteq Used) \tag{8}$$

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

$$null.rep = \emptyset$$

Definition 3.7 (Method invocation) 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.

Definition 3.8 (Class) A *class* is a set of object references.

The specification of a 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. 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.

Method specifications. Method specifications have the form:

$$\forall x \cdot \forall y \cdot \ \dots \ \textit{self}.l(x;\ y;\ \dots) \ \Rightarrow \ S \tag{9}$$

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 (9) is abbreviated by

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

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 $x := \textbf{new}\ \ C$ defined as follows:

$$x := \textbf{new}\ \ C \ = \ \Delta\{\textit{addr\_x}\} \ \wedge \ x' \in C \ \wedge \ (x\textbf{.}\textit{init})' \ \wedge \ (x\textbf{.}\textit{rep})' \subseteq \textit{Unused} \backslash \{\textit{addr\_x}\}$$

where x is a program variable.

3.4. Establishing and maintaining frame disjointness

The use of Theorem 3.1 to prove the preservation of values is only half of the story. The other half is how we establish and maintain frame disjointness, which is one of the requirements of Theorem 3.1. Establishing disjointness is easy, as we can see from the definition of the **new** statement, which creates a new frame from previously un-allocated memory, and therefore disjoint from any other frame in use.

The problem of maintaining disjointness is not trivial, because in general dynamic frames may vary with state. This does not happen in our simple rational number example, but it happens when we use more elaborated data structures, as we do in the following sections.

Dynamic frames are specification attributes too. To guarantee preservation of disjointness (or any other property of dynamic frames for that matter), dynamic frames need to be framed too. The most obvious solution is to make dynamic frames self-framing. Suppose that f, g are two dynamic frames. If g is self-framing and initially disjoint from f, then, by Theorem 3.1, the modification of f preserves the value of g. Thus, if we also ensure that f increases only by unused addresses, we ensure that f remains disjoint from g:

$$\begin{aligned} &f \cup g \subseteq \textit{Used} \ \wedge \ g \ \textbf{frames}\ g \ \wedge \ \textit{disjoint}[f;\ g] \ \wedge \ \Delta f \ \ \wedge \ f' \subseteq f \cup \textit{Unused} \\ \Longrightarrow\ &(\textit{disjoint}[f;\ g])' \end{aligned} \tag{10}$$

The requirement $f' \subseteq f \cup \textit{Unused}$ is our equivalent of what Leino and Nelson [LN02] call the *swinging pivots* requirement. In their theory, this is enforced. We can be more general. We can generalize (10) so that the region that we frame upon f is *different* from the frame h whose disjointness we want to ensure from g:

Theorem 3.2 (Disjointness preservation) Let f, g, h be dynamic frames. Then the following holds:

$$\begin{aligned} &f \cup g \subseteq \textit{Used} \ \wedge \ g \ \textbf{frames}\ g \ \wedge \ \textit{disjoint}[f;\ g] \ \wedge \ \Delta f \ \ \wedge \ h' \subseteq f \cup \textit{Unused} \\ \Longrightarrow\ &(\textit{disjoint}[h;\ g])' \end{aligned} \tag{11}$$

Proof. The proof of the theorem is as follows:

$$\begin{aligned} &f \cup g \subseteq \textit{Used} \ \wedge \ g \ \textbf{frames}\ g \ \wedge \ \textit{disjoint}[f;\ g] \ \wedge \ \Delta f \ \ \wedge \ h' \subseteq f \cup \textit{Unused} \\ &\hfill \text{Theorem 3.1} \\ \Longrightarrow\ \ &g' = g \ \wedge \ \textit{disjoint}[f \cup \textit{Unused}\,;\ \ g] \ \wedge \ h' \subseteq f \cup \textit{Unused} \\ \Longrightarrow\ \ &\textit{disjoint}[h';\ g'] \end{aligned}$$

□

Notice that (10) is a special case of Theorem 3.2, in which f and h are the same dynamic frame. But, more generally, we are allowed to have h "swallow" (part of) another frame f_1, if

$$f \ = \ h \cup f_1$$

a situation which violates the swinging pivots requirement and therefore is forbidden in [LN02].

3.5. Programming laws

In this section, we prove three programming laws, which calculate the overall framing and functional effect of sequences of specifications. The purpose of these laws is to remove sequential compositions from imperative specifications, which is an important part of correctness proofs.

Frame accumulation. The *frame accumulation law* calculates the overall framing effect of a sequential composition. Suppose that we have a sequential composition $P;Q$ and we want to prove that it refines Δf, where f varies with state. A way to do it is to prove that:

- P refines Δf.
- Q refines Δg, for dynamic frame g whose value at the intermediate state is included in the value of f in the initial state.

