A Deep Embedding of a Decidable Fragment of Separation Logic in HOL

Thomas Tuerk

16th July 2007

Smallfoot is a tool to automatically check separation logic specifications of sequential programs. It uses a decidable fragment of separation logic. In this paper, a deep embedding of a slightly different decidable fragment of separation logic inspired by Smallfoot is presented. Moreover, a HOL implementation of a decision procedure for entailments in this logic is described.

The main focus of this paper is on the introduction of the separation logic at an high level of abstraction and on inference rules for entailments. It is pointed out how these high-level concepts relate to the HOL embedding. However, the HOL embedding is not explained in detail.

Contents

1 Motivation

Separation logic [Rey02] is used to reason about shared mutable data structures. It allows one to express light-weight specifications about dynamically allocated pointer structures on a heap. For example, one can easily express that some pointer is the starting point of a linked list. More detailed properties, like e. g. constraints on the content of the list, are not expressible. The main feature of this logic is a special $*$-operator, which allows one to reason about disjoint parts of a heap. Most other logics have to express this disjointness explicitly. This usually means adding a number of constraints, which is quadratic in the number of resources used. In contrast, separation logic specifications are small. Moreover, simple and short proofs are possible in separation logic.

Smallfoot [BCO05b] is a tool that uses a decidable fragment of separation logic to automatically reason about specifications of programs written in a simple, imperative language. This work aims at formalising separation logic in the interactive theorem prover HOL [GM93]. Additionally, a decision procedure for entailments in the logic used has been implemented. As this decision procedure is a major part of Smallfoot, the HOL formalisation is likely to increase the trust in Smallfoot.

2 Basic Definitions

The definitions of the fragment of separation logic used here follow mainly the definition in [BCO05a]. However, some things have been slightly modified for the sake of the embedding.

Definition 1 (Values)**.** For a given set of values *Values* let $Values_{\mathsf{nil}}$ denote the extension of *Values* with a special value nil.

As nil is used to denote the null-pointer, it's often important that a set does not contain nil. Thus, we assume without loss of generality that *Values* does not contain nil. Moreover, it will be necessary in the following to introduce fresh values. Therefore, let *Values* denote a large enough set to always provide fresh values, i. e. let *Values* denote a countable infinite set.

Definition 2 (Expressions)**.** An *expression* e over a set of variables *Vars* and a set of values *Values* is defined to be either

- a constant $c \in \mathit{Values}_{\mathsf{nil}}$ or
- a variable $x \in \mathit{Vars}$.

The set of all *expressions* over *Vars* and *Values* is denoted by $\mathit{Exp}(\mathit{Vars}, \mathit{Values})$.

Definition 3 (Stacks)**.** A *stack* s over a set of variables *Vars* and a set of values *Values* is a function $s : \mathit{Vars} \to \mathit{Values}_{\mathsf{nil}}$.

The evaluation $[\![e]\!]s$ of an expression e in a stack s is given by

- $[\![c]\!]s := c$ for all constants c and
- $[\![x]\!]s := s(x)$ for all variables x.

Definition 4 (Pure Formulae)**.** The set $\mathit{pf}(\mathit{Vars}, \mathit{Values})$ of *pure formulae* over *Vars* and *Values* is recursively defined as the smallest set with

- $\mathsf{true} \in \mathit{pf}(\mathit{Vars}, \mathit{Values})$
- $(e_1 \doteq e_2) \in \mathit{pf}(\mathit{Vars}, \mathit{Values})$ for $e_1, e_2 \in \mathit{Exp}(\mathit{Vars}, \mathit{Values})$
- $(e_1 \not\doteq e_2) \in \mathit{pf}(\mathit{Vars}, \mathit{Values})$ for $e_1, e_2 \in \mathit{Exp}(\mathit{Vars}, \mathit{Values})$
- $(\mathit{pf}_1 \wedge \mathit{pf}_2) \in \mathit{pf}(\mathit{Vars}, \mathit{Values})$ for $\mathit{pf}_1, \mathit{pf}_2 \in \mathit{pf}(\mathit{Vars}, \mathit{Values})$.

For a stack s the semantics of a pure formula is given by

- $s \models_{pf} \mathsf{true}$
- $s \models_{pf} e_1 \doteq e_2$ iff $[\![e_1]\!]s = [\![e_2]\!]s$
- $s \models_{pf} e_1 \not\doteq e_2$ iff $[\![e_1]\!]s \neq [\![e_2]\!]s$
- $s \models_{pf} \mathit{pf}_1 \wedge \mathit{pf}_2$ iff $s \models_{pf} \mathit{pf}_1$ and $s \models_{pf} \mathit{pf}_2$

Thus, stacks are variable assignments and pure formula are simple restrictions on stacks. These constructs of stacks and pure formula are used to define the essential part of the logic: formulae over heaps.

Definition 5 (Fields)**.** To define heaps *fields* have to be introduced first. Fields are defined by there usage. Anything may be used as a field. In contrast to values, the set of fields may even be finite. In the HOL embedding strings or integers are usually used. Here, we will use the constants l, r, tl and t_i with $i \in \mathbb{N}_0$, which are assumed to be pairwise distinct.

Definition 6 (Heaps)**.** A *heap* h over a set of values *Values* and a set of fields *Fields* is a finite map $h : \mathit{Values} \xrightarrow{fin} (\mathit{Fields} \xrightarrow{fin} \mathit{Values}_{\mathsf{nil}})$.

As heaps are finite maps, a list notation can be used: Let $f := [x_1 \to y_1, \ldots, x_n \to y_n]$ denote the finite map $f : X \xrightarrow{fin} Y$ with $\mathrm{dom}(f) = \{x_1, \ldots, x_n\}$ and $f(x_i) = y_i$ for $1 \leq i \leq n$. Furthermore, let $h_1 \cup h_2$ denote the union of two heaps, i. e.

$$\begin{array}{rcl} \mathrm{dom}(h_1 \cup h_2) & := & \mathrm{dom}(h_1) \cup \mathrm{dom}(h_2) \\ (h_1 \cup h_2)(x) & := & \begin{cases} h_1(x) & \text{iff } x \in \mathrm{dom}(h_1) \\ h_2(x) & \text{otherwise} \end{cases} \end{array}$$

Using this definition of heaps and these notations, spatial formulae and their semantics can be defined:

Definition 7 (Spatial Formulae)**.** The set *sf*(*Vars*, *Values*) of *spatial formulae* over *Vars* and *Values* is recursively defined as the smallest set with

- $\mathsf{emp} \in$ *sf*(*Vars*, *Values*)
- $e \mapsto [t_1 : e_1, \ldots, t_k : e_k] \in$ *sf*(*Vars*, *Values*) for $e, e_1, \ldots, e_k \in$ *Exp*(*Vars*, *Values*) and $t_1, \ldots, t_k \in$ *Fields*
- $\mathsf{tree}\,((t_1, \ldots, t_k), es, e) \in$ *sf*(*Vars*, *Values*) for $es, e \in$ *Exp*(*Vars*, *Values*) and $t_1, \ldots, t_k \in$ *Fields*
- $sf_1 * sf_2 \in$ *sf*(*Vars*, *Values*) for $sf_1, sf_2 \in$ *sf*(*Vars*, *Values*).

