Dafny

Boogie translation

Dafny:

```
method Main() {
  TestAdd(3, 180);
  TestAdd(3, -180);
  TestAdd(0, 1);
  [...]
```

Boogie:

```
implementation [...].Main()
returns ($_reverifyPost: bool) {
  var $_Frame:
     <beta>[ref,Field beta]bool;
  var x##i (0..8): int;
  x##0 := LitInt(3);
  y##0 := LitInt(180);
  call Intra[...]TestAdd(x##0, y##0);
[...]
```

Boogie translation

```
method Add(x: int, y: int) returns (r: int)
  ensures r == x+y;
```

```
procedure Impl[...]AddAdd(
    x#0: int, y#0: int)
    returns (
        r#0: int, $_reverifyPost: bool);
```

Boogie translation

```
method Add [...]
{
  r := x;
  if (y < 0) {
      var n := y;
      while (n != 0)
      invariant r == x+y-n && 0 <= -n;
      {
      r := r - 1;
      n := n + 1;
      }
  } else [...]
```

```
implementation Impl[...]Add[...] {
 r#0 := x#0;
 if (y#0 < 0)
      {
      n#0_0 := y#0;
      $PreLoopHeap$loop#0_0 := $Heap;
      $decr_init$loop#0_00 := (if n#0_0 <= LitInt
(0) then 0 - n#0_0 else n#0_0 - 0);
      havoc $w$loop#0_0;
      while (true)
```

Boogie translation

```
method Add [...]
{
  r := x;
  if (y < 0) {
        var n := y;
        while (n != 0)
        invariant r == x+y-n && 0 <= -n;
        {
        r := r - 1;
        n := n + 1;
        }
  } else [...]
```

```
while (true)
        free invariant $w$loop#0_0 ==> r#0 == x#0
+ y#0 - n#0_0 ==> true;
        invariant $w$loop#0_0 ==> r#0 == x#0 +
y#0 - n#0_0;
        invariant $w$loop#0_0 ==> LitInt(0) <= 0 -
n#0_0;
        free invariant (if n#0_0 <= LitInt(0) then 0 -
n#0_0 else n#0_0 - 0) <= $decr_init$loop#0_00
        free invariant [heap related];
```

Boogie translation

```
method Add [...]
{
  r := x;
  if (y < 0) {
        var n := y;
        while (n != 0)
        invariant r == x+y-n && 0 <= -n;
        {
        r := r - 1;
        n := n + 1;
        }
} else [...]
```

```
if (!$w$loop#0_0) { [...] assume false; }
if (n#0_0 == 0) { break; }
$decr$loop#0_00 := (if n#0_0 <= LitInt(0) then 0
- n#0_0 else n#0_0 - 0);
r#0 := r#0 - 1;
n#0_0 := n#0_0 + 1;
assert 0 <= $decr$loop#0_00 || (if n#0_0 <=
LitInt(0) then 0 - n#0_0 else n#0_0 - 0) ==
$decr$loop#0_00;
assert (if n#0_0 <= LitInt(0) then 0 - n#0_0 else
n#0_0 - 0) < $decr$loop#0_00;
```

Control Flow

```
implementation Impl[...]Add[...] {
  r#0 := x#0;
 if (y#0 < 0)
        {
        n#0_0 := y#0;
        $PreLoopHeap$loop#0_0 := $Heap;
        $decr_init$loop#0_00 := (if n#0_0 <=
LitInt(0) then 0 - n#0_0 else n#0_0 - 0);
        havoc $w$loop#0_0;
        while (true)
```

```
implementation Impl[...]Add[...] {
anon0:
 r#0 := x#0;
 goto anon23_Then, anon23_Else;
anon23_Then:
  n#0_0 := y#0;
  $PreLoopHeap$loop#0_0 :=
$Heap;
  $decr_init$loop#0_00 := (if [...]);
  havoc $w$loop#0_0;
  goto anon24_LoopHead;
```

Control Flow

```
while (true)
        free invariant $w$loop#0_0 ==> r#0 == x#0
+ y#0 - n#0_0 ==> true;
        invariant $w$loop#0_0 ==> r#0 == x#0 +
y#0 - n#0_0;
        invariant $w$loop#0_0 ==> LitInt(0) <= 0 -
n#0_0;
        free invariant (if n#0_0 <= LitInt(0) then 0 -
n#0_0 else n#0_0 - 0) <= $decr_init$loop#0_00
        free invariant [heap related];
```

```
anon24_LoopHead:
    assume $w$loop#0_0 ==> rasdf#0 == x#0 + y#0
- n#0_0 ==> true;
    assert $w$loop#0_0 ==> rasdf#0 == x#0 + y#0 -
n#0_0;
    assert $w$loop#0_0 ==> LitInt(0) <= 0 - n#0_0;
    assume (if n#0_0 <= LitInt(0) then 0 - n#0_0 else
n#0_0 - 0) <= $decr_init$loop#0_00
    assume [heap related];
goto anon24_LoopDone, anon24_LoopBody;
```