Formally:

$$\Delta f \wedge g' \subseteq f \,;\, \Delta g \implies \Delta f$$

A small generalization is the following: because Δ allows modification of all unused addresses, the intermediate value of g may also contain some unused addresses. The final version of the frame accumulation law is as follows:

Theorem 3.3 (Frame accumulation) Let f, g be dynamic frames. Then the following holds:

$$\Delta f \wedge g' \subseteq f \cup \mathit{Unused} \,;\, \Delta g \implies \Delta f$$

Proof. The proof goes as follows:

$$\begin{array}{ll}
 & \Delta f \wedge g' \subseteq \Delta f \cup \mathit{Unused} \,;\, \Delta g \\
 & \hfill \text{sequential composition} \\
= & \exists \sigma'' \cdot \quad \underline{\sigma'' \rhd (\mathit{Used}\backslash f) = \sigma \rhd (\mathit{Used}\backslash f)} \;\wedge\; g'' \subseteq f \cup \mathit{Unused} \\
 & \qquad\quad \wedge\; \underline{\sigma' \rhd (\mathit{Used}''\backslash g'') = \sigma'' \rhd (\mathit{Used}''\backslash g'')} \\
 & \hfill \text{domain restriction composition} \\
\implies & \exists \sigma'' \cdot \quad \underline{\sigma' \rhd ((\mathit{Used}\backslash f) \cap (\mathit{Used}''\backslash g'')) = \sigma \rhd ((\mathit{Used}\backslash f) \cap (\mathit{Used}''\backslash g''))} \;\wedge\; g'' \subseteq f \cup \mathit{Unused} \\
 & \qquad\quad \wedge\; \sigma' \rhd (\mathit{Used}''\backslash g'') = \sigma'' \rhd (\mathit{Used}''\backslash g'') \\
 & \hfill \text{preservation operator} \\
= & \exists \sigma'' \cdot \quad \Xi((\mathit{Used}\backslash f) \cap (\mathit{Used}''\backslash g'')) \;\wedge\; g'' \subseteq f \cup \mathit{Unused} \\
 & \qquad\quad \wedge\; \sigma' \rhd (\mathit{Used}''\backslash g'') = \sigma'' \rhd (\mathit{Used}''\backslash g'')
\end{array}$$

We now pause the main proof and observe that the body of the existential quantification implies two things. First, we prove that: $\mathit{Used}'' \supseteq \mathit{Used}\backslash f$ as follows:

$$\begin{array}{ll}
 & \sigma'' \rhd (\mathit{Used}\backslash f) = \sigma \rhd (\mathit{Used}\backslash f) \\
 & \text{function equality implies domain equality} \\
\implies & \mathit{Used}'' \cap (\mathit{Used}\backslash f) = \mathit{Used} \cap (\mathit{Used}\backslash f) \\
= & \mathit{Used}'' \cap (\mathit{Used}\backslash f) = \mathit{Used}\backslash f \\
= & \mathit{Used}'' \supseteq \mathit{Used}\backslash f
\end{array}$$

Second, we prove that $\mathit{Used}''\backslash g'' \supseteq \mathit{Used}\backslash f$

$$\begin{array}{ll}
 & \underline{\mathit{Used}''}\backslash g'' \\
 & \hfill \text{from proof above} \\
\supseteq & (\mathit{Used}\backslash f)\backslash \underline{g''} \\
 & \text{from conjunct } g'' \subseteq f \cup \mathit{Unused} \\
\supseteq & (\mathit{Used}\backslash f)\backslash (f \cup \mathit{Unused}) \\
= & \mathit{Used}\backslash f
\end{array}$$

The main proof continues as follows:

$$\begin{array}{ll} & \Delta f \wedge g' \subseteq \Delta f \cup \mathit{Unused}\,;\, \Delta g \\ \Longrightarrow & \exists \sigma'' \cdot\ \Xi((\mathit{Used} \backslash f) \cap (\mathit{Used}'' \backslash g'')) \ \wedge\ \mathit{Used}'' \backslash g'' \supseteq \mathit{Used} \backslash f \\ \Longrightarrow & \Xi(\mathit{Used} \backslash f) \\ = & \Delta f \end{array}$$

□

A special case of Theorem 3.3 is:

$$f \subseteq \mathit{Used} \ \wedge\ (\Delta f \wedge f \subseteq f \cup \mathit{Used}\,;\, \Delta f) \ \Rightarrow\ \Delta f$$

where the first specification obeys the swinging pivots requirement.

Substitution. Like its counterpart in [Heh93], the substitution law is used to calculate the functional effect of $P;Q$, where P is an assignment. We make it more generally applicable, by allowing P to be any conjunction of imperative specifications of the form $X' = A$ where the X, A are expressions on σ.

Theorem 3.4 (Substitution) Let $X, Y, \ldots, A, B, \ldots$ be expressions on σ and F a predicate. Then

$$X' = A \wedge Y' = B \wedge \ldots\,;\, F\ X\ Y\ \ldots\sigma' \Longrightarrow F\ A\ B\ \ldots\ \sigma'$$

Proof. The proof is as follows:

$$\begin{array}{ll} & X' = A \wedge Y' = B \wedge \ldots\,;\, F\ X\ Y\ \ldots\sigma' \\ & \qquad\qquad\qquad \text{sequential composition} \\ = & \exists \sigma'' \cdot\ X'' = A \wedge Y'' = B \wedge \ldots\ \wedge\ F\ X''\ Y'' \ldots\sigma' \\ = & \exists \sigma'' \cdot\ X'' = A \wedge Y'' = B \wedge \ldots\ \wedge\ F\ A\ B \ldots\sigma' \\ \Longrightarrow & F\ A\ B \ldots\sigma' \end{array}$$

□

Frame restriction accumulation. Frame restrictions are imperative specifications of the form

$$f' \subseteq f \cup h$$

where the f, h are dynamic frames. Frame restrictions are very important for proving the preservation of frame disjointness (Theorem 3.2).