For a stack s and a heap h the semantics of a spatial formula is given by

- $s, h \models_{sf} \mathsf{emp}$ iff $h = []$
- $s, h \models_{sf} sf_1 * sf_2$ iff $\exists h_1, h_2$.
  - $h = h_1 \cup h_2$ and
  - $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$ and
  - $s, h_1 \models_{sf} sf_1$ and
  - $s, h_2 \models_{sf} sf_2$
- $s, h \models_{sf} e \mapsto [t_1 : e_1, \ldots, t_k : e_k]$ iff $\exists r$.
  - $h = [[\![e]\!]s \to r]$ (i. e. h is just defined for the value $[\![e]\!]s$ and maps it to r) and
  - $\forall\ 1 \leq i \leq k.\ t_i \in \mathrm{dom}(r)$ and
  - $\forall\ 1 \leq i \leq k.\ r(t_i) = [\![e_i]\!]s$
- $s, h \models_{sf} \mathsf{tree}\,((t_1, \ldots, t_k), es, e)$ iff
  - $s \models_{pf} e \doteq es$ and $h = []$ or
  - $s \models_{pf} e \not\doteq es$ and $\exists e_1, \ldots, e_k.\ s, h \models_{sf} e \mapsto [t_1 : e_1, \ldots, t_k : e_k] *$ $\mathsf{tree}\,((t_1, \ldots, t_k), es, e_1) * \ldots * \mathsf{tree}\,((t_1, \ldots, t_k), es, e_k)$

Notice, that this definition is well founded. This may be difficult to see in the case of tree. However, each recursion that occurs in the definition of tree removes – according to the semantics of $\mapsto$ and $*$ – the root of the tree from the heap. Since the heap is by definition finite, this recursion has to terminate and the definition is well founded. However, in the HOL embedding a definition is used that talks explicitly about this number of recursions, i. e. about the height of the tree. Then, it is shown that the two definitions are equivalent.

Spatial formulae are the core of separation logic. They contain the $*$-operator, which requires that a heap can be split into disjoint heaps that satisfy the subformulas. Additionally, there is a predicate emp to describe the empty heap. The remaining predicates may be harder to understand intuitively. Therefore, lets introduce some syntactic sugar first, which may help to explain the intended semantics:

- $\mathsf{list}(t, e_1, e_2) := \mathsf{tree}(t, e_2, e_1)$
- $\mathsf{bin\text{-}tree}(t_1, t_2, e) := \mathsf{tree}((t_1, t_2), \mathsf{nil}, e)$

Intuitively, $s, h \models_{sf} \mathsf{bin\text{-}tree}(l, r, e)$ means that h contains a binary tree with root e. The left child is pointed to by the field index l, the right child is addressed by r. If e evaluates to nil, i. e. if $s \models_{pf} e \doteq \mathsf{nil}$ holds, the binary tree is empty and the stack should be empty as well. Otherwise, one has to check that the root node e points to a left child e_l and a right child e_r such that binary trees exists with these childs as roots. Formally, the expression evaluates to $\exists e_l, e_r.\ s, h \models_{sf} e \mapsto [l : e_l, r : e_r] * \mathsf{bin\text{-}tree}(l, r, e_l) * \mathsf{bin\text{-}tree}(l, r, e_r)$. This means, that the heap can be split into three disjoint parts h_e, h_1 and h_2 with

- $s, h_e \models_{sf} e \mapsto [l : e_l, r : e_r]$,
- $s, h_l \models_{sf} \mathsf{bin\text{-}tree}(l, r, e_l)$ and
- $s, h_r \models_{sf} \mathsf{bin\text{-}tree}(l, r, e_r)$.

According to the semantics of spatial formulae, the heap h_e maps $[\![e]\!]s$ to a finite map $f : \mathit{Fields} \xrightarrow{fin} \mathit{Values}_{\mathsf{nil}}$ such that $f(l) = [\![e_l]\!]s$ and $f(r) = [\![e_r]\!]s$ hold. This example illustrates the usage of *Fields*. They are used as indexes for different branches of a tree. Notice further, that $\mathrm{dom}(h_e) = \{[\![e]\!]s\}$ holds and that the domains of h_e, h_l and h_r are disjoint. Thus, whole h is used to model the tree and – except for the leaves – each node occurs just once in the tree.

Similarly, $\mathsf{list}(tl, e_1, e_2)$ describes a single linked, acyclic list from e_1 to e_2 following the pointers indexed by tl. Formally, it is modeled as an unary tree. For lists it makes sense to have arbitrary expressions as end points / leaves, whereas for binary trees only nil is allowed. The predicate tree is a generalisation of bin-tree and list. It has been introduced mainly because it makes modelling of this logic easier in HOL since properties have to be proven just once. However, the logic is now able to talk about trees of arbitrary, but fixed width.

Definition 8 (Normal Form)**.** The operators $\wedge$ and $*$ are both associative and commutative. Therefore, a list notation can be used:

$$\begin{array}{lllll} s & \models_{pf} & pf_1, \ldots, pf_n & := & s \models_{pf} pf_1 \wedge \ldots \wedge pf_n \wedge \mathsf{true} \\ s, h & \models_{sf} & sf_1, \ldots, sf_n & := & s, h \models_{sf} sf_1 * \ldots * sf_n * \mathsf{emp} \end{array}$$

In fact, one could even use sets for pure formulae and multisets for spatial formulae. However, its been easier to use lists in the HOL formalisation and proof the additional properties of sets and multisets explicitly. To be close to the HOL formalisation, a list notation is used here as well.

