Local Reasoning about While-Loops

Thomas Tuerk

University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
`http://www.cl.cam.ac.uk`

Abstract. Separation logic is an extension of Hoare logic that allows local reasoning. Local reasoning is a powerful feature that often allows simpler specifications and proofs. However, this power is not used to reason about while-loops.
In this paper an inference rule is presented that allows using local reasoning to verify the partial correctness of while-loops. Instead of loop invariants this inference rule uses pre- and post-conditions for loops. This provides a different view of while-loops that is even without local reasoning often beneficial.

1 Motivation

There is a well known connection between loops and recursive procedures. Modern compilers routinely transform recursive procedures into loops as part of program optimisation and there are refactorings that depending on varying criteria transform iterative programs in recursive ones and vice-versa. However, when using separation logic, recursive implementations are often much easier to specify and verify than the corresponding imperative ones.

In the following I will look into this surprising observation a little closer considering the example of determining the length of a single-linked list. This example – like all others in this paper – has been verified using *Holfoot* [8], a *Separation logic* [5, 6] tool similar to the tool Smallfoot [1]. Both tools, Holfoot and Smallfoot, use a very similar programming language. However, Holfoot supports a richer specification language as well as interactive proofs. This enables it to reason about fully functional specifications, whereas Smallfoot just reasons about the shape of datastructures. Details about the syntax and semantics of the used specification and programming language are not important for this paper and are therefore not discussed here. The programming language is easy to understand, because its syntax is similar to C. The specification language has uncommon syntax like using underscores to denote existential quantification. Therefore, in this paper specifications will be presented in an informal, intuitive way. All specifications are just concerned with partial correctness; termination is not considered.

The examples as well as Holfoot itself are available at its webpage[1]. There is also a web-interface that allows experimenting with the examples. Holfoot

[1] `http://holfoot.heap-of-problems.org`

is implemented inside the HOL 4 [2, 7] theorem prover. Thus, there are formal, machine readable semantics of both the programming and the specification language and all reasoning is done by proof inside HOL 4. In particular this means that all inference rules – including the one presented in this paper – are proven correct inside HOL 4.

So, let's consider a recursive and an iterative implementation of determining the length of a single-linked list:

```
list_length(r;c) [list(c,cdata)] {
  local t;
  if (c == NULL) {
    r = 0;
  } else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
} [list(c,cdata) * (r = length(cdata))]
```

```
list_length_iter(r;c) [list(c,cdata)] {
  local t;
  r = 0; t = c;
  while (t != NULL) [∃cdata1 cdata2.
     lseg(c, cdata1, t) *
     list(t, cdata2) *
     (r = length(cdata1)) *
     (cdata = cdata1 + cdata2)] {
    t = t->tl;
    r = r + 1;
  }
} [list(c,cdata) * (r = length(cdata))]
```

Both procedures get two arguments: a call-by-reference argument `r` and a call by value one `c`. The preconditions demand that `c` points to the start of a single-linked list that contains some data `cdata`. The postconditions guarantee, that this list is left unmodified and r is updated to contain the length of the list. While this specification is sufficient for the recursive implementation, the iterative one needs a complicated loop invariant. The loop-invariant (see Fig. 1)

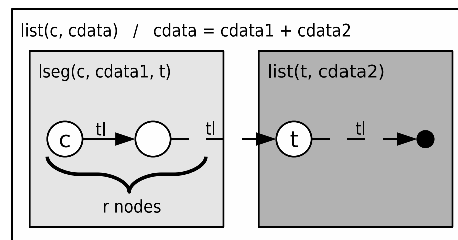

Fig. 1. Loop Invariant of list_length_iter

demands that the list can be split into two parts: the part that has already been counted and the one that still needs processing. The already counted part is a list-segment from `c` to `t` containing `cdata1`. The current value of `r` has to be the length of this already counted part. The unprocessed part is a null-terminated single-linked-list starting at `t` and containing `cdata2`. Combined, the two parts need to form the original list, i. e. appending `cdata1` and `cdata2` results in `cdata`.

Summing up, there are two implementations of the same algorithm. They have the same interface with exactly the same procedure specifications. However, while this specification is sufficient for the recursive implementation, the

iterative one needs to be annotated with a complicated loop invariant. This invariant is not just lengthy, it needs new concepts. Only the invariant needs to talk about list-segments, a partial datastructure. Both implementations do essentially the same, so why is it so much harder to specify the iterative one? The answer is that the specification of the recursive procedure call can utilise separation logic's local reasoning whereas the loop invariant does not use it.

