Automated Software Verification with Implicit Dynamic Frames

Malte Schwerhoff, ETH Zürich/KU Leuven
5th December 2013, Leuven
A long time ago, in a galaxy far, far away ...
Outline

1. Implicit Dynamic Frames
2. Our Tool Chain
3. Supporting Magic Wands
Heap-Dependent Expressions

- SL: Points-to relations and logical variables
  
  requires \(x.f \rightarrow ?v \&\& v > 0\)

- IDF: Access predicates and heap access
  
  requires \(acc(x.f) \&\& x.f > 0\)

- IDF-Assertions must be \textit{self-framing}, i.e., only talk about locations to which access is requested
  
  requires \(acc(x.f) \&\& x.f > 0\) \(\checkmark\) self-framing
  \[\times\] not self-framing
  \[\times\] neither (technical reason)

(SL-assertions are self-framing by design)
Example

Separation Logic

```plaintext
method inc(c: Cell)
  requires c.f |-> ?v && v > 0
  ensures  c.f |-> v + 1
{ c.f = c.f + 1 }
```

Implicit Dynamic Frames

```plaintext
method inc(c: Cell)
  requires acc(c.f) && c.f > 0
  ensures  acc(c.f) && c.f == old(c.f) + 1
{ c.f = c.f + 1 }
```

old(e) evaluates to the value e had in the pre-heap of the method call
SL: Abstract Predicates

predicate Cell(c: Cell; v: Int) { c.f |-> v}

method inc(c: Cell)
  requires Cell(c, ?v) && v > 0
  ensures Cell(c, v + 1)
{
  open Cell(c, v)
  c.f = c.f + 1
  close Cell(c, v + 1)
}

- **Ghost statements** open/close guide the verifier; erased at runtime

- Opening a predicate instance means *consuming* the instance and *producing* its body (closing is the inverse operation)
Data Abstraction

**IDF: Abstract Predicates and Pure Functions**

**Predicate**

```plaintext
predicate Cell(c: Cell) { acc(c.f) }
```

**Function**

```plaintext
function get(c: Cell): Int
    requires acc(Cell(c))
{ unfolding Cell(c) in c.f }
```

**Method**

```plaintext
method inc(c: Cell)
    requires acc(Cell(c)) && get(c) > 0
    ensures acc(Cell(c)) && get(c) == old(get(c)) + 1
{ unfold Cell(c)
    c.f = c.f + 1
    fold Cell(c)
}
```

*Ghost expression* unfolding \( P \) in \( e \) makes the body of \( P \) temporarily available.
Predicates and Functions

- SL: Predicates have *in-* and *out-parameters*; out-parameters are uniquely determined by the in-parameters and the predicate body

Separation Logic

\[
\text{predicate Cell}(c : \text{Cell}; v : \text{Int}) = c.f \implies v
\]

- IDF: Predicates have in-parameters, functions replace the out-parameters

Implicit Dynamic Frames

\[
\text{predicate Cell}(c : \text{Cell}) \{ \text{acc}(c.f) \}
\]

\[
\text{function get}(c : \text{Cell}) : \text{Int}
\]

\[
\text{requires acc(Cell}(c))
\]

\[
\{ \text{unfolding Cell}(c) \text{ in } c.f \} \]
Predicates and Functions

Implicit Dynamic Frames

predicate Node(n: Node) {
    acc(n.val) && acc(n.nxt) && (n.nxt != null ==> acc(Node(n.nxt)))
}

function length(n: Node): Int
    requires acc(Node(n))
{ unfolding Node(n) in 1 + (n.nxt == null ? 0 : length(n.nxt)) } 

function elems(n: Node): Seq[Int]
    requires acc(Node(n))
{ unfolding Node(n) in n.val :: (n.nxt == null ? Nil : elems(n.nxt)) } 

- SL: length and elems could be out-parameters
  - Adding additional out-parameters later on potentially entails lots of code changes
  - Adding new functions in subclasses feels “natural”

- IDF: Separate heap shape description from abstractions
Functions

- IDF: Functions can be used in code, too!

Implicit Dynamic Frames

if (length(node) > 2) ...