The frame restriction accumulation law calculates the overall frame restriction of a sequential composition $P;Q$. As in the case of frame accumulation, a way to prove that $P;Q$ refines $f' \subseteq f \cup h$, where f, h are dynamic frames, is to prove that:

- P refines $f' \subseteq f \cup h$
- Q refines $f' \subseteq f \cup g$ for some expression g whose value in the intermediate state is included in the value of $f \cup h$ in the initial state.

Theorem 3.5 (Frame restriction accumulation) Let f, g, h be dynamic frames. The following holds:

$$f' \cup g' \subseteq f \cup h\,;\, f' \subseteq f \cup g \Longrightarrow f' \subseteq f \cup h$$

Proof. The proof is as follows:

$$\begin{array}{ll} & f' \cup g' \subseteq f \cup h\,;\, f' \subseteq f \cup g \\ & \qquad\qquad\qquad \text{sequential composition} \\ = & \exists \sigma'' \cdot\ f'' \cup g'' \subseteq f \cup h \ \wedge\ f' \subseteq f'' \cup g'' \\ & \qquad\qquad\qquad\qquad \text{transitivity} \\ \Longrightarrow & f' \subseteq f \cup h \end{array}$$

□

3.6. Auxiliary notation

The Dynamic Frames notation which we have seen so far is quite verbose. In this section, we introduce auxiliary notation that attacks the verbosity problem.

Abstract assignment. More often than not, we will see that there is a single dynamic frame *rep* that frames the invariant, itself and all the other specification variables in the class:

$$\mathit{inv} \quad \Rightarrow \quad \mathit{rep}\ \textbf{frames}\ (\mathit{inv}, \mathit{rep}, x_1, x_2 ...)$$

A frequent case is to change one of the specification variables x_i while preserving all the others, including the invariant, and maintaining the swinging pivots requirement for *rep*. We will abbreviate this case with $x_i ::= E$ and we will call it *abstract assignment*:

$$x_i ::= E \;=\; \mathit{inv} \;\Rightarrow\; \Delta \mathit{rep} \;\wedge\; x_i' = E \;\wedge\; \mathit{inv}' \;\wedge\; \mathit{rep}' \subseteq \mathit{rep} \cup \mathit{Used} \;\wedge\; \forall j \neq i \cdot x_j' = x_j$$

The $\overline{\Delta}$ operation. Notice that an operation that satisfies Δf, where f is self-framing, is not in a position to extend f in any way other than by previously unallocated memory. So we can invent a stronger notation for both facts Δf and $f' \subseteq f \cup \mathit{Used}$. This is the $\overline{\Delta}$ operator, called *strong frame*:

$$\overline{\Delta} f \;=\; \Delta f \;\wedge\; f' \subseteq f \cup \mathit{Used}$$

Strong frames have a convenient accumulation law:

Theorem 3.6 (Strong frame accumulation) Let f, g be dynamic frames. The following holds:

$$\overline{\Delta} f \;\wedge\; g' \subseteq f \cup \mathit{Unused} \;\mathbf{;}\; \overline{\Delta} g \;\wedge\; f' \subseteq g \cup \mathit{Unused} \;\Longrightarrow\; \overline{\Delta} f$$

Proof. Immediate from Theorem 3.3 and

$$\begin{array}{cl}
 & \Delta f \\
= & \sigma' \rhd (\mathit{Used} \backslash f) = \sigma \rhd (\mathit{Used} \backslash f) \\
 & \text{domain of } \sigma' \rhd (\mathit{Used} \backslash f) \text{ is at least } \mathit{Used} \backslash f \\
\Longrightarrow & \mathit{Used}' \supseteq \mathit{Used} \backslash f \\
\Longrightarrow & \mathit{Unused}' \subseteq \mathit{Loc} \backslash (\mathit{Unused} \backslash f) \\
 & \hfill \text{set theory} \\
\Longrightarrow & \mathit{Unused}' \subseteq \mathit{Unused} \cup f
\end{array}$$

and

$$\begin{array}{cl}
 & \Delta f \;\wedge\; g' \subseteq f \cup \mathit{Unused}\,\mathbf{;}\, f' \subseteq g \cup \mathit{Unused} \\
 & \hfill \text{sequential composition and the above lemma} \\
= & \exists \sigma'' \cdot\; g'' \subseteq f \cup \mathit{Unused} \;\wedge\; f' \subseteq g'' \cup \mathit{Unused}'' \;\wedge\; \mathit{Unused}'' \subseteq \mathit{Unused} \cup f \\
 & \hfill \text{transitivity} \\
\Longrightarrow & f' \subseteq f \cup \mathit{Unused}
\end{array}$$

□

4. Examples

In this section, we present some examples of specification and implementation in the theory of Dynamic Frames. Most proofs of correctness are omitted, but some interesting ones are shown. We also omit pure methods that return information to the client, because they are not relevant to the paper. For more comprehensive coverage, the reader is referred to [Kas06b].

4.1. Lists

This example concerns the specification and implementation of a class *List* that formalizes lists of integers.

