Local Reasoning about While-Loops

Thomas Tuerk

Computer Lab, University of Cambridge

VSTTE'10
18th August 2010

Motivation I

- I'm developing Holfoot
- Holfoot is a separation logic tool in HOL
- observation: recursive implementations are often much simpler to specify than iterative ones
- why is this so?

Motivating Example

Length of a single linked list

```
list_length(r;c) {
  local t;
  if (c == NULL) {
    r = 0;
  } else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
}
```

```
list_length_iter(r;c) {
  local t;
  r = 0; t = c;
  while (t != NULL)  {
    t = t->tl;
    r = r + 1;
  }
}
```

Motivating Example

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)  {
    t = t->tl;
    r = r + 1;
  }
} [list(c,cdata) *
   (r = length(cdata))]
```

Motivating Example

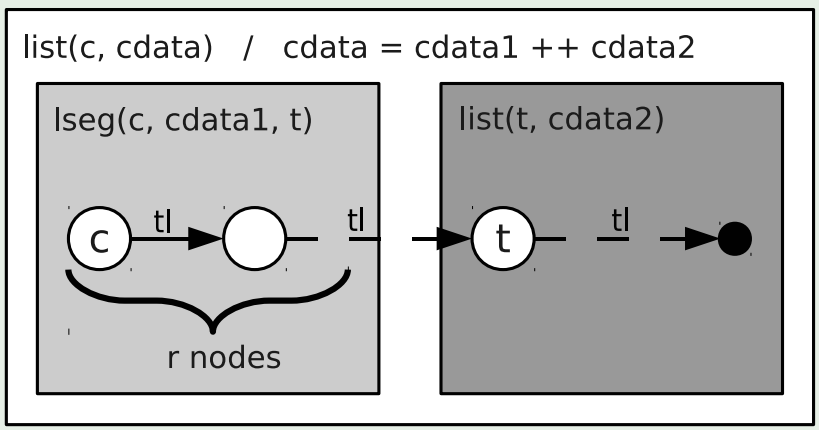
Length of a single linked list

```
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))]
```

Motivating Example

Length of a single linked list




```
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))]
```

Motivation II

- two implementations of an algorithm
- same interface
- same procedure specification
- recursive one is simple
- iterative one has complicated loop-invariant

Motivation II

- two implementations of an algorithm
- same interface
- same procedure specification
- recursive one is simple
- iterative one has complicated loop-invariant

Why???

Motivation II

- two implementations of an algorithm
- same interface
- same procedure specification
- recursive one is simple
- iterative one has complicated loop-invariant

Loop invariant does not support local reasoning!

Local reasoning

- Separation logic supports local reasoning
- Hoare triples can be extended by an arbitrary frame

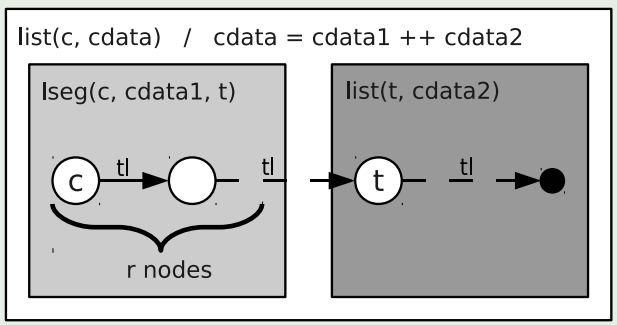
$$\frac{\{P\}\ prog\ \{Q\}}{\{P * R\}\ prog\ \{Q * R\}}$$

- this rule is essential for separation logic
- it helps simplifying specifications
- only important parts need to be mentioned explicitly
- everything not mentioned is implicatly not unchanged

Motivating Example

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))]
```

