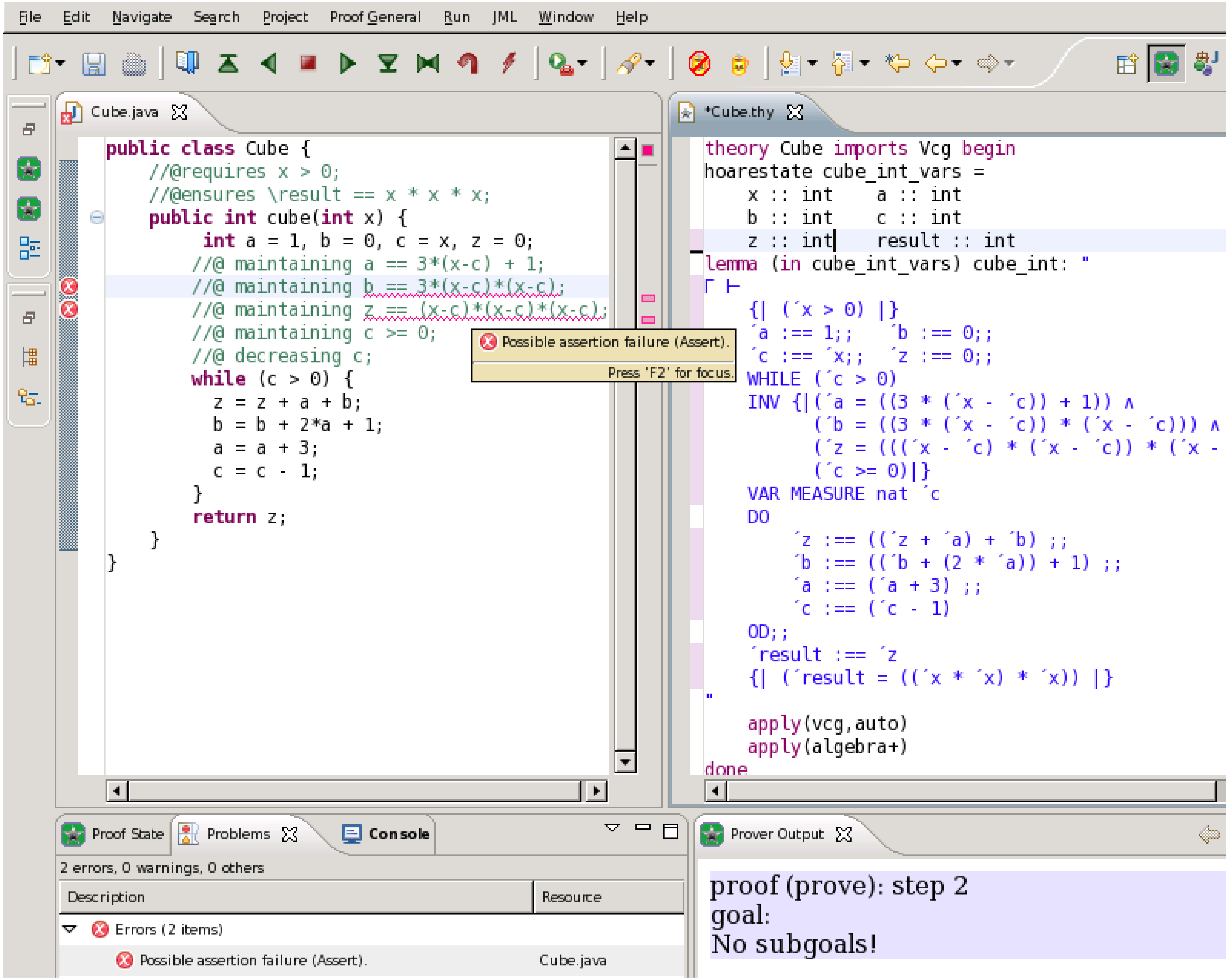
Total Correctness of Recursive Functions Using JML4 FSPV

George Karabotsos, Patrice Chalin,
Perry R. James, Leveda Giannas
Dependable Software Research Group

Dept. of Computer Science and Software Engineering
Concordia University, Montréal, Canada
{g_karab, chalin, perry, leveda}@dsrg.org