Notice that the following equivalences hold for a field index t occurring several times in a spatial formula:

$$\begin{array}{lll} s, h \models_{sf} & e \mapsto [t_1 : e_1, \ldots, t : e_n, \ldots, t : e_m, \ldots, t_k : e_k] & \iff \\ s \models_{pf} & e_n \doteq e_m \text{ and} & \\ s, h \models_{sf} & e \mapsto [t_1 : e_1, \ldots, t : e_n, \ldots, t_{m-1} : e_{m-1}, t_{m+1} : e_{m+1}, \ldots, t_k : e_k] & \end{array}$$

$$\begin{array}{lll} s, h \models_{sf} & \mathsf{tree}\,((t_1, \ldots, t, \ldots, t, \ldots, t_k), es, e) & \iff \\ s \models_{pf} & e \doteq es \text{ and} & \\ s, h \models_{sf} & \mathsf{emp} & \end{array}$$

These equivalences can be easily used to rewrite formulae. Therefore, we will just consider formulae such that all field indexes occurring in a tree or $\mapsto$ subformula are pairwise distinct. Additionally, we assume that true and emp do not occur in the lists. Otherwise they can be easily eliminated, since they are the identity.
As a writing convention let in the following

- pf denote a pure formula,
- sf denote a spatial formula,
- Π denote a list of pure formulae and
- Σ denote a list of spatial formulae.

These are the semantics of the subset of separation logic that will be considered here. We are mainly interested in checking the validity of entailments in this logic, i. e. we are interested in deciding whether statements of the form $\forall s, h.\ (s \models_{pf} \Pi) \wedge (s, h \models_{sf} \Sigma) \Rightarrow (s \models_{pf} \Pi') \wedge (s, h \models_{sf} \Sigma')$ hold.

However, to keep track of some information during calculations a slightly extended notation is introduced: a list of expressions is added such that the value of these expressions is not nil but not in $\mathrm{dom}(h)$. Moreover, all values of expressions in this list are pairwise distinct. This extension is useful to define the FRAME rule later. However, sometimes a expression should just be added to this list, if it is not equal to a second expression in the current stack. Adding a list of pairs of expressions for this purpose leads to the following definition:

Definition 9 (Entailments)**.** Let $\eta := e_1, \ldots, e_n$ be a list of expressions and $\pi := (e'_1, e''_1), \ldots, (e'_m, e''_m)$ be a list of pairs of expressions. Furthermore, let s be a stack and h a heap. Then the predicate *heap_distinct*(s, h, η, π) holds for s and h iff

- for all expressions e with $e \in \eta$ or $\exists e'.\ (e, e') \in \pi \wedge s \models_{pf} e \neq e'$ the following holds
  - $\neg\,(\llbracket e \rrbracket s = \mathsf{nil})$ and
  - $\neg\,(\llbracket e \rrbracket s \in \mathrm{dom}(h))$
- all values of expressions $e \in \eta$ and all values of expressions e' with $(e', e'') \in \pi$ and $s \models_{pf} e' \neq e''$ are pairwise distinct:
  - $\neg(i = j) \wedge 1 \leq i, j \leq n \Longrightarrow \neg\,(\llbracket e_i \rrbracket s = \llbracket e_j \rrbracket s)$ and
  - $\neg(i = j) \wedge 1 \leq i, j \leq m \wedge \Big(s \models_{pf} e'_i \neq e''_i \wedge e'_j \neq e''_j\Big) \Longrightarrow$ $\neg\,\Big(\llbracket e'_i \rrbracket s = \llbracket e'_j \rrbracket s\Big)$ and
  - $1 \leq i \leq n \wedge 1 \leq j \leq m \wedge \Big(s \models_{pf} e'_j \neq e''_j\Big) \Longrightarrow$ $\neg\,\Big(\llbracket e_i \rrbracket s = \llbracket e'_j \rrbracket s\Big)$

This definition of *heap_distinct* is quite technical. It is intended to preserve information, which would otherwise be lost during applications of the frame rule. The idea behind its definition will become more apparent as soon as the frame rule is presented. At the moment it is sufficient to notice, that for the empty list [] and for all stacks s and heaps h the statement *heap_distinct*$(s, h, [], [])$ holds. Therefore, the following notation is really capable of expressing entailments:

$$\begin{array}{lcl} \eta, \pi, \Pi, \Sigma & \vdash & \Pi', \Sigma' := \forall s, h.\mathit{heap\_distinct}(s, h, \eta, \pi) \wedge \\ & & \qquad (s \models_{pf} \Pi) \ \wedge \ (s, h \models_{sf} \Sigma) \Longrightarrow \\ & & \qquad (s \models_{pf} \Pi') \ \wedge \ (s, h \models_{sf} \Sigma') \end{array}$$

3 Comparison to the logic used by Smallfoot

The subset of separation logic presented here is very similar to the one used by Smallfoot. Its definitions follow mainly [BCO05a]. However, [BCO05a] just considers lists and binary trees. There is no predicate for trees with arbitrary, but fixed width. This predicate tree has been introduced here to help the deep-embedding in HOL. It is a generalisation of the predicates for lists and binary trees and helps to keep some proofs more succinct. The predicate *heap_distinct* is not used in [BCO05a]. Otherwise, the logics are identical.

However, Smallfoot uses an additional predicate for double linked lists and one for xor linked lists [BCO05b]. These could be easily added to the logic presented here. For sake of time and simplicity they have been omitted. Notice however, that these additions were kept in mind during this work. Thus, the extension should be straightforward.

4 Deep Embedding in HOL

`decidable_separationLogicScript.sml` contains the main part of the HOL embedding. Conversions that implement inference rules and a decision procedure for entailments can be found in `decidable_separationLogicLib.sml`. All other files are not interesting from a high-level point of view.

The following is intended to give an overview of the main parts of the implementation. This overview should – combined with some informations given in the next section – be sufficient to use HOL to reason about the validity of entailments. However, for everything else, one should use this short description just as an orientation and have a look at the files mentioned themselves.

The sets *Values*, *Fields* and *Vars* are modelled as free type variables. This leaves the problem that a constraint that the set *Values* is infinite has to be added to the precondition of some theorems. However, it is straightforward to define $Values_{\mathsf{nil}}$ and expressions using these free type variables. $Values_{\mathsf{nil}}$ is modelled by the HOL datatype `ds_value`, expressions by the HOL datatype `ds_expression`. Using these basic datatypes, the datatypes `ds_pure_formula` and `ds_spatial_formula` are defined for pure and spatial formulae, respectively:

```
val _ = Hol_datatype ‘ds_value =
     dsv_nil
   | dsv_const of 'value‘

val _ = Hol_datatype ‘ds_expression =
     dse_const of 'value ds_value
   | dse_var of 'vars‘;

val dse_nil_def = Define ‘dse_nil = dse_const dsv_nil‘

val _ = Hol_datatype ‘ds_pure_formula =
     pf_true
   | pf_equal of ('vars, 'value) ds_expression =>
                 ('vars, 'value) ds_expression
   | pf_unequal of ('vars, 'value) ds_expression =>
                   ('vars, 'value) ds_expression
   | pf_and of ds_pure_formula => ds_pure_formula‘;

val _ = Hol_datatype ‘ds_spatial_formula =
     sf_emp
   | sf_points_to of ('vars, 'value) ds_expression =>
                     ('field # ('vars, 'value) ds_expression) list
   | sf_tree of 'field list => ('vars, 'value) ds_expression =>
                ('vars, 'value) ds_expression
   | sf_star of ds_spatial_formula => ds_spatial_formula‘;
```

```
val sf_ls_def = Define ‘sf_ls f e1 e2 = sf_tree [f] e2 e1‘;

val sf_bin_tree_def = Define
   ‘sf_bin_tree (f1, f2) e = sf_tree [f1;f2] dse_nil e‘;
```