The recursive implementation checks first[2], whether the list is empty. If it is not empty, the first element can be split off. The recursive call then determines the length of the remaining list. Thanks to local reasoning the specification of the recursive call has to mention just the tail of the original list. It is implicitly guaranteed that the first element of the original list is not modified. In contrast, the loop invariant describes the whole state / the whole datastructure. The list-segment from `c` to `t`, which is handled implicitly in the recursive implementation, is mentioned explicitly in the loop invariant.

In the following a inference rule is presented that allows the usage of local reasoning for the verification of while loops. Instead of loop invariants, this inference rule uses pre- and post-conditions for while loops. It allows to specify the list-length example as follows:

```
list_length_iter(r;c) [list(c,cdata)] {
  local t;
  r = 0; t = c;
  loop_spec [list(t,data)] {
    while (t != NULL) { t = t->tl; r = r + 1; }
  } [list(old(t),data) * (r = old(r) + length(data))]
} [list(c,cdata) * (r = length(cdata))]
```

This specification states that assuming `t` points to a start of a single linked list before the loop, then after the execution of the loop, there is still the same list in memory and the value of the variable `r` has been increased by the length of the list. So, this specification is using local reasoning. The list-segment between `c` and `t` is handled implicitly.

2 A closer look at the inference rule for while-loops

As motivated, local reasoning is an powerful feature that often allows simpler, shorter specifications and proofs. However, the inference rule for while-loops does not allow to utilise it. Let's have a closer look at this inference rule and see whether it can be modified to allow local reasoning. Hoare Logic [4] provides the following inference rule to reason about the partial correctness of while-loops:

WHILE RULE
$$\frac{\{c \wedge I\}\ p\ \{I\}}{\{I\}\ \textbf{while}\ \ c\ \ \textbf{do}\ \ p\ \ \textbf{done}\ \{\neg c \wedge I\}}$$

[2] Holfoot's web-interface can be used to step through the proof. It can be found at `http://holfoot.heap-of-problems.org`.

This inference rule can informally be justified by an induction on the number of loop iterations:

$\{I\}$ **while** c **do** p **done** $\{\neg c \wedge I\}$	$\Leftarrow$	induction / unroll
$\{\neg c \wedge I\}$ $\{\neg c \wedge I\}$ $\wedge$ $\{c \wedge I\}$ $p;$ **while** c **do** p **done** $\{\neg c \wedge I\}$	$\Leftarrow$	use induction hypothesis that the while loop satisfies the specification
$\forall$*prog*. $\{I\}$ *prog* $\{\neg c \wedge I\} \longrightarrow$ $\{c \wedge I\}$ $p;$ *prog* $\{\neg c \wedge I\}$	$\Leftarrow$	sequential composition rule
$\{c \wedge I\}$ p $\{I\}$		

The possibility to use local reasoning is lost in the last step, the application of the composition rule. Separation logic's local reasoning guarantees that any Hoare triple can be extended by an arbitrary context R:

$$\{P\}\ \textit{prog}\ \{Q\} \iff \forall R.\ \{P * R\}\ \textit{prog}\ \{Q * R\}$$

However, applying the sequential composition rule in the described way, ignores the possibility to extend the specification of *prog* with a frame R. Let's try to extend the inference rule for while-loops to be able use local reasoning. This extension should be as general as possible.

Slightly generalised, the identified problem with the classical while-rule is, that it is designed for single Hoare triples $\{P\}$ *prog* $\{Q\}$. In practice however, one often reasons about families $\{P_1\}$ *prog* $\{Q_1\}$, $\{P_2\}$ *prog* $\{Q_2\}$, … of specifications. As seen with the frame rule before, such families are usually represented using higher order quantification, i. e. they are given in the form $\forall x.\ \{P(x)\}$ *prog* $\{Q(x)\}$. If the classical while-rule is used for such a family, it results in

WHILE RULE FOR FAMILIES

$$\frac{\forall x.\ \{c \wedge I(x)\}\ p\ \{I(x)\}}{\forall x.\ \{I(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I(x)\}}$$

