Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 117–125.

Separation logic style reasoning in a refinement based language[*]

Gergely Dévai, Zoltán Csörnyei

Eötvös Loránd University, Department of Programming Languages and Compilers
e-mail: deva@elte.hu, csz@inf.elte.hu

Abstract

Separation logic is an extension of classical logic to reason about programs that use pointers and dynamic memory management. It is known that separation logic can be expressed in classical logic, so its power lies in its reasoning style.

Refinement based programming starts with the formal description of the requirements concerning the program. This specification is then refined in several steps towards an implementation, which is correct by construction.

In this paper we show how to build proofs in a refinement based language using the style of separation logic. We transform special elements of separation logic back into classical logic in order to be able to handle them in the selected system. We also discuss our implementation of a well-known proof of separation logic.

1. Motivation

Formal methods in software development become more and more important, because the hidden errors in programs costs a lot: testing and bugfixing is a significant part of software projects' life cycle. It is quite promising that by developping verified programs most of these programming errors can be eliminated, but systems implementing formal methods are to be improved a lot to become widely used in industrial software development.

Both separation logic [14] and refinement based software development [12, 11] are techniques that can help. Separation logic is an elegant approach to reason about data structures dynamically created and modified. It is enough to consider the most popular object oriented languages that use objects created in run time, to understand the relevancy of this research area. Refinement based software development has certain advantages among formal methods: it can be highly integrated

[*]This work is supported by "Stiftung AKTION Österreich-Ungarn" (OMAA-ÖAU 66öu2) and "ELTE IKKK" (GVOP-3.2.2-2004-07-0005/3.0).

into the development process and it helps to detect (and prevent) design errors in an early stage of the development process [10].

In this paper we inspect how to use a refinement based language to implement proofs in the style of separation logic. In Section 2 we give a short presentation of separation logic from our point of view. Section 3 presents the selected proof language and system, while in Section 4 we integrate separation logic into this system. Some conclusions of a realistic example is presented in Section 5.

2. Separation logic

In classical *weakest precondition calculus* and *Hoare logic* [8] we exploit the hidden property of the underlying models that there is a one-to-one mapping between variable names and memory locations where the values are stored. For example to prove that from the precondition $y = z$ the instruction $x := 0$ results in the postcondition $x = 0 \wedge y = z$, we can compute the weakest precondition

$$wp(x := 0,\, x = 0 \wedge y = z) = (x = 0 \wedge y = z)[x/0] = (0 = 0 \wedge y = z) = (y = z),$$

or, if we want to use *forward reasoning*, we can use the axiom

$$\{True\}x := 0\{x = 0\}$$

and the *frame axiom*

$$\frac{\{Q\}x := 0\{R\}}{\{P \wedge Q\}x := 0\{P \wedge R\}},\ \text{if } P \text{ does not contain } x.$$

Each of these methods relies on the fact that the variable x can be syntactically distinguished from other variables, like y and z. Unfortunately these approaches fail if we use pointers in the program. Although the axiom

$$\{p \text{ points to a } \textit{valid} \text{ location}\}[p] := 0\{[p] = 0\}$$

remains true (if we can express somehow what a *valid location* is), both

$$\frac{\{Q\}[p] := 0\{R\}}{\{P \wedge Q\}[p] := 0\{P \wedge R\}},\ \text{if } P \text{ does not contain } [p]$$

and

$$wp([p] := 0,\, [p] = 0 \wedge [q] = z) = ([q] = z)$$

fail, because it is possible that the pointers p and q are pointing to the same location. Somehow we have to express that the pointers are referencing different parts of the heap memory, and separation logic shows an elegant way to do that.

Separation logic introduces the following predicates and connectives to describe the current state of the heap:

- *emp* means that the heap is empty,

- $p \mapsto v$ means that the heap contains the only value v pointed by the pointer p,
- $P * Q$ (*separating conjunction*) means that the heap can be split into two disjoint sub-heaps h_1 and h_2 such that P describes h_1 and Q describes h_2,
- $P - *Q$ (*separating implication*) means that if we extend the current heap with a disjoint part which satisfies P, then the extended heap satisfies Q.