The semantics of these formulas are defined by the predicates `PF_SEM` and `SF_SEM`. In order to define these semantics, stacks are modeled by functions, heaps are modeled using finite maps. The definition of `PF_SEM` is straightforward, whereas the definition of `SF_SEM` is tricky because of trees. It takes some time to prove that the recursive definition used in this paper is well founded. Therefore, a predicate `SF_SEM___sf_tree_len`, which recurses over the maximal height of the tree, is used instead of the one used here. That the resulting semantics of trees is really the intended one, is ensured by theorem `SF_SEM___sf_tree_THM`.

After introducing the basic definitions, some basic facts are proved. These facts include that $*$ and $\wedge$ are commutative and associative. Using this knowledge, list versions of the semantic predicates are introduced: `LIST_PF_SEM`, `LIST_SF_SEM` and the combination `LIST_DS_SEM`. Using these lists versions and a predicate `HEAP_DISTINCT` (that corresponds to *heap_distinct*), the predicate `LIST_DS_ENTAILS` is finally defined that represents entailments.

Using these definitions, for example the entailment

$$x_1, (x_3, x_4), x_2 \doteq x_3, x_2 \mapsto [hd : x_5, tl : x_3], \mathsf{list}(tl, x_3, x_4) \quad \vdash \quad x_2 \doteq \mathsf{nil}, \mathsf{bin\text{-}tree}(l, r, x_5)$$

can be encoded in HOL as

```
val t = ‘‘LIST_DS_ENTAILS ([dse_var 1],[(dse_var 3, dse_var 4)])
  ([pf_equal (dse_var 2) (dse_var 3)],
   [sf_points_to (dse_var 2) [("hd",dse_var 5); ("tl", dse_var 3)];
    sf_ls "tl" (dse_var 3) (dse_var 4)])

   ([pf_equal (dse_var 2) dse_nil],
    [sf_bin_tree ("l", "r") (dse_var 5)])‘‘;
```

5 Proof System for Entailments

This work is mainly concerned with the validity of entailments. To reason about entailments a set of inference rules will be given. As usual, the semantics of the inference rule

$$\frac{A_1 \quad \dots \quad A_n}{B}$$

is that if all A_i hold, we can conclude that also B holds. Thus, the semantics can be expressed by $(\bigwedge_{i=1\dots n} A_i) \Rightarrow B$. Additionally, the notation

$$\frac{\quad A_1 \quad \dots \quad A_n \quad}{\quad\quad B \quad\quad}$$

will be used to indicate $(\bigwedge_{i=1\ldots n} A_i) \Leftrightarrow B$.

In the following these notations are used to present inferences for entailments. Then it will be explained how these inferences can be combined to form a decision procedure.

5.1 Inferences for Entailments

5.1.1 `ds_inference_REMOVE_TRIVIAL`

Using our notations for inferences, one can for example easily express inferences that remove trivial parts from an entailment:

RemoveTrivial-EQ-L

$$\frac{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, e \doteq e, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

RemoveTrivial-EQ-R

$$\frac{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \Sigma \quad \vdash \quad e \doteq e, \Pi', \Sigma'}$$

RemoveTrivial-EmpTree-L

$$\frac{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \mathsf{tree}((t_1, \ldots, t_k), e, e), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

RemoveTrivial-EmpTree-R

$$\frac{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{tree}((t_1, \ldots, t_k), e, e), \Sigma'}$$

RemoveTrivial-Precond

$$\frac{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, (e, e), \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

According to the definition of $\vdash$, the ordering of the lists η, π, Π, Σ, Π' and Σ' does not matter. Thus, these inferences and similar ones should be understood as sets of inferences, containing the interesting formulae at arbitrary positions instead of the heads of the lists. Similarly, we do not distinguish between some formulae, which obviously have the same semantics. So, $e_1 \doteq e_2$ and $e_2 \doteq e_1$ are not distinguished. The same holds for $e_1 \neq e_2$. Additionally, the order, in which fields occur in tree and $\mapsto$ formulae is not important. Thus, we do for example not distinguish between bin-tree(l, r, e) and bin-tree(r, l, e) or between $e \mapsto [t_1 : e_1, t_2 : e_2, t_3 : e_3]$ and $e \mapsto [t_2 : e_2, t_3 : e_3, t_1 : e_1]$.

The HOL conversion `ds_inference_REMOVE_TRIVIAL___CONV` is an implementation of these RemoveTrivial-inferences. It can also handle list and bin-tree. However, since the corresponding inferences are just instantiations of the tree inference, they won't be explained here. Similarly, special inferences for list and bin-tree won't be presented in the following, if they are just instantiations of a corresponding tree inference.

5.1.2 `ds_inference_AXIOM`

Another simple example for an inference is:

Axiom

$$\frac{}{\eta, \pi, \Pi \quad \vdash}$$

This inference requires, that the lists Σ, Π' and Σ' are empty. Moreover, there are no preconditions. Therefore, this inference means, that its conclusion always holds. Altogether the semantics of this inference rule are expressed by:

$$\forall \eta, \pi, \Pi.\ \forall s, h.\quad (\mathit{heap\_distinct}(s, h, \eta, \pi)\ \wedge\ (s \models_{pf}\ \Pi)\ \wedge\ (s, h \models_{sf}\ \mathsf{nil}))\implies (s \models_{pf}\ \mathsf{true})\ \wedge\ (s, h \models_{sf}\ \mathsf{nil})$$

This inference is implemented by `ds_inference_AXIOM___CONV`.