One can do better than this. This derived rule reasons about every member of the family, about every instantiation of x separately. The other members, other instantiations are ignored. In order to use that additional knowledge let's replay the informal justification for the classic while-rule with the quantifier in mind:

$$\begin{array}{ll}
\forall x.\ \{I(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I(x)\} & \Leftarrow \\
\forall x, \textit{prog}.\ \big(\forall y.\ \{I(y)\}\ \textit{prog}\ \{\neg c \wedge I(y)\}\big) \longrightarrow & \\
\qquad \{c \wedge I(x)\}\ p;\ \textit{prog}\ \ \{\neg c \wedge I(x)\} & \Leftarrow \\
\forall x.\ \exists y.\ \{c \wedge I(x)\}\ p\ \{I(y)\}\ \wedge\ \big(\neg c \wedge I(y) \rightarrow \neg c \wedge I(x)\big) &
\end{array}$$

This rule is more general than the classical one. The classic one always instantiates y with x. Now the induction hypothesis is stronger and one can actually use a different instantiation for the next iteration through the loop. As discussed before, separation logic's local reasoning can be expressed as such

a family of specifications. Therefore, reasoning about families of specifications instead of single ones allows using local reasoning for loops. This is the core of the proposed inference rule for loops. It leads to the following rule:

$$\textsc{Extended While Rule}\quad \frac{\forall x, \mathit{prog}.\ \big(\forall y.\ \{I(y)\}\ \mathit{prog}\ \{\neg c \wedge I(y)\}\big) \longrightarrow \{c \wedge I(x)\}\ p;\ \mathit{prog}\ \ \{\neg c \wedge I(x)\}}{\forall x.\ \{I(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I(x)\}}$$

The extended while rule allows to choose for each loop iteration a different instantiation. So, I is still some kind of inductive property of the loop, but strictly speaking no invariant any more. So let's use proper pre- and postconditions for the loop. By introducing pre- and post-conditions, the case for skipping the loop becomes more complicated. There is an additional proof obligation that the pre-condition implies the post-condition. Since there is this additional proof obligation anyhow, one can easily allow code after the loop. This results in the inference rule proposed in this paper:

$$\textsc{Loop Specification Rule}\quad \frac{\begin{array}{c}\forall x.\ \{\neg c \wedge P(x)\}\ p_2\ \{Q(x)\} \\ \forall x, \mathit{prog}.\ \big(\forall y.\ \{P(y)\}\ \mathit{prog}\ \{Q(y)\}\big) \longrightarrow \{c \wedge P(x)\}\ p_1;\ \mathit{prog}\ \{Q(x)\}\end{array}}{\forall x.\ \{P(x)\}\ \textbf{while}\ c\ \textbf{do}\ p_1\ \textbf{done};\ p_2\ \{Q(x)\}}$$

Both extensions of the extended while rule can be justified using exactly the same reasoning as before. These extensions are natural and prove useful. Especially, using real pre- and post-conditions is convenient. However, these extensions are unrelated to the main idea of using local reasoning for loops.

Notice, that this discussion about inference rules is intentionally very informal. There has been no definition of the semantics of the programming language and no definition of Hoare triples. The purpose of this discussion is to convey the main ideas. In contrast to this informal discussion here, the implementation of this proposed inference rule in Holfoot is formal. Everything is defined using higher order logic and its soundness is machine checked using the HOL 4 theorem prover.

3 Examples

I hope, I could convince you that loop-specifications are advantageous for the initial example of calculating the length of a single-linked list. There are similar results for reversing a single-linked list, copying a single-linked list, appending two single-linked lists, removing an element from a single-linked list, etc. Due to space restrictions most of these examples are not discussed here. They can be found on Holfoot's web-page[3].

Before considering examples that are very similar to the motivating one, let's start by discussing examples that demonstrate that even without local reasoning loop specifications are still useful. In contrast to invariants, the pre- and

[3] http://holfoot.heap-of-problems.org

post-condition specify the behaviour of the block containing the while loop. Therefore, the loop specification rule is closely related to Eric Hehner's *specified blocks* [3]. Hehner uses single boolean expressions instead of a pre- and post-condition. Moreover, his work is much more general. However, he is not using local reasoning. Allowing for these differences, his method of reasoning about loops is very similar to the one proposed here.