Using these extensions of logic we can state that

$$\{\exists v.p \mapsto v\}[p] := 0\{p \mapsto 0\},$$

and, for example, by $(\exists v.p \mapsto v) * (q \mapsto z)$ we can express that p and q are pointing to different locations.

The *frame rule* of separation logic is the following:

$$\frac{\{Q\}s\{R\}}{\{P * Q\}s\{P * R\}}$$

It expresses that if a program operates in a part of the heap, it preserves the state of other heap parts. Using these rules, we can solve our previous problem and prove the following:

$$\{(\exists v.p \mapsto v) * q \mapsto z\}[p] := 0\{p \mapsto 0 * q \mapsto z\}.$$

3. The refinement based language *LaCert*

In this paper we mention only the language features required to understand Section 4, which explaines the enbedding of separation logic into *LaCert.* For more details the reader is referred to [6] and [1].

With *LaCert* one can produce *verified* program code. Unlike *testing* or *extended static checking* [5] a program accepted by *LaCert*'s compiler satisfies all the requirements of its specification. (Of course, just like in case of all other formal methods, the axioms describing mathematical functions and the temporal axioms of the target language instructions must be sound.)

LaCert aims to help in the development of *imperative programs.* Although *functional programming* is a promising field for formal methods, there are specifically imperative problems (like I/O management) that require special reasoning techniques [9]. The popularity of imperative languages is another argument for this design decision.

Unlike verification systems, where one writes program code first and then uses authomatic tools or a semi-authomatic *theorem proover* [3, 13, 15] to prove (or disprove) certain properties of the code, in *LaCert* one starts the development by the specification and reaches the implementation via its refinemets. This strategie has advantages, like *correctness by construction* and early recovery of errors. It is also used for example in the *B-method* [2], which is one of the few formal methods also applied in industry.

While in most refinement based systems one writes program code in the final refinement steps, in *LaCert* this is done automatically by a separate code generator module. Another point is the advanced *metaprogramming features* of this language. Often used proof parts can be parameterised and encapsulated in *templates.* These templates can later be called to use the proof fragment inside. Different kinds of templates exist to express axioms, temporal axioms, tactics and to help induction.

Specifications and their refinements consist of temporal properties, namely *progress-* and *safety properties.* These are built up by first order predicate logic formulas, whose variables are program variables and parameters. Safety properties are enclosed in [and] brackets, while the pre- and postconditions of a progress property is connected by the $\gg$ operator.

```
[y = z];
ip = K >> ip = L & x = 0;
```

These properties state that whenever the program execution reaches the label K, it has to reach L and then the variable x must be zero. The safety property ensures that during this progress the validity of $y = z$ is preserved.

Note the explicit usage of the *ip* (instruction pointer) variable and labels in the specification languge. This makes our progress properties more expressive compared to classical *Hoare logic.* (It is possible for example to specify non-terminating programs in this way.)

A progress property can be refined by two basic constructs: *sequence* and *case distinction.* The first one introduces intermediate steps, while the second one breaks a disjunction in the precondition into cases. The refining statements are to be refined further unless they are temporal axioms of instructions of the target language.

These axioms are defined in special templates. The calls of these templates (and the arguments of the calls) are collected by the compiler. The code generator (which is apart from the compiler) transformes these calls into the syntax of the target programming language. For example, in case of *C++* as the target language, the above specification would result in the following code fragment.

```
K: x = 0;
L:
```

4. Integration of separation logic into *LaCert*

In this section we present how to write specifications and proofs in the style of separation logic. As a result we can specify and implement verified programs that involve dynamic memory management.

As we wanted to integrate separation logic without modifying the compiler, we had to transform its extensions to the logic supported by *LaCert.*

4.1. Back to classical logic

The problem of representing separation logic in classical logic is a field currently under research. Approaches like [4] represent restricted versions of separation logic with the goal of obtaining formulas that can be managed efficiently by well-known automatic reasoning methods of classical logic. Our goal is different: we need *easily readable and compact* representation some features of separation logic. As we want to construct *forward reasoning style proofs*, we drop *separating implication* (which is mainly used to express the weakest precondition of memory allocation). As a second restriction, we will not merge the special connectives with classical logic ones. That is, we deal with formulas built of *emp*, $\mapsto$ and $*$ only. We will refer to these restricted formulas as *conjunctive heap formulas.*