5.1.3 `ds_inference_SUBSTITUTION`

A more interesting example is the substitution inference, which does some actual work in simplifying an entailment. However, to understand this inference the notation $l[e/x]$ needs to be introduced. Informally, it denotes the list resulting from replacing every occurrence of the variable x in l with the expression e. This definition allows to write the following inference rule:

SUBSTITUTION

$$\frac{\eta[e/x], \pi[e/x], \Pi[e/x], \Sigma[e/x] \quad\vdash\quad \Pi'[e/x], \Sigma'[e/x]}{\eta, x \doteq e, \pi, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}$$

This inference is implemented by `ds_inference_SUBSTITUTION___CONV`.

5.1.4 `ds_inference_HYPOTHESIS`

The hypothesis inference removes pure formulae that occur on the left hand side of an entailment from the right hand side. Since some information has been moved to the precondition, it has to be used as well to eliminate pure formulae.

HYPOTHESIS-BASE

$$\frac{\eta, \pi, \mathit{pf}, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}{\eta, \pi, \mathit{pf}, \Pi, \Sigma \quad\vdash\quad \mathit{pf}, \Pi', \Sigma'}$$

HYPOTHESIS-PRECOND-NIL-RIGHT

$$\frac{e, \eta, \pi, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}{e, \eta, \pi, \Pi, \Sigma \quad\vdash\quad e \neq \mathsf{nil}, \Pi', \Sigma'}$$

HYPOTHESIS-PRECOND-NIL-LEFT

$$\frac{e, \eta, \pi, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}{e, \eta, \pi, e \neq \mathsf{nil}, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}$$

HYPOTHESIS-PRECOND-UNEQUAL-RIGHT

$$\frac{e_1, e_2, \eta, \pi, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}{e_1, e_2, \eta, \pi, \Pi, \Sigma \quad\vdash\quad e_1 \neq e_2, \Pi', \Sigma'}$$

HYPOTHESIS-PRECOND-UNEQUAL-LEFT

$$\frac{e_1, e_2, \eta, \pi, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}{e_1, e_2, \eta, \pi, e_1 \neq e_2, \Pi, \Sigma \quad\vdash\quad \Pi', \Sigma'}$$

These inferences are implemented by `ds_inference_HYPOTHESIS___CONV`. Additionally, this conversion removes duplicates in Π and Π'.

5.1.5 ds_inference_FRAME

It's much more complicated to remove spatial formulae. There is a frame inference that is very similar to the basic hypothesis inference:

$$\textsc{Frame-base}\quad \frac{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, sf, \Sigma \quad \vdash \quad \Pi', sf, \Sigma'}$$

However, this inference describes a real implication. This can be shown by the following example:

$$\frac{e \mapsto [g : e_2] \quad \vdash}{e \mapsto [f : e_1], e \mapsto [g : e_2] \quad \vdash \quad e \mapsto [f : e_1]}$$

$e \mapsto [g : e_2] \quad \vdash$ does not hold, since there is a heap such that e points to e_2 by field index g and since this heap is obviously not empty. However, the original entailment holds, since no heap can be split into two disjoint heaps such that both contain the value of e.

Thus, one has to be careful, in which order to apply inferences using this basic frame rule. [BCO05a] uses backtracking and ordering of the application of inference rules. However, one can use the predicate *heap_distinct* to record the otherwise lost information and get a real equivalence instead of just an implication. A similar method is used by newer versions of Smallfoot. However, it is not described in [BCO05a].

$$\textsc{Frame-points\_to}\quad \frac{e, \eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, e \mapsto [t_1 : e_1, \ldots, t_n : e_n], \Sigma \quad \vdash \quad \Pi', e \mapsto [t_1 : e_1, \ldots, t_m : e_m], \Sigma'} \qquad m \leq n$$

$$\textsc{Frame-tree}\quad \frac{\eta, (e, es), \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \mathsf{tree}((t_1, \ldots, t_k), es, e), \Sigma \quad \vdash \quad \Pi', \mathsf{tree}((t_1, \ldots, t_k), es, e), \Sigma'}$$

For $\mapsto$, the expression e is added to η. This means, that the value of e is not in the remaining heap and that it is distinct to the values of everything else in η. These two properties capture the properties of the $*$-operation. Additionally, it is stored that e does not equal to nil, which captures the properties of the $\mapsto$-operator itself.

For trees, it's very similar. However, the tree may be empty. Therefore, (e, es) is added to π, which has the same effect as adding e to π guarded by the condition that e and es do not evaluate to the same value, i. e. that the tree is not empty.

These inferences are implemented by `ds_inference_FRAME___CONV` for $\mapsto$ and tree. Notice, that this conversion does not apply the general frame inference. Thus, it's a real conversion which always results in an equality theorem. `ds_inference_FRAME___IMPL___CONV` applies the general frame inference as well and may therefore result in an implication.

5.1.6 `ds_inference_INCONSISTENT`

If there is no stack and heap that satisfies the left hand side, i. e. if the left hand side is inconsistent, then the whole entailment holds.

INCONSISTENT-UNEQUAL

$$\frac{}{\eta, \pi, e \neq e, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

INCONSISTENT-POINTSTO-NIL

$$\frac{}{\eta, \pi, \Pi, \mathsf{nil} \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

INCONSISTENT-PRECOND-NIL

$$\frac{}{\mathsf{nil}, \eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

INCONSISTENT-PRECONDITION-NOT-DISTINCT

$$\frac{}{e, e, \eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

INCONSISTENT-PRECOND-POINTSTO

$$\frac{}{e, \eta, \pi, \Pi, e \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

INCONSISTENT-PRECOND-BINTREE

$$\frac{}{e, \eta, \pi, \Pi, \mathsf{bin\text{-}tree}(l, r, e), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

These inference are implemented by `ds_inference_INCONSISTENT___CONV`.

5.1.7 `ds_inference_NIL_NOT_LVAL`

According to the semantics of $\mapsto$, nil cannot point to anything. The following inferences make this fact explicit.

NIL-NOT-LVAL-POINTSTO

$$\frac{\eta, \pi, e \neq \mathsf{nil}, \Pi, e \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, e \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

NIL-NOT-LVAL-TREE

$$\frac{\eta, \pi, e \neq \mathsf{nil}, e \neq es, \Pi, \mathsf{tree}\,(\ldots, es, e)\,, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, e \neq es, \Pi, \mathsf{tree}\,(\ldots, es, e)\,, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

This inference is implemented by `ds_inference_NIL_NOT_LVAL___CONV`. To prevent the inference rule and therefore the final decision procedure from looping, only facts are added that are not already present in Π or η.

5.1.8 `ds_inference_PARTIAL`

This inference makes implicit unequalities explicit.