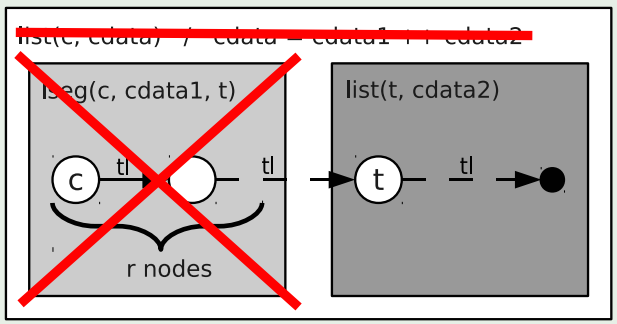



```
list_length_iter(r;c)
    [list(c,cdata)] {
  local t;
  r = 0; t = c;
  while (t != NULL) [
     ∃cdata1 cdata2.
     lseg(c, cdata1, t) *
     list(t, cdata2) *
     (r = length(cdata1)) *
     (cdata = cdata1 +
              cdata2)] {
    t = t->tl;
    r = r + 1;
  }
} [data_list(c,cdata) *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

Motivating Example

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;
  }
} [data_list(c,cdata) *
   (r = length(cdata))]
```

Inference rule for while-loops

Inference rule for loops (partial correctness)

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

Informal justification / induction on number of iterations:

$$\{I\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I\} \quad \Leftarrow$$

$$\{\neg c \wedge I\}\ \{\neg c \wedge I\}\ \wedge$$
$$\{c \wedge I\}\ p; \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I\} \quad \Leftarrow$$

$$\{c \wedge I\}\ p; \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I\} \quad \Leftarrow$$

$$\forall prog.\quad \{I\}\ prog\ \{\neg c \wedge I\} \longrightarrow$$
$$\{c \wedge I\}\ p;\ prog\ \{\neg c \wedge I\} \quad \Leftarrow$$

$$\{c \wedge I\}\ p\ \{I\}$$

Inference rule for while-loops

Inference rule for loops (partial correctness)

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

Informal justification / induction on number of iterations:

$\{I\}$ **while** c **do** p **done** $\{\neg c \wedge I\}$ $\Leftarrow$

$\{\neg c \wedge I\}\ \{\neg c \wedge I\}\ \wedge$
$\{c \wedge I\}\ p;$ **while** c **do** p **done** $\{\neg c \wedge I\}$ $\Leftarrow$

$\{c \wedge I\}\ p;$ **while** c **do** p **done** $\{\neg c \wedge I\}$ $\Leftarrow$

$\forall prog.\quad \{I\}\ prog\ \{\neg c \wedge I\} \longrightarrow$
$\{c \wedge I\}\ p;\ prog\ \{\neg c \wedge I\}$ $\Leftarrow$

$\{c \wedge I\}\ p\ \{I\}$

Families of Specifications

- try to utilise local reasoning
- however, keep it general, consider general families of specifications

$$\forall x.\ \{P(x)\}\ prog\ \{Q(x)\}$$

- local reasoning can be represented by such a family of specifications

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

Extended Inference rule for while-loops

Informal justification / induction on number of iterations:

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

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

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

$$\forall x, prog.\quad \big(\forall y.\ \{I(y)\}\ prog\ \{\neg c \wedge I(y)\}\big) \longrightarrow$$
$$\{c \wedge I(x)\}\ p;\ prog\ \{\neg c \wedge I(x)\} \quad \Leftarrow$$

$$\forall x.\ \exists y.\quad \{c \wedge I(x)\}\ p\ \{I(y)\} \wedge$$
$$\{\neg c \wedge I(y)\}\ \{\neg c \wedge I(x)\}$$

Extended Inference rule for while-loops

Informal justification / induction on number of iterations:

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

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

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

$$\forall x, prog.\quad \big(\forall y.\ \{I(y)\}\ prog\ \{\neg c \wedge I(y)\}\big) \longrightarrow$$
$$\{c \wedge I(x)\}\ p;\ prog\ \{\neg c \wedge I(x)\} \quad \Leftarrow$$

$$\forall x.\ \exists y.\quad \{c \wedge I(x)\}\ p\ \{I(y)\} \wedge$$
$$\{\neg c \wedge I(y)\}\ \{\neg c \wedge I(x)\}$$