In order to represent these formulas, we use the semantics of separation logic [16]: we let the heap appearing in the specification explicitly as a variable. To define the type *Heap* and to declare the variable, we write the following in the syntax of *LaCert*:

```
type( Heap );
variable( heap, Heap );
```

We implement the empty heap as a function without arguments returning a heap:

```
function( ”empty”, Heap );
```

The singleton heap consisting of a value of type $\#T$ and pointed by a pointer is represented by the $at : Pointer \times \#T \rightarrow Heap$ function. In the following generic declaration $\#T$ is a type parameter.

```
function( ”at”, Heap, Pointer, #T );
```

The key is the representation of the separating conjunction. Its semantics is usually given using a binary operator that combins two heaps and which is defined only if the combined heaps are disjoint. We overload the + operator of *LaCert* with the following declaration:

```
function( ”+”, Heap, Heap, Heap );
```

Using the following representation we map each conjunctive heap formula to a *LaCert* expression of type *Heap*:

$$\begin{aligned} Rep(emp) &= empty() \\ Rep(p \mapsto v) &= at(p, v) \\ Rep(P * Q) &= Rep(P) + Rep(Q) \end{aligned}$$

Using this representation, the *LaCert* fomula modelling the conjunctive heap formula P is

$$heap = Rep(P).$$

For example, the formula $(\exists v.p \mapsto v) * (q \mapsto z)$ from section 2 is transformed first to $\exists v.((p \mapsto v) * (q \mapsto z))$ using a rule of separation logic and then written in *LaCert* as follows:

```
exists(Integer @v, heap = at(p,@v) + at(q,z))
```

4.2. Specification of instructions related to the heap

Using the techniques presented in the previous sections we can give the specifications of the instructions related to the heap. Instructions are needed to *allocate* memory, to *set* and to *get the value* of a variable on the heap, and to *dispose* the variable.

Each instruction specification consists of a safety property and a progress property both of wich are enclosed in a template. To save space, we will not present the entire templates, nor the safety properties. These can be found in the technical report [7].

4.2.1. Allocation

Allocation of a new variable on the heap extends the current heap with a disjoint part, and the address of the newly allocated value is stored in a pointer. The parameters of the template containing the specification of this instruction are the following: $\#hVal$ is an expression describing the old value of the heap, $\#p$ is the pointer, $\#before$ and $\#after$ are the labels of this and the next instruction respectively. The progress property of this instruction:

```
ip = #before & heap = #hVal
>> ip = #after
& exists( Integer @v, heap = at(#p,@v) + #hVal );
```

The progress property states that whenever the execution of the program is at the label of the instruction ($\#before$) and the heap has the value $\#hVal$, the program steps to the following label ($\#after$) and the heap is extended by a disjoint part consisting of some value ($@v$) and pointed by pointer $\#p$.

This instruction modifies only the instruction pointer (ip), the pointer that we assign the new address to ($\#p$), and the heap. Any expression independent of these variables is a safety property of the instruction.

4.2.2. Writing the value of a variable on the heap

Modifying the value of a variable pointed by a pointer also modifies the heap, but the pointer remains unchanged. The progress property of this instruction can be written as follows.

```
ip = #before
& exists( Integer @v, heap = at(#p,@v) + #hVal )
>> ip = #after & heap = at(#p,#val) + #hVal;
```

The structure of the property is similar to the one's explained in the previous section, so instead of going into the details, let us compare it to the similar axiom of separation logic:

$$\{\exists v.(p \mapsto v)\}[p] := val\{p \mapsto val\}$$

This axiom is just the so called *footprint* of the instruction, i.e. it only involves the relevant part of the heap. If our heap is larger, we have to use the *frame rule of separation logic* that we mentioned in Section 2. This frame rule appears explicitly in the *LaCert* specification: the unmodified part of the heap is denoted by the $\#hVal$ template parameter.