Specification. 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 $L$
  $inv \;\Rightarrow\; L \in \mathbb{Z}^* \wedge rep \subseteq Used \wedge rep$ frames $(rep, inv, L)$
  $init \;\Rightarrow\; L = [] \wedge inv$
```

The method *insert* inserts an item to the beginning of the list. It does not need to violate the swinging pivots requirement: the representation region of the current object may only increase by newly allocated memory.

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

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

Finally, the method *paste* concatenates a list to the end 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 $paste(p)\cdot$
   $inv \wedge p \in List \wedge p.inv \wedge disjoint[rep\,;\; p.rep] \;\Rightarrow\; \overline{\Delta}(rep \cup p.rep) \wedge L' = L^\frown p.L \wedge inv'$
 end class
end module
```

Implementation. 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 $val,\; next$
  $init \;==\; next = null \wedge val \in \mathbb{Z}$
  $inv \;==\; (next = null \vee next \in Node) \wedge val \in \mathbb{Z}$
  $rep \;=\; \{addr\_val, addr\_next\}$
 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** $L = \lambda i \in \{0,..len\} \cdot head.[next]^i.val$
$rep \;=\; \{addr\_head\} \cup \bigcup i \in \{0,..len\} \cdot head.[next]^i.rep$
$inv \;\equiv\;$   $(\forall i \in \{0,..len\} \cdot head.[next]^i.val \in \mathbb{Z})$
           $\wedge\; disjoint(\;\; [\{addr\_head\}]$
                       $\frown \lambda\, i \in \{0,..len\} \cdot head.[next]^i.rep\;\; )$
$init \;\equiv\; head = null$

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

Refinement proof. The linked-list implementation refines the specification. We will not show the refinement proof here, which is tedious but easy. We will only prove the framing properties, which are the most interesting part. The proof depends on the theorem of chain framing:

Theorem 4.1 (Chain framing) Let a be a program attribute, $k \in \mathbb{N}$ and $self \in \mathcal{O}$, such that $[a]^k \in \mathcal{O}$. Then the following holds:

$$\{i \in \{0,..k\} \cdot\; [a]^k.(addr\_a)\} \;\; \textbf{frames} \;\; [a]^k \tag{12}$$

Proof. Induction on k. For $k = 0$, (12) becomes:

$$\emptyset \;\; \textbf{frames} \;\; self$$

which is true. For the induction step, we assume (12) and prove the formula for $k + 1$:

$$\begin{array}{ll}
 & \Xi\{i \in \{0,..k+1\} \cdot\; [a]^{k+1}.(addr\_a)\} \\
\equiv & \Xi\{i \in \{0,..k\} \cdot\; [a]^k.(addr\_a)\} \;\wedge\; \sigma'([a]^k.(addr\_a)) = \sigma([a]^k.(addr\_a)) \\
\equiv & \Xi\{i \in \{0,..k\} \cdot\; [a]^k.(addr\_a)\} \;\wedge\; \sigma'((addr\_a)([a]^k/self)) = \sigma((addr\_a)([a]^k/self)) \\
\equiv & \Xi\{i \in \{0,..k\} \cdot\; [a]^k.(addr\_a)\} \;\wedge\; (\sigma'(addr\_a))([a]^k/self) = (\sigma(addr\_a))([a]^k/self) \\
\equiv & \Xi\{i \in \{0,..k\} \cdot\; [a]^k.(addr\_a)\} \;\wedge\; [a]^k.a' = [a]^k.a \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{assumption (12)} \\
\Longrightarrow & ([a]^k)' = [a]^k \;\wedge\; [a]^k.a' = [a]^k.a \\
\Longrightarrow & ([a]^k.a)' = [a]^k.a
\end{array}$$

□

Using the chain framing theorem, we can prove that the specification attribute *len* is framed by *rep*. Assume Ξrep. Then:

$$\begin{array}{ll}
 & len' \\
= & min\{i \in \mathbb{N} \cdot\; (head.[next]^i)' = null\} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{assumption } \Xi rep \\
= & min\{i \in \mathbb{N} \cdot\; head.([next]^i)' = null\} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{chain framing theorem} \\
= & min(\underline{\{i \in \mathbb{N} \cdot\; i \le len \;\wedge\; head.[next]^i = null\}} \cup \{i \in \mathbb{N} \cdot\; i > len \;\wedge\; head.([next]^i)' = null\}) \\
 & \qquad\qquad\qquad\qquad \text{the minimum of the underlined part is } len \\
 & \qquad\qquad\qquad \text{the other set's minimum is definitely greater than } len \\
= & len
\end{array}$$

Now the proof for the framing of the rest of the attributes is a direct consequence of the chain framing theorem. For example, to prove *rep* **frames** L, assume Ξrep and:

$$\begin{array}{ll}
 & L' \\
= & (\lambda i \in \{0, ..len\} \cdot head.[next]^i.val)' \\
 & \hfill \text{lemma about } len \\
= & \lambda i \in \{0, ..len\} \cdot (head.[next]^i.val)' \\
 & \hfill \text{framing} \\
= & \lambda i \in \{0, ..len\} \cdot head.[next]^i.(val)' \\
 & \hfill \text{framing} \\
= & \lambda i \in \{0, ..len\} \cdot head.[next]^i.val \\
= & L
\end{array}$$

This example has illustrated that the theory of Dynamic Frames handles the issue of objects crossing encapsulation borders. It also showed, for the first time in the paper, a case where frames depend dynamically on the state.