Extended Inference Rule

Extended Inference rule for loops (partial correctness)

$$\frac{\forall x, prog.\ \left(\forall y.\ \{I(y)\}\ prog\ \{\neg c \wedge I(y)\}\right) \longrightarrow \{c \wedge I(x)\}\ p;\ prog\ \{\neg c \wedge I(x)\}}{\forall x.\ \{I(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I(x)\}}$$

- this extended rule allows to handle families of specifications
- local reasoning can be handled
- for each iteration a different specification (instantiation of x) can be used

$\Rightarrow$ $I(x)$ is not really an invariant any more

$\Rightarrow$ let's use proper pre- and post-conditions

Loop Specifications I

Informal justification / induction on number of iterations:

$$\forall x.\ \{P(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{Q(x)\} \quad \Leftarrow$$

$$\forall x.\ \{\neg c \wedge P(x)\}\ \{Q(x)\}\ \wedge$$
$$\forall x.\ \{c \wedge P(x)\}\ p; \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{Q(x)\} \quad \Leftarrow$$

$$\forall x, \mathit{prog}.\quad \big(\forall y.\ \{P(y)\}\ \mathit{prog}\ \{Q(y)\}\big) \longrightarrow \{c \wedge P(x)\}\ p;\ \mathit{prog}\ \{Q(x)\} \quad \wedge$$
$$\forall x.\ \{\neg c \wedge P(x)\}\ \{Q(x)\}$$

Loop Specifications I

Informal justification / induction on number of iterations:

$$\forall x.\ \{P(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done};\ p_2\ \{Q(x)\} \quad \Leftarrow$$

$$\forall x.\ \{\neg c \wedge P(x)\}\ p_2\ \{Q(x)\} \wedge$$
$$\forall x.\ \{c \wedge P(x)\}\ p; \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done};\ p_2\ \{Q(x)\} \quad \Leftarrow$$

$$\forall x, prog.\quad \left(\forall y.\ \{P(y)\}\ prog\ \{Q(y)\}\right) \longrightarrow$$
$$\{c \wedge P(x)\}\ p;\ prog\ \{Q(x)\} \quad \wedge$$
$$\forall x.\ \{\neg c \wedge P(x)\}\ p_2\ \{Q(x)\}$$

Loop Specifications II

Loop specification rule

$$\forall x, \mathit{prog}.\ \left(\forall y.\ \{P(y)\}\ \mathit{prog}\ \{Q(y)\}\right) \longrightarrow \{c \wedge P(x)\}\ p;\ \mathit{prog}\ \{Q(x)\}$$

$$\frac{\forall x.\ \{\neg c \wedge P(x)\}\ p_2\ \{Q(x)\}}{\forall x.\ \{P(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done};\ p_2\ \{Q(x)\}}$$

- loop specifications allow local reasoning
- p_2 and using pre- / post-conditions not essential, but useful

Example I

Length of a single linked list

```
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))]
```

```
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))]
```

Comments on Example I

- the new specification uses local reasoning
- specification becomes much simpler
- there are no partial data-structures

Comments on Example I

- the new specification uses local reasoning
- specification becomes much simpler
- there are no partial data-structures
- loop specifications lead to a different way of thinking
- often they are more *natural*
- very similar to Eric Hehner's specified blocks

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.

Example II

Incrementing Array Elements

```
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)]
```

Example II - 2

Loop invariant

$$\exists data_2.\quad \mathtt{array}(\mathtt{a}, \mathtt{n}, data_2) * \big(\forall x.\quad x < \mathtt{i} \implies data_2[x] = data[x] + 1\big) * \big(\forall x.\quad \mathtt{i} \le x < \mathtt{n} \implies data_2[x] = data[x]\big)$$

Loop specification (without local reasoning)

pre: $\mathtt{array}(\mathtt{a}, \mathtt{n}, data)$

post: $\exists data_2.\quad \mathtt{array}(\mathtt{a}, \mathtt{n}, data_2) * \big(\forall x.\quad x < \mathtt{old}(\mathtt{i}) \implies data_2[x] = data[x]\big) * \big(\forall x.\quad \mathtt{old}(\mathtt{i}) \le x < \mathtt{n} \implies data_2[x] = data[x] + 1\big)$

Loop invariants state what the loop has already done, loop specification state what will be done.