- SL: Predicate arguments and methods are needed

Separation Logic

predicate Cell(c: Cell; v: Int) { c.f \rightarrow v}

method length(c: Cell): Int
  requires Cell(c, ?v)
  ensures  result == v
{
  open Cell(c, v)
  return c.f
  close Cell(c, v)
}
Outline

1. Implicit Dynamic Frames
2. Our Tool Chain
3. Supporting Magic Wands
SIL and Silicon

- SIL is an *intermediate verification language*; programs with specifications can be encoded in SIL
  - Objects, fields, methods, if-else, loops
  - Simple: rudimentary type system (primitives + Ref), no inheritance (yet?), no concurrency
  - IDF-based assertion language; fractional permissions; sequences, sets, multisets; quantifiers; custom theories

- Silicon is a symbolic-execution-based verifier for SIL

- Z3 is used to discharge Boolean proof obligations
Chalice is a research language for concurrency
- Objects, fields, loops, ...
- Fork-join concurrency
- Communication via channels (message passing)
- Locking with lock invariants
- Deadlock-avoidance

Scala is a OO+FP hybrid language for the JVM
- ... with crazily many features
- Only translated basics, including
  - `val x = e`  
    (≈ final fields in Java)
  - `lazy val x = e`  
    (evaluated on first read)
Verification of SIL code

- SIL
  - verified by
  - Carbon
    - encode in
      - Boogie (Microsoft)
        - encode in
          - Z3 (Microsoft)
            - queries

Verification Condition Generation
Symbolic Execution
Short deviation: VCG vs SE

Query prover once with full information (VeriCool)

Program \rightarrow \text{Verifier} \rightarrow \text{WPs} \rightarrow \text{Prover}

Program read by \text{Verifier} calculates one weakest precondition per method given to \text{Prover}

Query prover often with limited information (VeriFast)

Program \rightarrow \text{Verifier} \rightarrow \text{Symbolic State } \sigma \rightarrow \text{Prover}

Program read by \text{Verifier}, \text{Verifier} maintains \text{Symbolic State } \sigma \text{ used by } \text{Prover}

symbolically execute every path through every method

\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \sigma_4 \rightarrow \sigma_5

query prover at every step if next statement is executable
Verification of SIL code

Parallel approach allows experimenting with new features and encodings; it helps uncovering weaknesses or performance problems.
Outline

1. Implicit Dynamic Frames
2. Our Tool Chain
3. Supporting Magic Wands
   (joint work with Alexander J. Summers)
- Boolean implication \( A \Rightarrow B \)
  “If \( A \) holds in the current state, then \( B \) also holds“

**Modus Ponens:** \( A \wedge (A \Rightarrow B) \models B \)

- Separating implication: \( A \rightarrow^* B \)
  “If \( A \) is added to the current state, then \( B \) also holds“

  (Kind of) **Modus Ponens:** \( A \ast (A \rightarrow^* B) \models B \)

- \( A \rightarrow^* B \) Can be read as an exchange promise
  “If \( A \) is given up, then \( B \) is guaranteed to hold“

**Magic Wands**
Semantics of the Wand:

\[ h \models A \quad \Rightarrow \quad \forall h' \perp h \cdot (h' \models A \Rightarrow h \cup h' \models B) \]

- Quantification over states, hence typically not supported in automated verifiers

- Used in proofs by hand, for example, when verifying linked lists with views (generalised iterators)

- The promise-interpretation lends itself to specifying partial data structures, for example,
  
  “Give up a list segment and you’ll get back the whole linked list“
Iterating Over Recursively-Defined Data Structures

```plaintext
var val: Int
var next: Ref

predicate List(ys: Ref) {
    acc(ys.val) && acc(ys.next) && (ys.next != null => acc(List(ys.next)))
}

function sum_rec(ys: Ref): Int
    requires acc(List(ys))
{    unfolding List(ys) in ys.val + (ys.next == null ? 0 : sum_rec(ys.next))
}

method sum_it(ys: Ref) returns (sum: Int)
    /* Iteratively compute the sum of the linked list s.t.
    * the result equals sum_rec(ys)
    */
```
Iterating Over Recursively-Defined Data Structures

method sum_it(ys: Ref) returns (sum: Int)
  requires ys != null && acc(List(ys))
  ensures acc(List(ys)) && sum == old(sum_rec(ys))
{
  var xs: Ref := ys  /* Pointer to the current node in the list */
  sum := 0  /* Sum computed so far */

  while (xs != null)
    invariant xs != null ==> acc(List(xs))
    invariant sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
    {
      var zs: Ref := xs
      unfold List(xs)
      sum := sum + xs.val
      xs := xs.next
      /* ??? */
    }
  }
}