PARTIAL-POINTSTO-POINTSTO

$$\frac{\eta, \pi, e_1 \neq e_2, \Pi, e_1 \mapsto [\ldots], e_2 \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, e_1 \mapsto [\ldots], e_2 \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

PARTIAL-POINTSTO-TREE

$$\frac{\eta, \pi, e_1 \neq e_2, e_2 \neq e_3, \Pi, e_1 \mapsto [\ldots], \mathsf{tree}(\ldots, e_3, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, e_2 \neq e_3, \Pi, e_1 \mapsto [\ldots], \mathsf{tree}(\ldots, e_3, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

PARTIAL-TREE-TREE

$$\frac{\eta, \pi, e_1 \neq e_2, e_1 \neq e_3, e_2 \neq e_4, \Pi, \mathsf{tree}(\ldots, e_3, e_1), \mathsf{tree}(\ldots, e_4, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, e_1 \neq e_3, e_2 \neq e_4, \Pi, \mathsf{tree}(\ldots, e_3, e_1), \mathsf{tree}(\ldots, e_4, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

PARTIAL-POINTSTO-PRECOND

$$\frac{e_1, \eta, \pi, e_1 \neq e_2, \Pi, e_2 \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}{e_1, \eta, \pi, \Pi, e_2 \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

PARTIAL-TREE-PRECOND

$$\frac{e_1, \eta, \pi, e_1 \neq e_2, e_2 \neq e_3, \Pi, \mathsf{tree}(\ldots, e_3, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}{e_1, \eta, \pi, e_2 \neq e_3, \Pi, \mathsf{tree}(\ldots, e_3, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

This inference is implemented by `ds_inference_PARTIAL___CONV`. To prevent the inference rule and therefore the final decision procedure from looping, only facts are added that are not already present in Π and η.

5.1.9 `ds_inference_SIMPLE_UNROLL`

These inferences perform unrolling of lists and binary trees for the simple cases, that either it can be deducted that a list is empty or that the head of a list / binary tree is known:

UNROLL-NILLIST

$$\frac{\eta, \pi, e \doteq \mathsf{nil}, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \mathsf{list}(tl, \mathsf{nil}, e), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

UNROLL-PRECOND-LIST

$$\frac{e_1, \eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{e_1, \eta, \pi, \Pi, \mathsf{list}(tl, e_1, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

UNROLL-RIGHT-LIST

$$\frac{e_1, \eta, \pi, e_1 \neq e_3, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_2, e_3), \Sigma'}{\eta, \pi, e_1 \neq e_3, \Pi, e_1 \mapsto [tl : e_2, \ldots], \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_1, e_3), \Sigma'}$$

UNROLL-RIGHT-BINTREE

$$\frac{e, \eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{bin\text{-}tree}(l, r, e_l), \mathsf{bin\text{-}tree}(l, r, e_r), \Sigma'}{\eta, \pi, \Pi, e \mapsto [l : e_l, r : e_r, \ldots], \Sigma \quad \vdash \quad \Pi', \mathsf{bin\text{-}tree}(l, r, e), \Sigma'}$$

These inferences are implemented by `ds_inference_SIMPLE_UNROLL___CONV`.

5.1.10 `ds_inference_STRENGTHEN_PRECONDITION`

The preconditions that result from the elimination of trees contain an implicit case split. These inferences are used to eliminate this case split:

STRENGTHEN-PRECOND-UNEQUAL

$$\frac{e_1, \eta, \pi, e_1 \neq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, (e_1, e_2), \pi, e_1 \neq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

STRENGTHEN-PRECOND-EQUAL-1

$$\frac{\eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, (e_1, e_2), (e_1, e_2), \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

STRENGTHEN-PRECOND-EQUAL-2

$$\frac{e_1, \eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{e_1, \eta, (e_1, e_2), \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

These inferences are implemented by `ds_inference_PRECONDITION_STRENGTHEN___CONV`.

5.1.11 `ds_inference_PRECONDITION_CASES`

It is sometimes necessary to do a case analysis on a precondition.

UNROLL-LEFT-PRECOND

$$\frac{\eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma' \qquad e_1, \eta, \pi, e_1 \neq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, (e_1, e_2), \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

This inference is implemented by `ds_inference_PRECONDITION_CASES___CONV`.

5.1.12 ds_inference_UNROLL

UNROLL-LIST

$$\frac{\eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma' \qquad \forall x.\ \eta, \pi, e_1 \not\doteq e_2, e_2 \not\doteq x, \Pi, e_1 \mapsto [\mathit{fl} : x], x \mapsto [\mathit{fl} : e_2], \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \mathsf{list}(\mathit{tl}, e_1, e_2), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

This inference does a case analysis on the length of lists. Its interesting to note that just two cases are sufficient to cover everything. Thus, it is not necessary to do induction for lists.

The general idea behind the correctness proof of this inference is that the logic can just access the two endpoints e_1 and e_2 of the list. It can not look at its internal representation, e. g. it cannot consider how many elements are between the endpoints. Therefore, a case analysis that considers the endpoints to be equal or to be unequal and not point at each other is sufficient.

This inference is an instance of a more general inference for arbitrary trees. However, besides lists, just the instance for binary trees is implemented as a HOL conversion:

UNROLL-BINTREE

$$\frac{\eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma' \qquad \forall x_1, x_2.\ \eta, \pi, e \not\doteq \mathsf{nil}, x_l \not\doteq \mathsf{nil}, x_r \not\doteq \mathit{nil}, \Pi, e \mapsto [l : x_l, r : x_r], x_l \mapsto [l : \mathsf{nil}, r : \mathsf{nil}], x_r \mapsto [l : \mathsf{nil}, r : \mathsf{nil}], \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \mathsf{bin\text{-}tree}(l, r, e), \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

However, these inferences are applied by the decision procedure as late as possible in order to avoid the case split and the introduction of fresh variables. Instead, they are used to derive specialised inference rules which are more efficient. And they are useful for the proof that the later described procedure is really a decision procedure.

`ds_inference_UNROLL_LIST___CONV` implements the list inference rule. There is a version that searches for $e_1 \not\doteq e_2$ in Π in order to be able to discard the first case: `ds_inference_UNROLL_LIST___NON_EMPTY___CONV`. The conversion for binary trees is called `ds_inference_UNROLL_BIN_TREE___CONV`.