4.2. A list client

In this example, we verify a client of the list module, showing the use of the reasoning theorems presented in Sects. 3.5 and 3.6. The specification asks us to put the first item to the end of the list:

```
module LClient
 import ListSpec
 proc cl(l)·
  l.(self ∈ List ∧ L ≠ [] ∧ inv ⇒ Δ̄rep ∧ L′ = L[1; ..#L]⌒L[0] ∧ inv′)
end module
```

The implementation uses the *cut* and *paste* operators:

```
module LClient
 import ListSpec
 proc cl(l) · var p · l.cut(addr_p, 1) ; l.paste(p)
end module
```

The proof of correctness goes as follows:

$$\begin{array}{ll}
 & l.(self \in List \ \wedge\ L \neq [] \ \wedge\ inv) \ \wedge\ cl(l) \\
= & \exists\, \underline{addr\_p \in Unused} \cdot\ l.(self \in List \ \wedge\ \underline{L \neq []} \ \wedge\ \underline{inv}) \ \wedge\ (l.cut(addr\_p, \underline{1})\ ;\ l.paste(p)) \\
 & \hfill \text{all preconditions of } cut \text{ are met} \\
\Rightarrow & \quad \underline{l \in List} \ \wedge\ l.L \neq [] \\
 & \wedge\ \exists\, addr\_p \in Unused \cdot \\
 & \qquad\quad l.(\overline{\Delta}(\{addr\_p\} \cup rep) \ \wedge\ L' = L[1;\ ..\#L] \ \wedge\ \underline{inv'}) \\
 & \qquad \wedge\ \underline{p' \in List} \ \wedge\ (p.L)' = [l.L0] \ \wedge\ \underline{(p.inv)'} \ \wedge\ (p.rep)' \subseteq l.rep \cup Unused \\
 & \qquad \wedge\ \underline{(disjoint[l.rep\ ;\ \ p.rep\ ;\ \ \{addr\_p\}])'} \\
 & \quad ;\ l.paste(p) \\
 & \hfill \text{all preconditions of } paste \text{ are met} \\
\Rightarrow & \quad l \in List \ \wedge\ l.L \neq [] \\
 & \wedge\ \exists\, \underline{addr\_p \in Unused} \cdot \\
 & \qquad\quad l.(\underline{\overline{\Delta}(\{addr\_p\} \cup rep)} \ \wedge\ L' = L[1;\ ..\#L]) \\
 & \qquad \wedge\ (p.L)' = [l.L0] \ \wedge\ (p.rep)' \subseteq l.rep \cup Unused \\
 & \quad ;\ \overline{\Delta}(l.rep \cup p.rep) \ \wedge\ l.L' = l.L^\frown p.L \ \wedge\ (l.inv)' \\
 & \hfill \text{since } addr\_p \in Unused, \text{ it is } \Delta(\{addr\_p\} \cup rep) = \Delta rep
\end{array}$$

$$
\begin{array}{ll}
= & l \in List \ \wedge\ l.L \neq [] \\
& \wedge\ \exists\, addr\_p \in Unused \cdot \\
& \quad l.(\overline{\Delta rep} \ \wedge\ L' = L[1;\ ..\#L]) \\
& \quad \wedge\ \underline{(p.L)'} = [l.L0] \ \wedge\ \underline{(p.rep)' \subseteq l.rep \cup Unused} \\
& \quad ;\ \underline{\overline{\Delta}(l.rep \cup p.rep)} \ \wedge\ l.L' = l.L^\frown p.L \ \wedge\ (l.inv)' \\
& \hfill \text{Theorem 3.6 for } g = l.rep \cup p.rep \\
\Rightarrow & l.(\overline{\Delta rep} \ \wedge\ self \in List \ \wedge\ L \neq []) \\
& \wedge\ \exists\, addr\_p \in Unused \cdot \\
& \quad l.(L' = L[1;\ ..\#L]) \ \wedge\ (p.L)' = [l.L0]\ ;\ l.L' = l.L^\frown p.L \ \wedge\ \underline{(l.inv)'} \\
& \hfill \text{extract } (l.inv)' \text{ from quantification} \\
= & l.(\overline{\Delta rep} \ \wedge\ inv' \ \wedge\ self \in List \ \wedge\ L \neq []) \\
& \wedge\ \exists\, addr\_p \in Unused \cdot\ l.(L' = L[1;\ ..\#L]) \ \wedge\ \underline{(p.L)' = [l.L0]}\ ;\ \underline{l.L' = l.L^\frown p.L} \\
& \hfill \text{Theorem 3.4} \\
\Rightarrow & l.(\overline{\Delta rep} \ \wedge\ L' = L[1;\ ..\#L]^\frown L[0] \ \wedge\ inv' \ \wedge\ rep' \subseteq rep \cup Unused)
\end{array}
$$

4.3. 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** $insert(x) \cdot\ x \in \mathbb{Z} \ \Rightarrow\ S ::= S \cup \{x\}$
  **method** $paste(p) \cdot$
      $inv \ \wedge\ p \in Set \ \wedge\ p.inv \ \wedge\ disjoint[rep;\ p.rep]$
   $\Rightarrow \overline{\Delta}(rep \cup p.rep) \ \wedge\ S' = p.S \cup S \ \wedge\ inv'$
 **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** $contents$
  $inv \ \equiv\ contents \in List \ \wedge\ contents.inv \ \wedge\ addr\_contents \notin contents.rep$
  $init \ \equiv\ inv \ \wedge\ contents.init$
  $rep \ \ = \ \ \{addr\_contents\} \cup contents.rep$
  $S = \{x \in \mathbb{Z} \cdot \exists\, i \in \{0, ..\#(contents.L)\} \cdot contents.L\ i = x\}$

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

The correctness proof is very easy in this case and it is omitted.