Example II - 2

Loop invariant

$$\exists data_2.\quad \mathtt{array}(\mathtt{a}, \mathtt{n}, data_2) \ast \left(\forall x.\quad x < \mathtt{i} \implies data_2[x] = data[x] + 1\right) \ast \left(\forall x.\quad \mathtt{i} \leq x < \mathtt{n} \implies data_2[x] = data[x]\right)$$

Loop specification (with local reasoning)

pre: $\mathtt{array}(\mathtt{a} + \mathtt{i}, \mathtt{n} - \mathtt{i}, data)$

post: $\mathtt{array}(\mathtt{a} + \mathrm{old}(\mathtt{i}), \mathtt{n} - \mathrm{old}(\mathtt{i}), \mathrm{map}\ (+1)\ data)$

Loop invariants state what the loop has already done, loop specification state what will be done.

Example III

Filtering a list

```
list_filter(l;x) [list(l,data)] {
  local y, z, e;
  y = l; z = NULL;
  while (y != NULL) {
    e = y->dta;
    if (e == x) {
      if(y == l) {
        l = y->tl; dispose y; y = l;
      } else {
        e = y->tl; z->tl = e;
        dispose y; y = z->tl;
      }
    } else {
      z = y; y = y->tl;
    }
  }
} [list(l, filter x from data)]
```

Example III - 2

Loop invariant

if $(\mathtt{y} = \mathtt{l})$ **then**

$$\exists data_1.\ (data = (\text{some } \mathtt{xs}) + data_1) * \text{list}(\mathtt{l}, data_1)$$

else

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

Expressing the inverse of a partial application of filter is ugly!

Example III - 3

Loop specification (no local reasoning)

pre: $\mathrm{list}(\mathtt{y}, \mathit{data}_2) \ *$
if $(\mathtt{y} \neq \mathtt{l})$ **then** $\mathrm{lseg}(\mathtt{l}, \mathit{data}, \mathtt{z}) \ * \ (\mathtt{z} \mapsto [\mathtt{tl} : \mathtt{y}, \mathtt{dta} : \mathit{zdate}])$ **else** emp

post: **if** $(\mathrm{old}(\mathtt{y}) = \mathrm{old}(\mathtt{l}))$ **then** $\mathrm{list}(\mathtt{l}, \mathrm{filtered}\ \mathit{data}_2)$
else $\mathrm{list}(\mathtt{l}, \mathit{data} + \mathit{zdate} + \mathrm{filtered}\ \mathit{data}_2)$

Example IV

Deleting the minimum key of a binary search tree

```
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))]
```

Example IV - 2

- loop is used to find node holding minimal key
- after loop
  - p point to this minimal node
  - pp points to parent of minimal node
  - tree is unmodified
- that's hard to express: partial trees
- usual solution: magic wand (complicated)
- Holfoot can't handle magic-wand
- loop specification of loop + deleting code after loop used
- no partial data-structure needed!

Conclusion I

- loop specifications are an additional inference rule for while-loops
- they allow to use local reasoning
- similar to Eric Hehner's specified blocks
- different view on loops:
  - loop invariant: what has been done
  - loop specification: what will be done
- simpler, more natural specifications
- already useful for classical Hoare logic

Conclusion II

- local reasoning adds extra value
- can be used to avoid partial datastructures
- allows Holfoot to handle interesting, additional examples
- more examples are available on Holfoot's web-page `http://holfoot.heap-of-problems.org`
  - reversing a single-linked list
  - copying a single-linked list
  - removing an element from a single-linked list
  - tree traversal
  - binary search
  - quicksort
  - ...