5.1.13 ds_inference_UNROLL_RIGHT_CASES

In general its possible to do a case split, whenever one thinks it is useful:

EXCLUDEDMIDDLE

$$\frac{\eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma' \qquad \eta, \pi, e_1 \not\doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \Sigma'}$$

To really get a decision procedure in the very end, a case split is performed on trees

on the right hand side:

UNROLL-RIGHT-CASES

$$\frac{\eta, \pi, e_1 \doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{tree}((t_1, \ldots, t_k), e_2, e_1), \Sigma' \qquad \eta, \pi, e_1 \not\doteq e_2, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{tree}((t_1, \ldots, t_k), e_2, e_1), \Sigma'}{\eta, \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{tree}((t_1, \ldots, t_k), e_2, e_1), \Sigma'}$$

This case split is implemented by `ds_inference_UNROLL_RIGHT_CASES___CONV`.

5.1.14 `ds_inference_APPEND_LIST`

A combination of the inferences `ds_inference_UNROLL`, `ds_inference_SIMPLE_UNROLL` and `ds_inference_FRAME` leads to the following inference rules for lists:

APPEND-LIST-NIL

$$\frac{\eta, (e_1, e_2), \pi, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_2, \mathsf{nil}), \Sigma'}{\eta, \pi, \Pi, \mathsf{list}(tl, e_1, e_2), \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_1, \mathsf{nil}), \Sigma'}$$

APPEND-LIST-PRECOND

$$\frac{e3, \eta, (e_1, e_2), \pi, e_1 \not\doteq e_3, \Pi, \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_2, e_3), \Sigma'}{e3, \eta, \pi, e_1 \not\doteq e_3, \Pi, \mathsf{list}(tl, e_1, e_2), \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_1, e_3), \Sigma'}$$

APPEND-LIST-POINTSTO

$$\frac{\eta, (e_1, e_2), \pi, e_1 \not\doteq e_3, \Pi, e_3 \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_2, e_3), \Sigma'}{\eta, \pi, e_1 \not\doteq e_3, \Pi, \mathsf{list}(tl, e_1, e_2), e_3 \mapsto [\ldots], \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_1, e_3), \Sigma'}$$

APPEND-LIST-TREE

$$\frac{\eta, (e_1, e_2), \pi, e_1 \not\doteq e_3, e_3 \not\doteq e_4, \Pi, \mathsf{tree}(\ldots, e_4, e_3), \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_2, e_3), \Sigma'}{\eta, \pi, e_1 \not\doteq e_3, \Pi, \mathsf{list}(tl, e_1, e_2), \mathsf{tree}(\ldots, e_4, e_3), \Sigma \quad \vdash \quad \Pi', \mathsf{list}(tl, e_1, e_3), \Sigma'}$$

These inferences are implemented by `ds_inference_LIST_APPEND___CONV`.

5.2 The decision procedure

The inferences presented in the previous section can be easily combined to form a decision procedure (`ds_DECIDE_CONV`) for entailments not containing the general tree predicate, but just the instantiations list and bin-tree:

1. bring the entailment $\eta, \pi, \Pi, \Sigma \vdash \Pi', \Sigma'$ to normal form, i. e.
   - remove all occurrences of nil, $*$, true and $\wedge$ from $\Pi, \Sigma, \Pi', \Sigma'$
   - remove multiple occurrences of the same field index from Σ and Σ'
2. apply all the equality inferences (all except FRAME-BASE) in an arbitrary order, until no further inference applications are possible
3. the original entailment holds iff and only iff it has been simplified to *true*

To show that this algorithm is really a decision procedure for entailments, notice first that all steps preserve equality with the original entailment due to the correctness of the inference and rewrite rules. Moreover, all steps terminate. That's obvious for step 1. Step 2 may loop because of applications of HYPOTHESIS and NIL-NOT-LVAL / PARTIAL inferences. However, the sidecondition that no facts that are already explicitly present are added prevents such loops. Thus, the first two steps terminate and transform $\eta, \pi, \Pi, \Sigma \vdash \Pi', \Sigma'$ to a finite list of entailments $\eta_i, \pi_i, \Pi_i, \Sigma_i \vdash \Pi'_i, \Sigma'_i$ such that the following expression folds:

$$\left(\forall \vec{x}. \bigwedge_i \eta_i, \pi_i, \Pi_i, \Sigma_i \vdash \Pi'_i, \Sigma'_i\right) \iff \eta, \pi, \Pi, \Sigma \vdash \Pi', \Sigma'$$

Thus, if the list of entailments $\eta_i, \pi_i, \Pi_i, \Sigma_i \vdash \Pi'_i, \Sigma'_i$ is empty, the original entailment really holds. If it is not empty, it remains to show that the original entailment does not hold. Let $\eta_x, \pi_x, \Pi_x, \Sigma_x \vdash \Pi'_x, \Sigma'_x$ be an element of the list. Then it is sufficient to show, that this element does not hold.

This can be shown by constructing a concrete counterexample: We know that the entailment $\eta_x, \pi_x, \Pi_x, \Sigma_x \vdash \Pi'_x, \Sigma'_x$ is in normal form and that no inference rule is applicable. Thus, the following holds:

- $\pi_x = []$ because of UNROLL-LEFT-PRECOND
- Π_x contains just disequations $e_1 \not\doteq e_2$ with $e_1 \neq e_2$ because of SUBSTITUTION, REMOVETRIVIAL-EQ-L and INCONSISTENT-UNEQUAL.
- Σ_x contains just expressions of the form $e \mapsto [t_1 : e_1, \ldots, t_k : e_k]$ such that $e \notin \eta_x$, all t_i are distinct and all e are distinct

A case analysis on the exact form of the entailment, combined with the knowledge that no inference rule is applicable allows the construction of a counterexample for each case. The actual proof is tedious and omitted here.

6 Examples

6.1 Example 1

The following entailment is proofed by Smallfoot while the example specification `list.sf` is checked:

```
0!=z_3 * 0!=x * 0!=z * z_3!=y * z_3!=z * a!=e * x!=y * x!=z * y!=z *
z |-> hd:e,tl:y * listseg(tl; y, 0) * listseg(tl; x, z_3) * z_3 |-> tl:z
|-
listseg(tl; x, z) * z |-> tl:y * listseg(tl; y, 0)
```

Using the variable renaming $z_3 \rightarrow x_0$, $x \rightarrow x_1$, $z \rightarrow x_2$, $y \rightarrow x_3$, $a \rightarrow x_4$, $e \rightarrow x_5$, this can be written as

$$\begin{array}{l} x_0 \neq \mathsf{nil}, x_1 \neq \mathsf{nil}, x_2 \neq \mathsf{nil}, x_0 \neq x_3, x_0 \neq x_2, x_4 \neq x_5, x_1 \neq x_3, x_1 \neq x_2, x_3 \neq x_2, \\ x_2 \mapsto [hd : x_5, tl : x_3], \mathsf{list}(tl, x_3, \mathsf{nil}), \mathsf{list}(tl, x_1, x_0), x_0 \mapsto [tl, x_2] \\ \mathsf{list}(tl, x_1, x_2), x_2 \mapsto [tl : x_3], \mathsf{list}(tl, x_3, \mathsf{nil}) \end{array} \vdash$$

in the syntax of this work. It will now be shown, how the inferences rules can be used to show that this entailment holds. To keep things readable, let $\Pi := [x_0 \neq \mathsf{nil}, x_1 \neq \mathsf{nil}, x_2 \neq \mathsf{nil}, x_0 \neq x_3, x_0 \neq x_2, x_4 \neq x_5, x_1 \neq x_3, x_1 \neq x_2, x_3 \neq x_2]$ be used as an abbreviation.