4.4. Iterators

This example shows how the theory handles sharing and friend classes. We specify iterators in a module *IteratorSpec* which imports the *ListSpec* module from Sect. 4.1. 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. The class of iterators supports methods for attachment and traversal.

```
module $IteratorSpec$
  import $ListSpec$

  class $Iterator$
    prog attr $attl$
    spec attr $pos$
    $inv \;\Rightarrow\; (attl = null \vee (attl \in List \wedge attl.inv)) \;\wedge\; disjoint[rep;\; attl.rep]$
    $inv \;\Rightarrow\; rep$ frames $(attl, rep) \;\wedge\; (rep \cup attl.rep)$ frames $inv$
    $inv \wedge attl \neq null \;\Rightarrow\; pos \in \{0, .., \#attl.L\} \;\wedge\; (rep \cup attl.rep)$ frames $pos$
    $init \;\Rightarrow\; attl = null$

    method $attach(l) \cdot\; inv \wedge l \in List \wedge l.inv \;\Rightarrow\; \overline{\Delta} rep \wedge (inv \wedge pos = 0 \wedge attl = l)'$
    method $next() \cdot$
      $inv \wedge pos < \#attl.L \;\Rightarrow\; \overline{\Delta} rep \wedge pos' = pos + 1 \wedge attl' = attl \wedge inv'$
  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 $pos = min\{i \in \mathbb{N} \cdot\; attl.head.[next]^i = currentNode\}$
    $inv \;=\;\; (attl = null \vee (attl \in List \wedge attl.inv \wedge pos \in \{0, .., \#attl.L\}))$
            $\wedge\; disjoint[rep;\; attl.rep] \wedge rep \subseteq Used$
    $init \;=\; attl = currentNode = null$
    $rep \;=\; \{addr\_attl, addr\_currentNode\}$

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

4.5. Observers

This is an example of the Subjects and Observers pattern, which shows how the theory handles sharing without friendship. The specification of Subjects follows. For simplicity, we only use an integer specification variable *val* to represent the visible state of the object, which is being observed. We also omit subscription and notification of observers; it is going to be the responsibility of each observer to update itself accordingly.

```
module SubjectSpec
  class Subject
    spec var val
    inv  ⇒  val ∈ ℤ ∧ rep frames (rep, val, inv)
    method set(v) · v ∈ ℤ ⇒ val ::= v
  end class
end module
```

We do not want to implement *SubjectSpec*: we assume that another implementer has done that for us. We want to create various observer classes without touching that implementation. In what follows, we create a module which specifies such an observer class. It supports methods *attach* to attach itself to a subject and *update* to update its state. It also supports a boolean specification attribute *updated* that is true if and only if the internal representation of the observer (which is not shown in its specification) is in synch with the value of the subject.

```
module ObserverSpec
  import SubjectSpec

  class Observer
    spec attr updated
    prog attr subject
    inv  ⇒  updated ∈ 𝔹 ∧ subject ∈ Subject ∪ {null} ∧ rep frames (rep, subject, inv)
```

The interesting part here is the framing of *updated*, which is not achieved by **frames** alone. The specification attribute *update* also depends on the observed subject. Since however, we don't have access to the private attributes of the subject, we must content ourselves with the public attributes, generalizing **frames** as follows:

$$inv \wedge subject \neq null \wedge subject.inv \wedge \Xi rep \wedge subject.val' = subject.val \Rightarrow updated' = updated$$

The methods of *Observer* are specified as follows:

```
    method attach(s)·
      s ∈ Subject ∧ inv  ⇒  Δ̄rep ∧ subject′ = s ∧ inv′ ∧ updated′
    method update()·
      inv  ⇒  Δ̄rep ∧ subject′ = subject ∧ inv′ ∧ updated′
  end class
end module
```

5. Other work

5.1. Work based on Dynamic Frames

Ghost state in place of model state. The main work that followed [Kas06a] focused on the automation of proof with Dynamic Frames. The problem is hard to deal with in its general case, since some examples involve higher-order logic and induction-based proofs (see for example Theorem 4.1). *Region logic* [BBN08, BNR08] and the Dafny automatic verifier [Lei08] replaced our specification attributes with *ghost attributes*, i.e. with attributes that are assigned to by the programmer, instead of deriving their value automatically from definitions, as is done here. While this circumvents the problem and permits automated theorem proving, it might be cumbersome and error-prone for the programmer.

A rival verifier for Dynamic Frames, that does not have this problem, but has not been demonstrated in such higher-order examples, is VeriCool [SJP08b]. VeriCool was created with a focus to support concurrency.

Implicit dynamic frames. Perhaps the most important development on Dynamic Frames research is the advent of *Implicit Dynamic Frames* [SJP08a, SJP09] and the respective automatic verifier VeriFast. The framework of Implicit Dynamic Frames achieves more concise specifications, as framing specifications of a method are inferred from the rest of its specification. The treatment is influenced by Separation Logic, to the point one can say that Implicit Dynamic Frames are a unification of Dynamic Frames with Separation Logic.

A difference between the original version of Dynamic Frames and Implicit Dynamic Frames lies in the proof laws: Implicit Dynamic Frames introduce a *swinging pivots axiom* that facilitates proofs in the presence of sequential composition. This idea inspired the notion of *strong frame* which we introduced in Sect. 3.6 and the corresponding accumulation law, Theorem 3.6. Strong frames appear here for the first time: the original version of Dynamic Frames [Kas06b] used only the reasoning laws of Sect. 3.5. Using strong frames achieves more concise specifications and simpler proofs. A strong frame is roughly equivalent to the *required access set* of Implicit Dynamic Frames.

5.2. Other frameworks

Older approaches. Leino and Nelson's work [LN02] 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 [DLN98], rules out the implementation for *paste* in Sect. 4.1. In a variant [LPHZ02], the authors use *data groups* [Lei98] 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 [LN02] are enforced.

The *Universes* type system [Mül02] is a much simpler and more uniform approach to the problem, also adopted by the JML language [MPHL03, DM05]. It too imposes restrictions that have to do with objects travelling through encapsulation boundaries. Our implementation for *List* is possible in [Mül02], although somewhat awkwardly, by declaring the node objects "peers" to their containing list object. Our implementation of the *paste* method for *Set* in Sect. 4.3 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 [BDF+04, LM04, BN04, LM06] used in Spec# [BLS04]. 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. The restriction remains in the treatments of friends and sharing of [BN04]. 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 [LM04].

The Dynamic Frames theory imposes no such restriction and therefore it is more flexible than Boogie. The Iterator example of Sect. 4.4 and the Observer example of Sect. 4.5 show examples of sharing. Sharing with or without friendship is supported: the class *Iterator* is a *friend* of the class *List*, while observers are not friends to the class *Subject*. In either case, the creation of a new sharing class will not affect the specification and the proof of the shared class.

Separation logic. The development of Separation Logic [HRY01, Rey02] attacks the framing problem from a different 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.

O'Hearn et al.'s work [HYR04] 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 [PB05]. 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 [PB05] solve the problem by providing a much more complete treatment based on their introduction of *abstract predicates* (very similar to our notion of invariant). Their 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.

The use of the Frame Rule is inappropriate in the case of sharing, like the examples of Sects. 4.4 and 4.5. For example, 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, even though the representation regions of the two iterators (their *rep* frame) are disjoint. 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.

The difference between the two approaches seems to be that the frames of operations and specification attributes are made explicit in Dynamic Frames, while they are implicit in Separation Logic. In Implicit Dynamic Frames this issue appears and it is dealt with using an extra predicate *untouched*, to guarantee that some locations (in the present example, the representation region of the attached list) remain untouched even though they are part of the frame of a method.

Unified theories of programming. The present paper is very close in spirit to the relational frameworks of Unified Theories of Programming (UTP) [HH98]. However, the treatment of pointers in UTP-influenced theories has departed from the simple modelling of "pointers as locations" which undelies Dynamic Frames. In particular, works like [HCW08, SG08], following ideas from [Bro86], attempt a more abstract view of pointers, in which heaps are represented by a function V that can take arbitrarily long lists of references and returns values, as well as an equivalence relation S that describes which of these lists of references point to the same address. The details of the models have been well worked out. We are looking forward to see how far these abstractions can go, by applying them to some interesting examples.

5.3. Future work

The theory of Dynamic Frames is a very flexible way to deal with the framing problem in the presence of references and encapsulation. The research on Dynamic Frames has so far focused on building automated provers on it or its variants. The difficulty rises basically on the higher-order axioms that are required to deal with some frequent patterns, such as the linked list of Sect. 4.1. Another methodological problem of the theory is the verbosity of its specifications, which is however effectively dealt with in the Implicit Dynamic Frames variant.

