A deeper understanding of the deep frame axiom (extended abstract)

Nathaniel Charlton and Bernhard Reus

University of Sussex, Brighton

Abstract. Separation logic [7] is well known to provide a local reasoning principle for local store, the frame rule. Local reasoning is extended to programs with modules and higher-order functions respectively by the hypothetical frame rule [4] and higher-order frame rule [2]. This means, for instance, that the private state of a library module can be hidden, so that clients do not need to know or worry about the private state; this leads to modular proofs.

Higher-order functions can be expressed in low level languages using code pointers and recursion through the store, as in [1, 8]. However, local reasoning for such languages is surprisingly tricky: in [8], a version of the (higher-order) deep frame rule is observed to hold only as a *rule* and not as an *axiom*. This means one can only apply the rule to the top level triple under consideration, but not to nested triples appearing inside assertions. Assuming the axiom version allows one to "launder" specifications [6, 8] and thus derive memory-safety of programs that actually crash. On the other hand, it appears that without a reasoning principle resembling the axiom version, there are safe programs whose correctness cannot be shown.

In this paper we investigate and analyse the technique of "laundering" in more detail and propose a remedy, an idiom for writing specifications (Hoare triples) that "promise" that applications of the deep frame axiom are sound. This is done with the help of second order logic (quantification over assertions). We demonstrate the utility of this technique for proving real programs by means of an example: a generic memoiser for recursive functions.

1 Introduction and motivation

In low level languages one uses code pointers to simulate the effect of higher-order procedures, with the additional complication that updating code pointers can lead to recursion through the store. In this paper we study local reasoning for one such language with immutable ML-like variables and value parameters; see [3] for the programming language and logic we work with.

To illustrate the issues mentioned in the abstract suppose there is a procedure C that, maybe among other things, frees a specified cell on the heap. The procedure has two arguments: a, the address of the cell and f, the pointer to a procedure that performs concrete low level deallocation, maybe with some additional administration and bookkeeping. Abstracting the concrete deallocation in

parameter f allows C to be reusable. According to the above, the specification for such a procedure C is as follows:

$$\forall f, a. \left\{ \begin{array}{l} f \mapsto \mathit{DeleteArg}(\_) \\ \star\ a \mapsto \_ \end{array} \right\}\ C(f,a)\ \ \{\, f \mapsto \mathit{DeleteArg}(\_)\,\} \tag{1}$$

where

$$\mathit{DeleteArg}(P) := \forall x.\, \{x \mapsto \_\}\, P(x)\, \{\mathsf{emp}\}$$

Now suppose C uses f that contains code which also increments a counter whenever it is used to free a cell; is it safe to use such code with C? Formally, does (1) imply the following?

$$\forall f, a, \mathit{cnt}.$$
$$\left\{ \begin{array}{l} f \mapsto \mathit{DeleteArg}(\_) \otimes \mathit{cnt} \mapsto \_ \\ \star\ a \mapsto \_ \\ \star\ \mathit{cnt} \mapsto \_ \end{array} \right\}\ C(f,a)\ \left\{ \begin{array}{l} f \mapsto \mathit{DeleteArg}(\_) \otimes \mathit{cnt} \mapsto \_ \\ \mathit{cnt} \mapsto \_ \end{array} \right\} \tag{2}$$

Here $\Phi \otimes \Theta$ adds the invariant Θ to the pre- and post-conditions of all triples in Φ (at all nesting levels, see e.g. [8]). Intuitively one might think this implication holds: C does not know about the counter cell at all, and the code in f leaves that cell in place; therefore, the counter cell will be in place throughout the execution of C. The implication does indeed hold if we consider the above triple on the "top level", due to what is called the *deep frame rule* (DFR) which has been shown sound in [8]:

$$\frac{\Phi}{\Phi \otimes \Theta}$$

If we interpret this implication in a stronger sense, namely as instance of the *deep frame axiom* (DFA)

$$\Phi \ \Rightarrow\ \Phi \otimes \Theta$$

then things are very different: it was discovered that this axiom is not sound for higher-order store if the invariant Θ contains code pointers (see [8]). Yet, if we store C on the heap and call it from the main program, the rule version is of no help, and the axiom version seems to be exactly what we need. To see this in more detail consider the program *Prog* in Figure 1 which calls the procedure C twice with different deallocation procedures f_1 and f_2, respectively. *Prog* uses variable c to store a pointer to such a procedure C_i (shortly we will give three concrete versions of this procedure). The pointer variables f_1 and f_2 point to procedures that are supposed to perform the low level deallocation as explained earlier; f_1 does some extra bookkeeping in the form of counting the deleted cells in *cnt*. Pointers a_1 and a_2 refer to the cells to be deleted. We consider three possibilities for the code C_i:

$$C_1 := \text{`}\lambda f, a.\ \mathsf{eval}\ [h](a)\ ;\ [h] := [f]\text{'}$$
$$C_2 := \text{`}\lambda f, a.\ \mathsf{eval}\ [f](a)\text{'}$$
$$C_3 := \text{`}\lambda f, a.\ \mathsf{eval}\ [h](a)\text{'}$$

```
let h = new 'λx. free x' in
let c = new C_i in
let cnt = new 0 in
let a_1 = new 0 in
let a_2 = new 0 in
let f_1 = new 'λx. free x ; [cnt] := [cnt] + 1' in
let f_2 = new 'λx. free x' in
  eval [c](f_1, a_1) ;
  free cnt ;
  eval [c](f_2, a_2)
```

Fig. 1. Main program *Prog*.

Now for which C_is does *Prog* run without memory fault?

If C_1 is used, the program faults: when the code in C_1 (stored in c) is run a second time, the code in cell h still needs to access the counter cnt, but this has been freed meanwhile. Yet, using the DFA, one can prove *Prog* safe, that is, prove that it satisfies $\{\mathsf{emp}\} \cdot \{\mathsf{true}\}$. Oddly, on the other hand, with C_2 and C_3 the program runs safely; yet without something like the DFA, we have no way to prove this modularly.

The problem with C_1 is that it misbehaves by privately copying the "outside" code in f into h, rather than running it. This gives rise to a phenomenon that Pottier called "laundering" [6]. When running *Prog*, more precisely the statement $\mathsf{eval}\ [c](f_1, a_1)$, the procedure f_1 for which one assumes

$$(f_1 \mapsto \mathit{DeleteArg}(\_) \otimes \mathit{cnt} \mapsto \_) \otimes h \mapsto \ldots$$

will be saved away in h for which in the postcondition only a specification

$$h \mapsto \mathit{DeleteArg}(\_) \otimes h \mapsto \ldots$$

is required; this does not mention cnt any longer. This means that f_1 has been laundered of the invariant $cnt \mapsto \_$.

This copying of code provided by the outside world into hidden state seems to be exactly the problem with the deep frame axiom. Procedures C_2, C_3 do not perform such copying. C_2 does not use hidden state at all. C_3 does use the cell h to store code, but outside code never flows into h.

In this paper we propose a sound replacement for the DFA, so that one can prove the program safe with C_2 and C_3, but not with C_1.

2 Laundering: how to prove a faulty program with the deep frame axiom

In this section we explain how one can use the DFA to prove that program $Prog$ with C_1 is safe, when in fact it crashes. We define H, the invariant describing the cell h as a recursive assertion

$$H \quad \Leftrightarrow \quad h \mapsto DeleteArg(\_) \otimes H$$

the existence of which follows from e.g. [8]. We can then prove that C_1 satisfies (1) in the presence of H, that is,

$$\left(\forall f, a. \left\{ \begin{array}{l} f \mapsto DeleteArg(\_) \\ \star\ a \mapsto \_ \end{array} \right\} \ C_1(f,a) \ \left\{ f \mapsto DeleteArg(\_) \right\} \right) \otimes H$$

One then applies the DFR to hide H and thus the cell h[1]; it remains to prove

$$\left\{ c \mapsto \forall f, a. \left\{ \begin{array}{l} f \mapsto DeleteArg(\_) \\ \star\ a \mapsto \_ \end{array} \right\} \ \_\,(f,a) \ \left\{ f \mapsto DeleteArg(\_) \right\} \right\}$$