Control Flow

```
if (!$w$loop#0_0)
    { [...] assume false; }
if (n#0_0 == 0) { break; }
$decr$loop#0_00 := (if n#0_0 <= LitInt(0) then 0 - n#0_0 else n#0_0 - 0);
r#0 := r#0 - 1;
n#0_0 := n#0_0 + 1;
assert 0 <= $decr$loop#0_00 || (if n#0_0 <= LitInt(0) then 0 - n#0_0 else n#0_0 - 0) == $decr$loop#0_00;
assert (if n#0_0 <= LitInt(0) then 0 - n#0_0 else n#0_0 - 0) < $decr$loop#0_00;
```

```
anon24_LoopBody:
  assume true;
  goto anon25_Then, anon25_Else;
anon25_Then:
  assume {:partition} !$w$loop#0_0;
  goto [...] (assume false);
anon25_Else:
  assume {:partition} $w$loop#0_0;
  goto anon28_Then, anon28_Else;
anon28_Then:
        assume {:partition} n#0_0 == 0;
        return;
```

Control Flow

```
if (!$w$loop#0_0) { [...] assume false; }
if (n#0_0 == 0) { break; }
$decr$loop#0_00 := (if n#0_0 <= LitInt(0) then 0 - n#0_0 else n#0_0 - 0);
r#0 := r#0 - 1;
n#0_0 := n#0_0 + 1;
assert 0 <= $decr$loop#0_00 || (if n#0_0 <= LitInt(0) then 0 - n#0_0 else n#0_0 - 0) == $decr$loop#0_00;
assert (if n#0_0 <= LitInt(0) then 0 - n#0_0 else n#0_0 - 0) < $decr$loop#0_00;
```

```
anon28_Else:
  assume {:partition} n#0_0 != 0;
  goto anon11;
anon11:
  $decr$loop#0_00 := (if n#0_0 <= LitInt(0) then 0 - n#0_0 else n#0_0 - 0);
[...]
goto anon24_LoopHead;
```

It’s a kind of maaagic

(

let ((anon28_Else_correct (=> (! (and %lbl%+11822 true) :lblpos +11822) (=> (and (and (not (= |n#0_0@0| 0)) (= |$decr$loop#0_00@1| (ite (<= |n#0_0@0| (LitInt 0)) (- 0 |n#0_0@0|) (- |n#0_0@0| 0)))) (and (= |rasdf#0@3| (- |rasdf#0@2| 1)) (= |n#0_0@1| (+ |n#0_0@0| 1)))) (and (! (or %lbl%@28679 (or (<= 0 |$decr$loop#0_00@1|) (= (ite (<= |n#0_0@1| (LitInt 0)) (- 0 |n#0_0@1|) (- |n#0_0@1| 0)) |$decr$loop#0_00@1|))) :lblneg @28679) (=> (or (<= 0 |$decr$loop#0_00@1|) (= (ite (<= |n#0_0@1| (LitInt 0)) (- 0 |n#0_0@1|) (- |n#0_0@1| 0)) |$decr$loop#0_00@1|)) (and (! (or %lbl%@28708 (< (ite (<= |n#0_0@1| (LitInt 0)) (- 0 |n#0_0@1|) (- |n#0_0@1| 0)) |$decr$loop#0_00@1|)) :lblneg @28708) (=> (< (ite (<= |n#0_0@1| (LitInt 0)) (- 0 |n#0_0@1|) (- |n#0_0@1| 0)) |$decr$loop#0_00@1|) (=> (=> (= |rasdf#0@3| (- (+ |x#0| |y#0|) |n#0_0@1|)) true) (and (! (or %lbl%@28746 (=> |$w$loop#0_0@0| (= |rasdf#0@3| (- (+ |x#0| |y#0|) |n#0_0@1|)))) :lblneg @28746) (=> (=> |$w$loop#0_0@0| (= |rasdf#0@3| (- (+ |x#0| |y#0|) |n#0_0@1|))) (! (or %lbl%@28761 (=> |$w$loop#0_0@0| (<= (LitInt 0) (- 0 |n#0_0@1|)))) :lblneg @28761))))))))))))

+50lines

)

Heap

```
class C_A {
  var X : int;
  method Set(Y : int)
  modifies this;
  ensures X == Y;
  {
        X := Y;
  }
}
```

```
method Sum(X: C_A, Y: C_A) returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

Heap