3.1 Array Increment Example

Similar to Hehner's specified blocks, loop specifications slightly change how to think about loops. As a rule of thumb, loop invariants express what the loop has already done, whereas loop specifications express what it will still do. Talking about what still needs doing instead of what has already been done, often leads to more natural specifications. Even without local reasoning, Hehner prefers loops specified as blocks to invariants. He claims *that it is simpler and more direct to say what's left to be done, rather than to formulate an invariant* [3]. This difference between loop invariants and loop specifications is demonstrated by one of Hehner's examples:

```
inc(;a,n) [array(a,n,data)] {
   local i, tmp;
   i = 0;
   while (i < n) {
      tmp = (a + i) -> dta;
      (a + i) -> dta = tmp + 1;
      i = i + 1;
   }
} [array(a,n,map +1 data)]
```

This procedure increments every element of an array. The loop can be specified with the following invariant:

$$\begin{aligned}\exists \mathit{data}_2.\ &\mathrm{array}(\mathtt{a},\mathtt{n},\mathit{data}_2) \ast \\ &\big(\forall x.\quad x < \mathtt{i} \quad \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x] + 1\big) \ast \\ &\big(\forall x.\ \mathtt{i} \le x < \mathtt{n} \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x]\big)\end{aligned}$$

The invariant states that there is an array of length n starting at a and containing some existentially quantified data $data_2$. For all indices up to i the array contains the incremented value, for all other indices it still contains the original one. If a loop specification is used, it is the other way round:

$$\begin{aligned}\textbf{pre:}\ &\mathrm{array}(\mathtt{a},\mathtt{n},\mathit{data}) \\ \textbf{post:}\ &\exists \mathit{data}_2.\ \mathrm{array}(\mathtt{a},\mathtt{n},\mathit{data}_2) \ast \\ &\quad\big(\forall x.\quad x < \mathrm{old}(\mathtt{i}) \quad \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x]\big) \ast \\ &\quad\big(\forall x.\ \mathrm{old}(\mathtt{i}) \le x < \mathtt{n} \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x] + 1\big)\end{aligned}$$

This specification states that all the indices starting at the value of i will be updated, while all smaller than i are not touched. Notice, that no local reasoning is involved here yet. Using local reasoning, the loop specification can however be simplified by implicitly handling the part of the array that is not touched.

$$\begin{aligned}\textbf{pre:}\ &\mathrm{array}(\mathtt{a}+\mathtt{i},\mathtt{n}-\mathtt{i},\mathit{data}) \\ \textbf{post:}\ &\mathrm{array}(\mathtt{a}+\mathrm{old}(\mathtt{i}),\mathtt{n}-\mathrm{old}(\mathtt{i}),\mathrm{map}\ (+1)\ \mathit{data})\end{aligned}$$

This specification now states that given an array starting from $\mathtt{a} + \mathtt{i}$ of length $\mathtt{n} - \mathtt{i}$ – i. e. just the part of the original array starting at index $\mathtt{i}$ – all elements of this array are incremented. There is no need any more for some complicated expressions about indices.

3.2 List Filtering Example

The last example demonstrates that loop invariants usually specify what has already been done, whereas loop specifications specify what will be done. However, both views were easy to express. The following example of filtering a list demonstrates that it might be much simpler to express what the loop will still do. Notice that this example is not exploiting local reasoning.

```
list_filter(l;x) [list(l,data)] {
  local y, z, e;
  y = l; z = NULL;
  while (y != NULL) {
     e = y->dta;
     if (e == x) { /* need to remove y */
       if (y == l) { /* first link */
          l = y->tl; dispose y; y = l;
       } else { /* not first link */
          e = y->tl; z->tl = e; dispose y; y = z->tl;
       }
     } else { /* don't need to remove y */
        z = y;  y = y->tl;
     }
  }
} [list(l, filter x from data)]
```

The loop invariant describes that parts of the list got already filtered. This partial filtering is complicated to express:

$$
\begin{array}{l}
\textbf{if } (\mathtt{y} = \mathtt{l}) \textbf{ then} \\
\quad \exists data_1.\ (data = (\text{some } \mathtt{xs}) + data_1) * \text{list}(\mathtt{l}, data_1) \\
\textbf{else} \\
\quad \exists data_1, date, data_2.\ (data = data_1 + date + (\text{some } \mathtt{xs}) + data_2) * \\
\qquad\qquad \text{lseg}(\mathtt{l}, \text{filtered } data_1, \mathtt{z}) * (z \mapsto [\mathtt{tl} : \mathtt{y},\ \mathtt{dta} : date]) * \\
\qquad\qquad date \neq \mathtt{x} * \text{list}(\mathtt{y}, data_2)
\end{array}
$$