$$\begin{array}{l} \mathsf{let}\ cnt = \mathsf{new}\ 0\ \mathsf{in} \\ \mathsf{let}\ a_1 = \mathsf{new}\ 0\ \mathsf{in} \\ \mathsf{let}\ a_2 = \mathsf{new}\ 0\ \mathsf{in} \\ \mathsf{let}\ f_1 = \mathsf{new}\ \text{`}\lambda x.\ \mathsf{free}\ x\ ;\ [cnt] := [cnt] + 1\text{'}\ \mathsf{in} \\ \mathsf{let}\ f_2 = \mathsf{new}\ \text{`}\lambda x.\ \mathsf{free}\ x\text{'}\ \mathsf{in} \\ \quad \mathsf{eval}\ [c](f_1, a_1)\ ; \\ \quad \mathsf{free}\ cnt\ ; \\ \quad \mathsf{eval}\ [c](f_2, a_2) \end{array}$$

$$\left\{ \mathsf{true} \right\}$$

Here the DFA is needed to reason about the first eval statement for which we need to prove the triple

$$\left\{ \begin{array}{l} c \mapsto \forall f, a. \left\{ \begin{array}{l} f \mapsto DeleteArg(\_) \\ \star\ a \mapsto \_ \end{array} \right\} \ \_\,(f,a) \ \left\{ f \mapsto DeleteArg(\_) \right\} \\ *\ a_1 \mapsto \_ *\ a_2 \mapsto \_ \ * cnt \mapsto \_ \\ *\ f_1 \mapsto DeleteArg(\_) \otimes (cnt \mapsto \_) \ *\ f_2 \mapsto DeleteArg(\_) \end{array} \right\}$$

$$\mathsf{eval}\ [c](f_1, a_1)$$

$$\left\{ \begin{array}{l} c \mapsto \forall f, a. \left\{ \begin{array}{l} f \mapsto DeleteArg(\_) \\ \star\ a \mapsto \_ \end{array} \right\} \ \_\,(f,a) \ \left\{ f \mapsto DeleteArg(\_) \right\} \\ *\ a_2 \mapsto \_ *\ cnt \mapsto \_ \\ *\ f_1 \mapsto DeleteArg(\_) \otimes (cnt \mapsto \_) \ *\ f_2 \mapsto DeleteArg(\_) \end{array} \right\}$$

[1] Note that once we have applied the DFR, in the code for C_1 we must assume the code in f has specification $DeleteArg(\_) \otimes H$ rather than $DeleteArg(\_)$; in Pottier's terminology [5] the DFR is "paranoid" and (for good reason) assumes that functions obtained from "outside" might also depend on the added invariant H. Thus, the code in h must also be specified with $\otimes H$ added; this explains the need for a recursively defined predicate.

Note that, informally, we need to perform deep framing on the nested triple for c, and not at the top level, so the DFR cannot be used.

3 A remedy: a specification idiom to support deep framing

We have seen that not all commands behave in a way which admits the deep frame axiom; but some commands do. Therefore, whether or not a particular command C supports the DFA is a matter that must be agreed in the "contract" between C and its clients. We thus suggest a specification idiom which allows a command C to promise to its clients to behave in a way which supports the DFA; in particular, such a promise means that C cannot copy outside code to its hidden cells, as in the laundering example.

Concretely, we specify a command with

$$\forall X.\, \forall \boldsymbol{x}.\, \{P\}\ \_\ (\boldsymbol{p})\, \{Q\} \otimes X$$

to say, intuitively, that the command behaves as

$$\forall \boldsymbol{x}.\, \{P\}\ \_\ (\boldsymbol{p})\, \{Q\}$$

and also admits application of the DFA. Here we are using a second order logic, with the variable X ranging over assertions. Then the DFA is simulated simply by

$$\forall X.(\Phi \otimes X) \quad \Rightarrow \quad \forall X.((\Phi \otimes \Theta) \otimes X)$$

which can be derived from the usual $\forall$-instantiation axiom

$$\forall X.\Phi \quad \Rightarrow \quad \Phi[X \backslash \Theta]$$

3.1 Observation 1: We can now prove our program with C_2

Using our specification idiom, we can show that C_2 supports the DFA. We need to prove:

$$\left(\forall X.\, \forall f, a.\, \left\{ \begin{array}{l} f \mapsto \mathit{DeleteArg}(\_) \\ \star\ a \mapsto \_ \end{array} \right\}\ C_2(f,a)\ \ \left\{ f \mapsto \mathit{DeleteArg}(\_) \right\} \otimes X\right) \otimes H$$

But C_2 makes no use of h, so we simply show

$$\forall f, a.\, \left\{ \begin{array}{l} f \mapsto \mathit{DeleteArg}(\_) \\ \star\ a \mapsto \_ \end{array} \right\}\ C_2(f,a)\ \ \left\{ f \mapsto \mathit{DeleteArg}(\_) \right\}$$

and then use the DFR to successively add invariants X and H. This works in general: for code that makes no use of hidden state, the $\forall X. \cdots \otimes X$ to support the DFA comes for free.