How to bookkeep permissions to the “list seen so far”?
Our Solution

```cpp
var xs: Ref := ys  /* Pointer to the current node in the list */
sum := 0  /* Sum computed so far */

/* Short-hands to keep the specifications concise */
define A xs != null => acc(List(xs))
define B acc(List(ys))
```

Here, $A \rightsquigarrow B$ reflects the promise
“If you give up the current tail of the list ($xs$), then you’ll get back the whole list ($ys$)”
Our Solution

define A xs != null ==> acc(List(xs))
define B acc(List(ys))

package A --* B

while (xs != null)
  invariant (xs != null ==> acc(List(xs))) && A --* B
  invariant sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
{
  wand w := A --* B /* Give magic wand instance the name w */

  var zs: Ref := xs
  unfold List(xs)
  sum := sum + xs.val
  xs := xs.next

  package A --* folding List(zs) in applying w in B

apply A --* B
Our Solution

define A xs != null ==> acc(List(xs))
define B acc(List(ys))

package A -->* B Establish wand

while (xs != null)
    invariant (xs != null ==> acc(List(xs))) && A -->* B
    invariant sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
{
    wand w := A -->* B /* Give magic wand instance the name w */

    var zs: Ref := xs
    unfold List(xs)
    sum := sum + xs.val
    xs := xs.next

    package A -->* folding List(zs) in applying w in B
}

apply A -->* B
Our Solution

define A xs !\equiv \text{null} \implies \text{acc}(\text{List}(xs))
define B \text{acc}(\text{List}(ys))

package A \text{--*} B

while (xs !\equiv \text{null})
    \text{invariant} (xs !\equiv \text{null} \implies \text{acc}(\text{List}(xs))) 
    \text{&}
    \text{invariant} \text{sum} = \text{old}(\text{sum}_{\text{rec}}(ys)) - (xs = \text{null} ? 0 : \text{sum}_{\text{rec}}(xs))
{
    \text{wand} w := A \text{--*} B /* \text{Give magic wand instance the name} w */

    \text{var} zs: \text{Ref} := xs
    \text{unfold} \text{List}(xs)
    \text{sum} := \text{sum} + xs.\text{val}
    xs := xs.\text{next}

    \text{package} A \text{--* folding} \text{List}(zs) \text{in applying} w \text{in} B
}

apply A \text{--*} B
Our Solution

```plaintext
define A xs != null ==> acc(List(xs))
define B acc(List(ys))

package A --* B

while (xs != null)
    invariant (xs != null ==> acc(List(xs))) && A --* B
    invariant sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
{
    wand w := A --* B /* Give magic wand instance the name w */

    var zs: Ref := xs
    unfold List(xs)
    sum := sum + xs.val
    xs := xs.next
```

[Update wand]

```plaintext
package A --* folding List(zs) in applying w in B
apply A --* B
```
Our Solution

define A xs != null ==> acc(List(xs))
define B acc(List(ys))

package A --* B

while (xs != null)
  invariant (xs != null ==> acc(List(xs))) && A --* B
  invariant sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
{
  wand w := A --* B /* Give magic wand instance the name w */

  var zs: Ref := xs
  unfold List(xs)
  sum := sum + xs.val
  xs := xs.next

  package A --* folding List(zs) in applying w in B
}

apply A --* B Use wand
Lifecycle

- Lifecycle of wand and predicate instances are similar
  - Created (packaged/folded)
  - Passed around (loop invariants, postconditions)
  - Destroyed (applied/unfolded)

- Unfolding a predicate gives assumptions about the heap

- Sound, because the permissions that *frame* these assumptions are consumed when the predicate is folded (and the assumptions are checked)

- These permissions are the *footprint* of a predicate

- What is the footprint of a wand?
Footprints

- Examples
  - true →∗ acc(x.f) | acc(x.f)
  - acc(x.f) →∗ acc(x.f) | emp
  - acc(x.f, 1/3) →∗ acc(x.f, 1/1) | acc(x.f, 2/3)
  - acc(x.f) →∗ acc(x.g) | acc(x.g)

- The footprint of A →∗ B is the delta between A and B

- Consumed when A →∗ B is packaged and produced when A →∗ B is applied
Footprints and Assumptions