4.2.3. Reading the value of a variable on the heap

Among our four instructions discussed here, this is the only one which does not modify the heap. That is why it does not appear in the postcondition. The compiler can use the safety property axiom to infer that an assertion describing the state of the heap is preserved by this instruction.

```
ip = #before & heap = at(p,#val) + #hVal
>> ip = #after & #var = #val;
```

4.2.4. Disposing a variable on the heap

Deallocation is the inverse instruction of allocation and this fact is well reflected by the specification: pre- and postconditions are swapped. There is one notable difference: deallocation does not affect the value of the pointer.

```
ip = #before
& exists( Integer @v, heap = at(p,@v) + #hVal )
>> ip = #after & heap = #hVal;
```

5. Listreversal example

In-place reversal of a linked list is a classical example of separation logic. At first, we have to express that a pointer points to the beginning of a linked list. In the separation logic solution this is expressed using an inductively defined predicate in terms of the special predicates and connectives. In the spirit of the representation described in Section 4.1, we use the following heap-valued function instead.

```
list( p, s )
```

This describes a heap-part containing a linked list with the sequence s and pointed by the pointer p. Using this function and the *reverse* function defined on sequences, the specification of the listreversal is the following:

```
ip = ListRevStart & heap = list( p, s )
>> ip = ListRevStop & heap = list( q, reverse(s) )
```

The current version of the refinements are rather long due to the currently limited reasoning capabilities of *LaCert*. The resulting *C++* program consists of 10 instructions, while the depth of the proof tree of the refinements in *LaCert* is 5 and it involves 74 basic proof steps. Most of these steps are quite simple, therefore we expect that after further development of the system this number will decrease to 16–20, which is more acceptable.

6. Summary

In this paper we pointed out that the reasoning style of separation logic can also be used in systems based on classical logic. For this embedding, we have chosen a refinement based system because it helps also during the development process not only in posterior error-recovery.

We transformed the needed extensions of separation logic back to classical logic based on its semantics. The progress properties of the basic heap management instructions were presented. It was also observed that the so-called frame rule of separation logic is implicitly present in these progress properties.

As the result of this work it is now possible to specify and implement verified programs that use dynamic memory management. Investigating a not completely trivial example we concluded that the automatic reasoning capabilities of the system are to be developed further to be able to decrease the size of the user-written proof.

References

[1] Home of LaCert `http://deva.web.elte.hu/LaCert`

[2] Abrial, J.-R., The B-book: assigning programs to meanings, *Cambridge University Press*, New York, NY, USA, (1996).

[3] Bertot, Y., Castéran, P., Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, *Texts in Theoretical Computer Science, Springer Verlag*, (2004).

[4] Calcagno, C., Gardner, P., Hague, M., From separation logic to first-order logic, In *FOSSACS*, (2005).

[5] Cok, D. R., Kiniry, J. R., ESC/Java2: Uniting ESC/Java and JML, In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Springer, vol. 3362 (2005), 108–128.

[6] Dévai, G., Programming language elements for correctness proofs, *Acta Cybernetica* (accepted for publication), (2007).

[7] Dévai, G., Specification of heap related instructions in LaCert, *Technical report*, (2007).

[8] Hoare, C. A. R., An axiomatic basis for computer programming, *Commun. ACM*, 12(10) (1969), 576–580.

[9] Horváth, Z., Kozsik, T., Tejfel, M., Extending the Sparkle core language with object abstraction, *Acta Cybernetica*, 17 (2005), 419–445.

[10] McDonald, J., Anton, J., Specware - producing software correct by construction, (2001).

[11] Morgan, C., Programming from specifications, *Prentice Hall International (UK) Ltd.*, second edition, (1994).

[12] Morris, J. M., A theoretical basis for stepwise refinement and the programming calculus, *Sci. Comput. Program.*, 9(3) (1987), 287–306.

[13] Nipkow, T., Paulson, L. C., Wenzel, M., Isabelle/HOL – A Proof Assistant for Higher-Order Logic, volume 2283 of *LNCS*, Springer-Verlag, (2002).

[14] O'Hearn, P., Reynolds, J., Yang, H., Local reasoning about programs that alter data structures, *Lecture Notes in Computer Science*, 2142, (2001).

[15] Pásztor Varga, K., Várterész, M., Usability of some theorem proving systems, *PU.M.A.*, 15(2-3) (2004), 273–284.

[16] Yang, H., O'Hearn, P., A semantic basis for local reasoning, In *Foundations of Software Science and Computation Structure*, (2002), 402–416.