3.2 Observation 2: We can now prove our program with C_3

C_3 does use hidden state, so we cannot use the DFR to get the $\forall X. \cdots \otimes X$ for free. (One can certainly use the DFR to add an arbitrary X as an invariant, but this gives $\otimes X$ outside of $\otimes H$, and not inside it as we need.) However, we can prove

$$\left(\forall X. \forall f, a. \left\{ \begin{array}{l} f \mapsto \mathit{DeleteArg}(\_) \\ \star\ a \mapsto \_ \end{array} \right\}\ C_3(f,a)\ \ \left\{ f \mapsto \mathit{DeleteArg}(\_) \right\} \otimes X \right) \otimes H$$

"on foot" by using distribution axioms for $\otimes$.

3.3 Observation 3: We cannot prove our program with (faulty) C_1

We cannot prove

$$\left(\forall X. \forall f, a. \left\{ \begin{array}{l} f \mapsto \mathit{DeleteArg}(\_) \\ \star\ a \mapsto \_ \end{array} \right\}\ C_1(f,a)\ \ \left\{ f \mapsto \mathit{DeleteArg}(\_) \right\} \otimes X \right) \otimes H$$

Because C_1 uses H, it is not possible to use the DFR to add the $\forall X. \cdots \otimes X$. But we cannot prove this "on foot" either: one gets stuck needing to show

$$\left\{ \begin{array}{l} f \mapsto (\mathit{DeleteArg}(\_) \otimes X) \otimes H \\ \star\ a \mapsto \_ \\ \star\ (X \otimes H) \\ \star\ H \end{array} \right\}\ C_1(f,a)\ \left\{ \begin{array}{l} f \mapsto (\mathit{DeleteArg}(\_) \otimes X) \otimes H \\ *\ (X \otimes H) \\ \star\ H \end{array} \right\}$$

The reason is that the statement $[h] := [f]$, which copies the "outside" procedure in f into the cell h, does not respect the invariant H: the procedure in f has specification

$$\forall x. \{x \mapsto \_\}\ \_\ (x)\ \{\mathsf{emp}\} \otimes X \otimes H$$

which depends on an extra unknown invariant X, whereas to maintain H we would need the specification

$$\forall x. \{x \mapsto \_\}\ \_\ (x)\ \{\mathsf{emp}\} \otimes H$$

4 Application: proving safety of generic memoisation for recursive functions

We have used our specification idiom to prove safety of a generic memoiser for recursive functions, as we now describe. Our memoiser works with recursively defined functions $f : \mathbb{Z} \to \mathbb{Z}$ such as the factorial function, which we implement as follows.

$$\begin{array}{l} \mathit{FACT} := \\ \quad \lambda\ n, \mathit{res}\ . \\ \qquad \mathsf{if}\ n = 0\ \mathsf{then}\ [\mathit{res}] := 1 \\ \qquad \mathsf{else} \\ \qquad\quad \mathsf{let}\ a = [A]\ \mathsf{in}\ \mathsf{eval}\ [a](n-1, \mathit{res})\ ; \\ \qquad\quad \mathsf{let}\ m = [\mathit{res}]\ \mathsf{in}\ [\mathit{res}] := m \times n \end{array}$$

Note that we do not use an explicit fixpoint operator since in the low level language we work with all recursion is through the higher order heap. This code makes its recursive call through the pointer found in cell A. Thus to use a function like $FACT$, one writes it into a heap cell f, and then sets $[A] := f$ to set up the recursion, as in:

$$\begin{array}{l}\mathsf{let}\ A = \mathsf{new}\ 0\ \mathsf{in}\\ \mathsf{let}\ f = \mathsf{new}\ FACT\ \mathsf{in}\\ \mathsf{let}\ \mathit{result} = \mathsf{new}\ 0\ \mathsf{in}\\ \quad [A] := f\ ;\\ \quad \mathsf{eval}[f](3, \mathit{result})\end{array}$$