- Examples

  - \( \sigma: acc(x.f) \&\& x.f = 1 \)
    
    package true \( \rightarrow \ast \) acc(x.f) \&\& x.f = 1  

  - \( \sigma: acc(x.f) \&\& x.f = 1 \)
    
    package acc(x.f) \( \rightarrow \ast \) acc(x.f) \&\& x.f = 1

  - \( \sigma: acc(x.f) \&\& x.f = 1 \&\& acc(x.g) \&\& x.g = 2 \)
    
    package acc(x.f) \&\& x.f =2
    
    \( \rightarrow \ast \) acc(x.f) \&\& acc(x.g) \&\& x.f = x.g

- When checking assumptions of the RHS, use the assumptions from the LHS, and those of the current state if framed by the footprint

- Claim: sound, regardless of the computed footprint
Footprints and Assumptions

- Circularity problem
  - Footprint is determined by permissions requested by the RHS (and not provided by the LHS)
  - Permissions might be conditionally requested (if-then-else)
  - Guards of these conditionals might be determined by the current heap
  - Assumptions about the current heap can only be used if framed by the footprint
  - ... which we currently try to compute :-(

Our Solution

- Compute footprint in parallel with checking the RHS
  - Setup
    - Let $\sigma_{\text{curr}}$ be the current heap
    - Let $\sigma_{\text{foot}}$ be the initially empty footprint state
    - Produce the LHS into $\text{emp}$ to get $\sigma_{\text{lhs}}$
  - Algorithm
    - Consume permissions requested by the RHS from $\sigma_{\text{lhs}}$, and **only** from $\sigma_{\text{curr}}$ if $\sigma_{\text{lhs}}$ does not provide sufficient permissions
    - If taken from $\sigma_{\text{curr}}$, move effected permissions (and move/copy assumptions) into $\sigma_{\text{foot}}$
    - Check assumptions made by the RHS in the combination of $\sigma_{\text{lhs}}$ and $\sigma_{\text{foot}}$
define A xs != null ==> acc(List(xs))
define B acc(List(ys))

package A --* B

while (xs != null)
  invariant (xs != null ==> acc(List(xs))) && A --* B
  invariant sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
{
  wand w := A --* B /* Give magic wand instance the name w */

  var zs: Ref := xs
  unfold List(xs)
  sum := sum + xs.val
  xs := xs.next

  package A --* folding List(zs) in applying w in B

apply A --* B
Packaging Wands with Ghost Operations

```java
var zs := xs; unfold List(xs); xs := xs.next

package List(xs) -->*

List(xs) --* List(ys)
```

```
var zs := xs; unfold List(xs); xs := xs.next

package List(xs) -->*

List(xs) --* List(ys)
```

```
var zs := xs; unfold List(xs); xs := xs.next

package List(xs) -->*

List(xs) --* List(ys)
```
Packaging Wands with Ghost Operations

- Interaction with the footprint: delta between permissions produced/consumed by ghost operations

- Prover hints only, created wand instance is \( \text{List(xs)} \rightarrow \ast \text{List(ys)} \)

- Nicely blend into SIL, which has \textit{unfolding} already
define A xs != null ==> acc(List(xs))
define B acc(List(ys))

package A --* B

while (xs != null)
  invariant (xs != null ==> acc(List(xs))) && A --* B
  invariant sum == old(sum_rec(ys)) - (xs == null ? 0 : sum_rec(xs))
{
  wand w := A --* B /* Give magic wand instance the name w */

  var zs: Ref := xs
  unfold List(xs)
  sum := sum + xs.val
  xs := xs.next

  package A --* folding List(zs) in applying w in B
}

apply A --* B
Conclusion

- Implicit Dynamic Frames
  - Allows for (relatively) nice specifications
  - Simplifies contrasting VCG and SE

- Intermediate Verification Language SIL
  - Potential to encode other languages into it looks promising
  - VCG and SE backends facilitate experiments

- Magic Wands
  - Useful for specifying partial data structures
  - Lightweight support that nicely integrates into IDF
Future Work

- Tool Chain
  - Polish it (documentation, IDE, debugger)
  - Release it (and merge various branches)
  - Continue Scala2Sil

- Magic Wands
  - Demonstrate other applications
  - More examples
  - Support in VCG?
Questions?

malte.schwerhoff@inf.ethz.ch
http://www.pm.inf.ethz.ch/