```
method Sum(X: C_A, Y: C_A)
    returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
procedure Impl$$_module.__default.Sum(
  X#0: ref
    where
      $Is(X#0, Tclass._module.C__A()) &&
      $IsAlloc(
        X#0, Tclass._module.C__A(), $Heap),
  Y#0: ref
    where
      $Is(Y#0, Tclass._module.C__A()) &&
      $IsAlloc(
        Y#0, Tclass._module.C__A(), $Heap))
```

Heap

```
method Sum(X: C_A, Y: C_A)
    returns (sum: C_A)
  requires X != null;
  requires Y != null;
  modifies X;
  ensures fresh(sum);
  ensures sum.X == old(X.X) + Y.X;
  ensures X.X == Y.X;
  {
    sum := new C_A.Set(X.X + Y.X);
    X.Set(Y.X);
  }
```

```
returns (
sum#0: ref
  where
      $Is(sum#0, Tclass._module.C__A()) &&
      $IsAlloc(
          sum#0, Tclass._module.C__A(), $Heap),
$_reverifyPost: bool);
```

Heap

```
method Sum(X: C_A, Y: C_A)
    returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
// user-defined preconditions
requires X#0 != null;
requires Y#0 != null;
modifies $Heap, $Tick;
// user-defined postconditions
ensures sum#0 != null && !read(old($Heap),
sum#0, alloc);
```

Heap

```
method Sum(X: C_A, Y: C_A)
    returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
ensures
  read($Heap, sum#0, _module.C__A.X)
  == read(old($Heap), X#0, _module.C__A.X)
  + read($Heap, Y#0, _module.C__A.X);
```

```
ensures
  read($Heap, X#0, _module.C__A.X) ==
  read($Heap, Y#0, _module.C__A.X);
```

Heap

```
method Sum(X: C_A, Y: C_A)
    returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
// frame condition
  free ensures (forall<alpha> $o: ref, $f: Field
alpha ::
  { read($Heap, $o, $f) }
  $o != null && read(old($Heap), $o, alloc)
        ==> read($Heap, $o, $f) == read(old
($Heap), $o, $f)
|| $o == X#0);
```

Heap

```
method Sum(X: C_A, Y: C_A)
    returns (sum: C_A)
requires X != null;
requires Y != null;

modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
implementation Impl$$_module.__default.Sum
(X#0: ref, Y#0: ref) returns (sum#0: ref,
$_reverifyPost: bool)
{
  var $_Frame: <beta>[ref,Field beta]bool;
  var $nw: ref;

  var Y##0: int;
  var Y##1: int;
```

Heap

```
method Sum(X: C_A, Y: C_A) returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
$_Frame := (lambda<alpha> $o: ref, $f: Field alpha ::
  $o != null && read($Heap, $o, alloc)
     ==> $o == X#0);
$_reverifyPost := false;
havoc $nw;
assume
  $nw != null &&
  !read($Heap, $nw, alloc) &&
  dtype($nw) == Tclass._module.C__A();
$Heap := update($Heap, $nw, alloc, true);
assume $IsGoodHeap($Heap);
```

Heap

```
method Sum(X: C_A, Y: C_A) returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

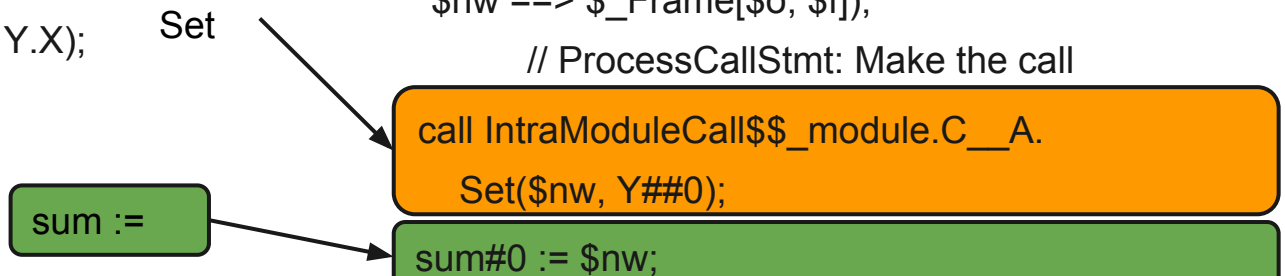
Modifies Set <=
Modifies Sum

```
assert X#0 != null; assert Y#0 != null;
        // TrCallStmt: Before ProcessCallStmt
Y##0 :=
   read($Heap, X#0, _module.C__A.X) +
   read($Heap, Y#0, _module.C__A.X);
        // ProcessCallStmt: CheckSubrange
assert (forall<alpha> $o: ref, $f: Field alpha ::
  $o != null && read($Heap, $o, alloc) && $o ==
$nw ==> $_Frame[$o, $f]);
        // ProcessCallStmt: Make the call
call IntraModuleCall$$_module.C__A.
    Set($nw, Y##0);
sum#0 := $nw;
```