Apart from improvements to the theory itself, it is interesting to see how the theory of Dynamic Frames handles concurrency, a field in which heap separation as well as sharing are of vital importance. An extension of Implicit Dynamic Frames with Fractional Permissions [Boy03], presented in [LM09] is an interesting approach and can serve as a starting point.

6. Conclusion

This paper is an extension of [Kas06a]. We have introduced the full theory of Dynamic Frames together with its reasoning laws, some interesting examples and a report on the most important work that was based on it. The Dynamic Frames theory remains one of the most simple and flexible formalisms to attack the frame problem in the presence of encapsulation and pointers, while at the same time it has been influential on most subsequent research on the subject. Current research on the theory focuses on automating the verification of programs specified with Dynamic Frames, with various proposals already implemented.

Acknowledgments

I wish to thank E. C. R. Hehner, who has been my academic supervisor when most of this research was conducted. I also wish to thank the anonymous reviewers of the paper for their very helpful feedback.

References

[BBN08] Banerjee A, Barnett M, Naumann D (2008) Boogie meets regions: a verification experience report. Technical Report MSR-TR-2008-79, Microsoft Research

[BDF$^+$04] Barnett M, DeLine R, Fähndrich M, Leino KRM, Schulte W (2004) Verification of object oriented programs with invariants. J Object Technol 3(6). http://www.jot.fm/issues/issue_2004_06/article2/article2.pdf

[BLS04] Barnett M, Leino KRM, Schulte W (2004) The Spec# specification language: an overview. In: Barthe G, Burdy L, Huisman M, Lanet J-L, Muntean T (eds) CASSIS'04. Lecture notes in computer science, vol 3362. Springer, Berlin, pp 49–69

[BN04] Barnett M, Naumann D (2004) Friends need a bit more: maintaining invariants over shared state. In: Kozen D (ed) MPC'04. Lecture notes in computer science, vol 3125. Springer, Berlin, pp 54–84

[BNR08] Banerjee A, Naumann D, Rosenberg S (2008) Regional logic for local reasoning about global invariants. In: ECOOP'08. Lecture notes in computer science, vol 5142. Springer, Berlin, pp 387–411

[Boy03] Boyland J (2003) Checking interference with fractional permissions. In: Cousot R (ed) SA'03. Lecture notes in computer science, vol 2694. Springer, Berlin, pp 55–72

[Bro86] Brookes SD (1986) A fully abstract semantics and a proof system for an Algol-like language with sharing. In: Melton A (ed) MFPS'85. Lecture notes in computer science, vol 239. Springer, Berlin, pp 59–100

[DLN98] Detlefs DL, Leino KRM, Nelson G (1998) Wrestling with rep-exposure. Technical Report 156, DEC-SRC

[DM05] Dietl W, Müller P (2005) Universes: lightweight ownership for JML. J Object Technol 4(8):5–32

[HCW08] Harwood W, Cavalcanti A, Woodcock J (2008) A theory of pointers for the UTP. In: ICTAC'08. Lecture notes in computer science, vol 5160. Springer, Berlin, pp 141–155

[Heh93] Hehner ECR (1993) A Practical Theory of Programming. Texts and Monographs in Computer Science. Springer, Berlin

[HH98] Hoare CAR, He J (1998) Unifying Theories of Programming. Prentice Hall Series in Computer Science. Prentice Hall, New Jersey

[HRY01] Hearn PO', Reynolds J, Yang H (2001) Local reasoning about programs that alter data structures. In: CSL'01. Lecture notes in computer science, vol 2142. Springer, Berlin, pp 1–19

[HYR04] Hearn PO', Yang H, Reynolds J (2004) Separation and information hiding. In: POPL'04, pp 268–280

[Kas06a] Kassios IT (2006) Dynamic frames: support for framing, dependencies and sharing without restrictions. In: Misra J, Nipkow T, Sekerinski E (eds) FM'06. Lecture notes in computer science, vol 4085. Springer, Berlin, pp 268–283

[Kas06b] Kassios IT (2006) A theory of object oriented refinement. PhD thesis, University of Toronto

[Lei98] Leino KRM (1998) Data groups: specifying the modification of extended state. In: OOPSLA'98. ACM, New York, pp 144–153

[Lei08] Leino KRM (2008) Specification and verification of object-oriented software. In: Marktoberdorf International Summer School 2008, Lecture Notes

[LM04] Leino KRM, Müller P (2004) Object invariants in dynamic contexts. In: Odersky M (ed) ECOOP'04. Lecture notes in computer science, vol 3086. Springer, Berlin, pp 491–516

[LM06] Leino KRM, Müller P (2006) A verification methodology for model fields. In: Sestoft P (ed) ESOP'06. Lecture notes in computer science, vol 3924. Springer, Berlin, pp 115–130

[LM09] Leino KRM, Müller P (2009) A basis for verifying multi-threaded programs. In: Castagna G (ed) ESOP'09. Lecture notes in computer science, vol 5502. Springer, Berlin, pp 378–393

[LN02] Leino KRM, Nelson G (2002) Data abstraction and information hiding. ACM Trans Program Lang Syst 24(5):491–553

[LPHZ02] Leino KRM, Poetzsch-Heffter A, Zhou Y (2002) Using data groups to specify and check side effects. In: PLDI'02. ACM, New York, pp 246–257

[MH69] McCarthy J, Hayes PJ (1969) Some philosophical problems from the standpoint of artificial intelligence. Mach Intell 4:463–502

[MPHL03] Müller P, Poetzsch-Heffter A, Leavens G (2003) Modular specification of frame properties in JML. Concurrency Comput Pract Experience 15:117–154

[Mül02] Müller P (2002) Modular Specification and Verification of Object-Oriented Programs. Lecture notes in computer science, vol 2262. Springer, Berlin

[PB05] Parkinson M, Bierman G (2005) Separation logic and abstraction. In: POPL'05, pp 247–258

[Rey02] Reynolds J (2002) Separation logic: a logic for shared mutable data structures. In: LICS'02. IEEE Computer Society, USA, pp 55–74

[SG08] Smith MA, Gibbons J (2008) Unifying theories of locations. In: Butterfield A (ed) UTP'08

[SJP08a] Smans J, Jacobs B, Piessens F (2008) Implicit dynamic frames. In: FTfJP'08

[SJP08b] Smans J, Jacobs B, Piessens F (2008) VeriCool: an automatic verifier for a concurrent object-oriented language. In: FMOODS '08. Lecture notes in computer science. Springer, Berlin, pp 220–239

[SJP09] Smans J, Jacobs B, Piessens F (2009) Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP'09

Received 5 August 2009

Accepted in revised form 15 March 2010 by J.C.P. Woodcock

Published online 15 April 2010