This invariant is even worse than it looks, because the shorthand (some $\mathtt{x}s$) is used to denote a list of unknown length that consists of just the element $\mathtt{x}$. In contrast, the loop specification is straightforward, because it describes that the whole list starting at $\mathtt{y}$ will be filtered.

$$
\begin{array}{ll}
\textbf{pre:} & \text{list}(\mathtt{y}, data_2) * \\
& \textbf{if } (\mathtt{y} \neq \mathtt{l}) \textbf{ then } \text{lseg}(\mathtt{l}, data, \mathtt{z}) * (\mathtt{z} \mapsto [\mathtt{tl} : \mathtt{y}, \mathtt{dta} : zdate]) \textbf{ else } \text{emp} \\
\textbf{post:} & \textbf{if } (\text{old}(\mathtt{y}) = \text{old}(\mathtt{l})) \textbf{ then } \text{list}(\mathtt{l}, \text{filtered } data_2) \\
& \textbf{else } \text{list}(\mathtt{l}, data + zdate + \text{filtered } data_2)
\end{array}
$$

3.3 List Copy Example

After considering examples for which loop specifications proved beneficial even without local reasoning, let's have a look at an example with local reasoning:

```
list_copy(z;c) [list(c,data)] {
  local x,y,w,d;
  if (c==NULL) { z=NULL; }
  else {
    z=new(); z->tl=NULL; x = c->dta; z->dta = x; w=z; y=c->tl;
    while (y!=NULL) {
      d=new(); d->tl=NULL; x=y->dta; d->dta=x; w->tl=d; w=d; y=y->tl;
    }
  }
} [list(c,data) * list(z,data)]
```

This procedure copies a single-linked list that starts at c and updates the call-by-reference argument z such that z points to the copy after execution. The procedure first checks, whether the list is empty. In this case, nothing needs to be copied. Otherwise, the first element is copied and auxiliary variables w and y initialised. After this initialisation, z points to the beginning of the copy, w points to its last element and y points to the part of the original list that still needs to be copied. Then a while loop is used to copy the remainder of the list by copying the element pointed to by y and then advancing y and w.

The while-loop can be specified with the following invariant:

$$\begin{aligned}\exists data_1,\ cdate,\ data_2.\ & (data = data_1 + cdate + data_2) * \\ & \mathrm{lseg}(\mathtt{c}, data_1 + cdate, y) * \\ & \mathrm{lseg}(\mathtt{z}, data_1, w)\ *\ (\mathtt{w} \mapsto [\mathtt{tl}: 0, \mathtt{dta}: cdate]) * \\ & \mathrm{list}(\mathtt{y}, data_2)\end{aligned}$$

This invariant states that the original data can be split into three parts: two lists $data_1$, $data_2$ and a single element *cdate*. There is a list-segment from c to y containing $data_1$ followed by *cdate*. This part of the original list has already been copied. The data $data_1$ has been copied to a list-segment from z to w. The last entry *cdate* is pointed to by w. Finally, $data_2$ still needs to be copied. It is stored in a list starting at y.

Using a loop specification simplifies reasoning about the loop significantly:

$$\begin{aligned}&\textbf{pre:}\quad \mathtt{w} \mapsto [\mathtt{tl}: 0, \mathtt{dta}: cdate]\ *\ \mathrm{list}(\mathtt{y}, data_2) \\ &\textbf{post:}\ \mathrm{list}(\mathrm{old}(\mathtt{w}), cdate + data_2)\ *\ \mathrm{list}(\mathrm{old}(\mathtt{y}), data_2)\end{aligned}$$

This specification states that if before the loop is executed w points to some data *cdate* and there is a list starting at y containing $\mathtt{data}_2$, then the list starting at y is copied such that the old value of w points to a list containing *cdate* followed by $data_2$ after the execution of the loop. The part of the list that has already been copied, i. e. the list-segment from c to y does not need to be mentioned explicitly. It is handled implicitly using local reasoning. Notice moreover that the loop specification does not use list-segments.