Heap

```
method Sum(X: C_A, Y: C_A) returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
assert X#0 != null; assert Y#0 != null;
        // TrCallStmt: Before ProcessCallStmt
Y##0 :=
   read($Heap, X#0, _module.C__A.X) +
   read($Heap, Y#0, _module.C__A.X);
        // ProcessCallStmt: CheckSubrange
assert (forall<alpha> $o: ref, $f: Field alpha ::
  $o != null && read($Heap, $o, alloc) && $o ==
$nw ==> $_Frame[$o, $f]);
        // ProcessCallStmt: Make the call
call IntraModuleCall$$_module.C__A.
   Set($nw, Y##0);
sum#0 := $nw;
```



Heap

```
method Sum(X: C_A, Y: C_A) returns (sum: C_A)
requires X != null;
requires Y != null;
modifies X;
ensures fresh(sum);
ensures sum.X == old(X.X) + Y.X;
ensures X.X == Y.X;
{
  sum := new C_A.Set(X.X + Y.X);
  X.Set(Y.X);
}
```

```
assert X#0 != null; assert Y#0 != null;
        // ProcessCallStmt: CheckSubrange
Y##1 := read($Heap, Y#0, _module.C__A.X);
assert (forall<alpha> $o: ref, $f: Field alpha ::
  $o != null && read($Heap, $o, alloc) &&
  $o == X#0 ==> $_Frame[$o, $f]);
        // ProcessCallStmt: Make the call
call IntraModuleCall$$_module.C__A.
  Set(X#0, Y##1);
```

(40 lines +

($HeapSucc $Heap@1 $Heap@2))) (and (! (or %lbl%@24666 (and (not (= $nw@0 null)) (not (U_2_bool (MapType1Select $Heap@@3 $nw@0 alloc))))) :lblneg @24666) (=> (and (not (= $nw@0 null)) (not (U_2_bool (MapType1Select $Heap@@3 $nw@0 alloc)))) (and (! (or %lbl%@24681 (= (U_2_int (MapType1Select $Heap@2 $nw@0 _module.C__A.X)) (+ (U_2_int (MapType1Select $Heap@@3 |X#0@@2| _module.C__A.X)) (U_2_int (MapType1Select $Heap@2 |Y#0@@1| _module.C__A.X))))) :lblneg @24681) (=>

(= (U_2_int (MapType1Select $Heap@2 $nw@0 _module.C__A.X)) (+ (U_2_int (MapType1Select $Heap@@3 |X#0@@2| _module.C__A.X)) (U_2_int (MapType1Select $Heap@2 |Y#0@@1| _module.C__A.X))))

(! (or %lbl%@24705 (= (U_2_int (MapType1Select $Heap@2 |X#0@@2| _module.C__A.X)) (U_2_int (MapType1Select $Heap@2 |Y#0@@1| _module.C__A.X)))) :lblneg @24705))))))))))))))))))))))))

(let ((PreconditionGeneratedEntry_correct@@2 (=> (! (and %lbl%+24092 true) :lblpos +24092) (=> ($IsGoodHeap $Heap@@3) (=> (and ($Is |X#0@@2| Tclass._module.C__A) ($IsAlloc |X#0@@2| Tclass._module.C__A $Heap@@3)) (=> (and (and (and ($Is |Y#0@@1| Tclass._module.C__A) ($IsAlloc |Y#0@@1| Tclass._module.C__A $Heap@@3)) (and ($Is |sum#0@@0| Tclass._module.C__A) ($IsAlloc |sum#0@@0| Tclass._module.C__A $Heap@@3))) (and (and (= 0 $ModuleContextHeight) (= 1 $FunctionContextHeight)) (and (not (= |X#0@@2| null)) (not (= |Y#0@@1| null))))) anon0_correct@@2)))))) PreconditionGeneratedEntry_correct@@2))

```
(=
  (U_2_int
    (MapType1Select
      $Heap@2 $nw@0 _module.C__A.X))
```

sum.X

```
(+
  (U_2_int
    (MapType1Select
      $Heap@@3 |X#0@@2| _module.C__A.X))
  (U_2_int
    (MapType1Select
      $Heap@2 |Y#0@@1| _module.C__A.X))))
```

X.X

Y.X

Triggers

```
type Heap = <alpha>[ref,Field alpha]alpha;
axiom
  (forall<alpha>
       h: Heap, r: ref,
       f: Field alpha, x: alpha ::
  { update(h, r, f, x) }
  $IsGoodHeap(update(h, r, f, x))
      ==> $HeapSucc(h, update(h, r, f, x)));
```

Flags

dafny /print:boogie-output.bpl dafny-input.dfy

boogie
  /printInstrumented
  /print:boogie-output.bpl
  /prover-log:smt-output.txt
  boogie-input.dfy