One can specify such a function with

$$\mathit{RecFunc}(F) \quad := \quad \forall X.\, \forall n, r, a.\, \{\Psi(a) \star r \mapsto \_\}\, F(n, r)\, \{\Psi(a) \star r \mapsto \_\} \otimes X$$

where $\Psi(a)$ is a recursively defined predicate with argument as in [3]:

$$\Psi(a) \quad := \quad \left(\begin{array}{c}\qquad\qquad\qquad\qquad \left\{\, R(a) \star r \mapsto \_ \,\right\} \\ \mu\, R(a)\, .\, A \mapsto a\ \star\ a \mapsto \forall n, r, a. \qquad \_\ (n, r) \\ \qquad\qquad\qquad\qquad \left\{\, R(a) \star r \mapsto \_ \,\right\}\end{array}\right)(a)$$

Here the pre- and post-condition of the specification $\mathit{RecFunc}(F)$ describe *one function* on the heap, used for the recursive call, which will in fact be the same function F again.

Now, by making the pointer in A point to a different piece of code, one can "trap" the *internal* recursive calls made by functions such as factorial. The code in Figure 2 uses this mechanism to implement a memoiser which also memoises internal calls, so that for example evaluating 10! will make use of a cached result for 9!. This memoiser is generic in that it works with any similar recursive function $\mathbb{Z} \to \mathbb{Z}$. To specify functions for use with the memoiser, one can write

$$\mathit{RecFunc}_{mem}(F) \quad := \quad \forall n, r, a.\, \{\Psi_{mem}(a) \star r \mapsto \_\}\, F(n, r)\, \{\Psi_{mem}(a) \star r \mapsto \_\}$$

where

$$\begin{array}{l}\Psi_{mem}(a) := \\ \left(\begin{array}{l}\mu\, R(a)\, . \\ \quad A \mapsto a \\ \star\ f \mapsto \forall n, r, a.\ \{R(a) \star r \mapsto \_\}\ \_\ (n, r)\ \{R(a) \star r \mapsto \_\} \\ \star\ a \mapsto \forall n, r, a.\ \{R(a) \star r \mapsto \_\}\ \_\ (n, r)\ \{R(a) \star r \mapsto \_\}\end{array}\right)(a)\end{array}$$

This time, the pre- and post-condition of the specification $\mathit{RecFunc}_{mem}(F)$ describe *two functions* on the heap in cells f and a: one will be the function F again, and the other will be the memoiser code.

```
let A = new 0 in
let f = new FACT in
let table = new {} in
let mem = new
  ‘λ n, res .
    let t = [table] in
      if n ∈ dom(t) then
        [res] := t(n)
      else
        (eval [f](n, res) ;
        let x = [res] in
          [table] := t ⊕ {n ↦ x})
  ,
in
let result = new 0 in
  [A] := mem ;
    eval [f](9, result) ;
    eval [f](10, result)
```

Fig. 2. Code for a generic memoiser for recursively defined functions $\mathbb{Z} \to \mathbb{Z}$.

For reasons of modularity, we do not want to have to prove $\mathit{RecFunc}_{mem}(F)$ in addition to $\mathit{RecFunc}(F)$ for every function we may want to memoise; rather, one would like to have $\mathit{RecFunc}(F) \Rightarrow \mathit{RecFunc}_{mem}(F)$. This entailment can indeed be shown, but this is only possible because our specification idiom $\forall X. \cdots \otimes X$ in $\mathit{RecFunc}(F)$ allows us to do DFA-like reasoning.

5 Conclusions and Further Work

In the full version of this work we will provide detailed proofs as well as a soundness argument for the extension of the logic in [3] with second order assertions and some other minor additions. Moreover, we would like to investigate a similar extension to the logic with an *anti-frame rule* [6, 9] as this allows for a more natural treatment of hidden state in programs.

Acknowledgement This work has been supported by EPSRC grant EP/G003173/1. This material is as yet unpublished.

References

1. N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*, pages 3–14, 2009.
2. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5), 2006.
3. N. Charlton and B. Reus. Specification patterns and proofs for recursion through the store. Submitted, Apr. 2010.
4. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.
5. F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, pages 331–340, Pittsburgh, Pennsylvania, June 2008.
6. F. Pottier. Three comments on the anti-frame rule. Unpublished, July 2009.
7. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
8. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *CSL*, pages 440–454, 2009.
9. J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus. A semantic foundation for hidden state. In *FOSSACS*, pages 2–17, 2010.