3.4 Partial Datastructures

Loop specifications can utilise local reasoning in order to implicitly handle some part of the state that loop invariants mention explicitly. This implicitly handled part of the state is usually a partial datastructure. For the examples so far, these partial data structures are easy to express. For lists, the partial datastructure is a list-segment and for arrays it is an array. Let's now consider a slightly more complicated datastructure: trees. For trees, the corresponding partial datastructure is a tree with a hole for some other tree. This is difficult to express. Separation logic's magic-wand operator can be used, but reasoning about this additional operator is not straightforward and Holfoot is not able to do it. Therefore, Holfoot usually can't handle the invariants of loops that operate on trees. However, loop specifications can be used to avoid the partial datastructure. This allows Holfoot to reason about additional examples like the following:

```
search_tree_delete_min (t,m;) [binary_search_tree(t;keys) * (keys ≠ ∅)] {
  local tt, pp, p;
  p = t->l;
  if (p == 0) { m = t->dta; tt = t->r; dispose (t); t = tt; } else {
    pp = t; tt = p->l;
    loop_spec [binary_search_tree(p, keys2) & (pp points to p and p to tt)] {
      while (tt != NULL) { pp = p; p = tt; tt = p->l; }
      m = p->dta; tt = p->r; dispose(p); pp->l = tt;
    } [∃p2. binary_search_tree(p2, keys2 without min(keys2)) & (pp points to p2)]
  }
} [binary_search_tree(t;keys without min(keys)) * (m = min(keys))]
```

This procedure deletes the minimal key from a non-empty binary search tree. The while-loop is used to search for the node storing the minimal key. After the loop has been executed, the original binary-search tree is unmodified and the variable `p` points to the node holding the minimal key and `pp` to its parent node. However, expressing these properties of `p` and `pp` is complicated and would require some kind of partial tree datastructure. Therefore, the code that deletes the minimal element is included in the loop specification. Thus, the post-condition of the loop specification can state, that the minimal key of the original tree has been deleted. In contrast to the corresponding loop invariant, the loop specification does not need partial tree datastructures.

Besides demonstrating that loop specifications can be used to eliminate the need for partial datastructures, the last example also demonstrates why it is useful that loop specifications allow code after the while-loop. Allowing code after the loop is a minor extension, that is not used by most of the examples that I considered so far. However, as this example illustrates, it sometimes results in much simpler post-conditions.

4 Conclusion

In this paper an additional inference rule for while loops is presented. This *loop specification rule* uses pre- and post-conditions instead of invariants. Loop invariants express what the loop has done so far. In contrast loop specifications state what the loop will still do. This often leads to more natural specifications.

The loop specifications presented here are very similar to Eric Hehner's specified blocks [3]. Even without local reasoning they often lead to simpler, more natural specifications as demonstrated by the list filtering example. However, they have mainly been introduced in order to be able to use separation logic's local reasoning for loops. Using local reasoning, loop specifications gain their full potential. Besides leading to even simpler specification, local reasoning can be used to avoid the need for predicates describing partial datastructures.

Loop specifications have been implemented inside Holfoot. This implementation includes a formal correctness proof inside the HOL 4 theorem prover. There are many Holfoot examples available that demonstrate that loop specifications can simplify the specification and verification of loops considerably. There are examples for single linked lists like reversing a single-linked list, copying a single-linked list, appending two single-linked lists, removing an element from a single-linked list, examples for arrays like copying an array, binary search, quicksort and examples for binary trees like binary search tree lookup and deletion or traversing a tree with a user managed stack. These examples and many others can be found on Holfoot's webpage [4]. The binary tree examples might be especially interesting. These could not be handled by Holfoot without loop specifications, because Holfoot does not support predicates that are able to describe the otherwise necessary partial datastructures.

Acknowledgements

I would like to thank Rustan Leino, David Naumann, Peter O'Hearn, Matthew Parkinson and Hongseok Yang for discussions, comments and especially for pointing me to Eric Hehner's work.

[4] `http://holfoot.heap-of-problems.org`

Bibliography

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

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

[3] Eric C. R. Hehner. Specified blocks. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[5] P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, September 2001.

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

[7] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[8] Thomas Tuerk. A formalisation of Smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer, 2009.