$$\begin{array}{lll} \Pi, x_2 \mapsto [hd : x_5, tl : x_3], \mathsf{list}(tl, x_3, \mathsf{nil}), \mathsf{list}(tl, x_1, x_0), x_0 \mapsto [tl, x_2] & \vdash & \textsc{Frame} \\ \mathsf{list}(tl, x_1, x_2), x_2 \mapsto [tl : x_3], \mathsf{list}(tl, x_3, \mathsf{nil}) & & \Longleftrightarrow \\ \\ x_2, (x_3, \mathsf{nil}), \Pi, \mathsf{list}(tl, x_1, x_0), x_0 \mapsto [tl, x_2] & \vdash & \textsc{Append-List} \\ \mathsf{list}(tl, x_1, x_2) & & \Longleftrightarrow \\ \\ x_2, (x_3, \mathsf{nil}), (x_1, x_0), \Pi, x_0 \mapsto [tl, x_2] & \vdash & \textsc{Simple-Unroll} \\ \mathsf{list}(tl, x_0, x_2) & & \Longleftrightarrow \\ \\ x_2, x_0, (x_3, \mathsf{nil}), (x_1, x_0), \Pi & \vdash & \textsc{Remove-Trivial} \\ \mathsf{list}(tl, x_2, x_2) & & \Longleftrightarrow \\ \\ x_2, x_0, (x_3, \mathsf{nil}), (x_1, x_0), \Pi & \vdash & \textsc{Axiom} \\ & & \Longleftrightarrow \\ \\ \top & & \end{array}$$

Do do the same proof in HOL, one can either apply single inference rules manually or just call the decision procedure:

```
val t = ``LIST_DS_ENTAILS ([],[])
  ([pf_unequal (dse_var 0) dse_nil;
    pf_unequal (dse_var 1) dse_nil;
    pf_unequal (dse_var 2) dse_nil;
    pf_unequal (dse_var 0) (dse_var 3);
    pf_unequal (dse_var 0) (dse_var 2);
    pf_unequal (dse_var 4) (dse_var 5);
    pf_unequal (dse_var 1) (dse_var 3);
    pf_unequal (dse_var 1) (dse_var 2);
    pf_unequal (dse_var 3) (dse_var 2)],
   [sf_points_to (dse_var 2) [("hd",dse_var 5); ("tl", dse_var 3)];
    sf_ls "tl" (dse_var 3) dse_nil;
    sf_ls "tl" (dse_var 1) (dse_var 0);
    sf_points_to (dse_var 0) [("tl", (dse_var 2))]])
```

```
   ([],
    [sf_ls "tl" (dse_var 1) (dse_var 2);
     sf_points_to (dse_var 2) [("tl", dse_var 3)];
     sf_ls "tl" (dse_var 3) dse_nil])‘‘;

val thm1 =
(ds_inference_FRAME___CONV THENC
 ds_inference_APPEND_LIST___CONV THENC
 ds_inference_SIMPLE_UNROLL___CONV THENC
 ds_inference_REMOVE_TRIVIAL___CONV THENC
 ds_inference_AXIOM___CONV) t;

val thm2 = ds_DECIDE_CONV t;
```

6.2 Example 2

Lets now consider an entailment that does not hold:

$$
\begin{array}{lcll}
x \mapsto [tl : y] & \vdash & \mathsf{list}(tl, x, y) & \overset{\mathsf{Unroll-Right-Cases}}{\Longleftrightarrow} \\
 & & & \\
x \not\doteq y, x \mapsto [tl : y] & \vdash & \mathsf{list}(tl, x, y) \ \wedge & \\
x \doteq y, x \mapsto [tl : y] & \vdash & \mathsf{list}(tl, x, y) & \Longleftrightarrow \\
 & \vdots & & \\
x \doteq y, x \mapsto [tl : y] & \vdash & \mathsf{list}(tl, x, y) & \overset{\mathsf{Substitution}}{\Longleftrightarrow} \\
 & & & \\
x \mapsto [tl : x] & \vdash & \mathsf{list}(tl, x, x) & \overset{\mathsf{Remove-Trivial}}{\Longleftrightarrow} \\
 & & & \\
x \mapsto [tl : x] & \vdash & & \overset{\mathsf{NIL-NOT-LVAL}}{\Longleftrightarrow} \\
x \not\doteq \mathsf{nil}, x \mapsto [tl : x] & \vdash & &
\end{array}
$$

Now, no more inferences are applicable. However, one can easily see, that the entailment does not hold.

Inferences to detect false entailments are not implemented in HOL. Thus, the above reasoning looks in HOL as follows:

```
> val t = ‘‘LIST_DS_ENTAILS ([],[])
              ([], [sf_points_to (dse_var 0) [("tl", dse_var 1)]])
              ([], [sf_ls "tl" (dse_var 0) (dse_var 1)])‘‘;

- val thm = ds_DECIDE_CONV t;

> val thm =
     []
```

```
|- LIST_DS_ENTAILS ([],[])
     ([],[sf_points_to (dse_var 0) [("tl",dse_var 1)]])
     ([],[sf_ls "tl" (dse_var 0) (dse_var 1)]) =
   LIST_DS_ENTAILS ([],[])
     ([pf_unequal (dse_var 0) dse_nil],
      [sf_points_to (dse_var 0) [("tl",dse_var 0)]]) ([],[]) : thm
```

7 Conclusions and Future Work

We formalised the subset of separation logic described in [BCO05a] in HOL. This formalisation contains correctness proofs for the inferences used in Smallfoot. Thus, the work increases the trust in this tool. Moreover, the decision procedure for entailments has been implemented in HOL.

In the future, this formalisation and the decision procedure could be used to implement a HOL version of Smallfoot. Another interesting project may be to extend the fragment of separation logic used here and explore possibilities to interactively prove more complicated statements in this extended logic.

References

[BCO05a] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic, 2005.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University, 1993.

[Rey02] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.