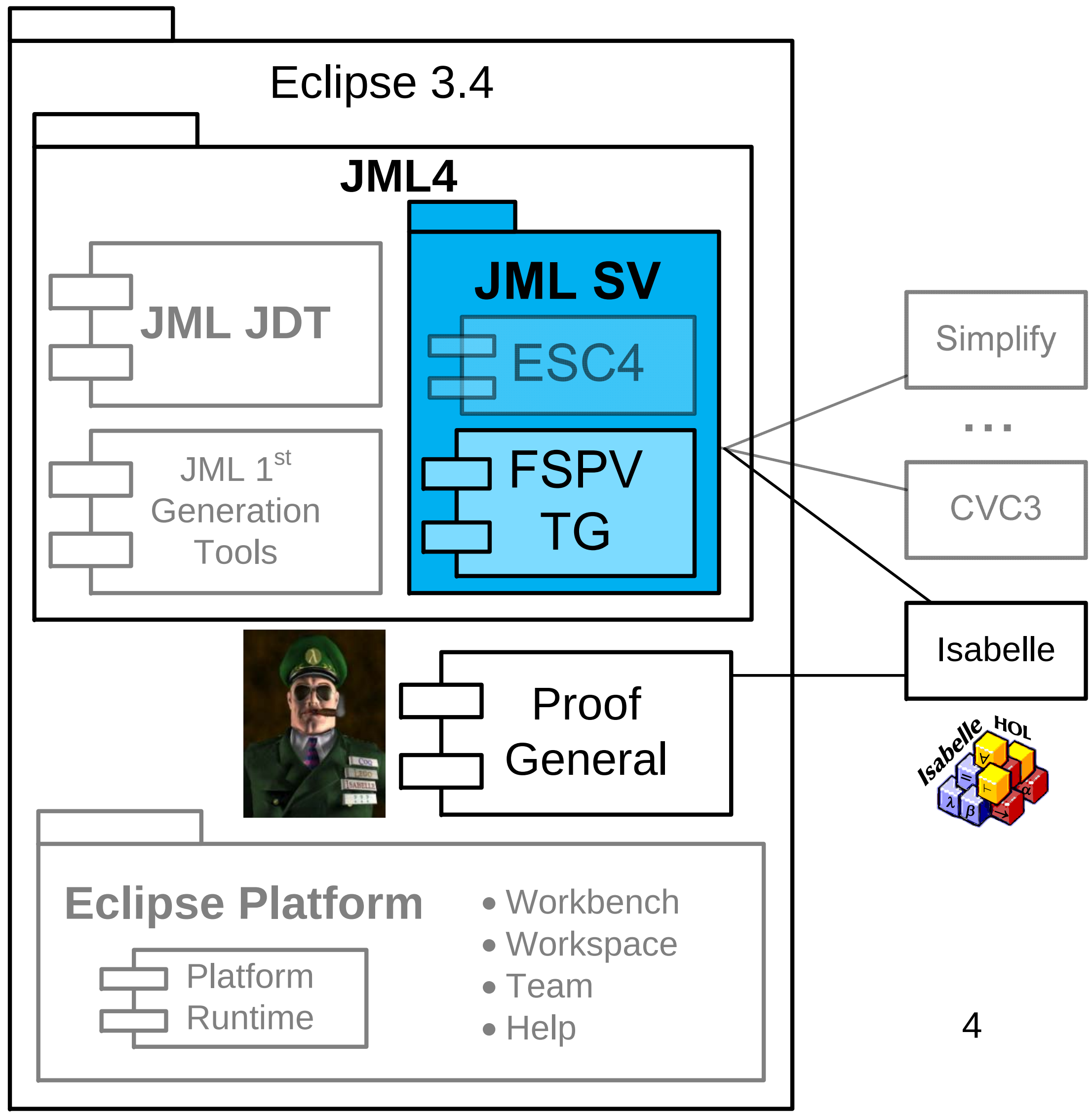
Research Context

- Java Modeling Language (JML)
- Program Verification
  - Aging 1st generation tools
- Integrated Verification Environment
  - JML4 …


File Edit Navigate Search Project Proof General Run JML Window Help
Cube.java
public class Cube {
//@requires x > 0;
//@ensures \result == x * x * x;
public int cube(int x) {
int a = 1, b = 0, c = x, z = 0;
//@ maintaining a == 3*(x-c) + 1;
//@ maintaining b == 3*(x-c)*(x-c);
//@ maintaining z == (x-c)*(x-c)*(x-c);
//@ maintaining c >= 0;
//@ decreasing c;
while (c > 0) {
z = z + a + b;
b = b + 2*a + 1;
a = a + 3;
c = c - 1;
}
return z;
}
}
Possible assertion failure (Assert).
Press 'F2' for focus.
*Cube.thy
theory Cube imports Vcg begin
hoarestate cube_int_vars =
x :: int a :: int
b :: int c :: int
z :: int result :: int
lemma (in cube_int_vars) cube_int: "
Γ ⊢
{| (´x > 0) |}
´a :== 1;; ´b :== 0;;
´c :== ´x;; ´z :== 0;;
WHILE (´c > 0)
INV {|(´a = ((3 * (´x - ´c)) + 1)) ∧
(´b = ((3 * (´x - ´c)) * (´x - ´c))) ∧
(´z = (((´x - ´c) * (´x - ´c)) * (´x -
(´c >= 0)|}
VAR MEASURE nat ´c
DO
´z :== ((´z + ´a) + ´b) ;;
´b :== ((´b + (2 * ´a)) + 1) ;;
´a :== (´a + 3) ;;
´c :== (´c - 1)
OD;;
´result :== ´z
{| (´result = ((´x * ´x) * ´x)) |}
"
apply(vcg,auto)
apply(algebra+)
done
Proof State
Problems
Console
2 errors, 0 warnings, 0 others
Description
Resource
Errors (2 items)
Possible assertion failure (Assert).
Cube.java
Prover Output
proof (prove): step 2
goal:
No subgoals!

Eclipse-based IVE

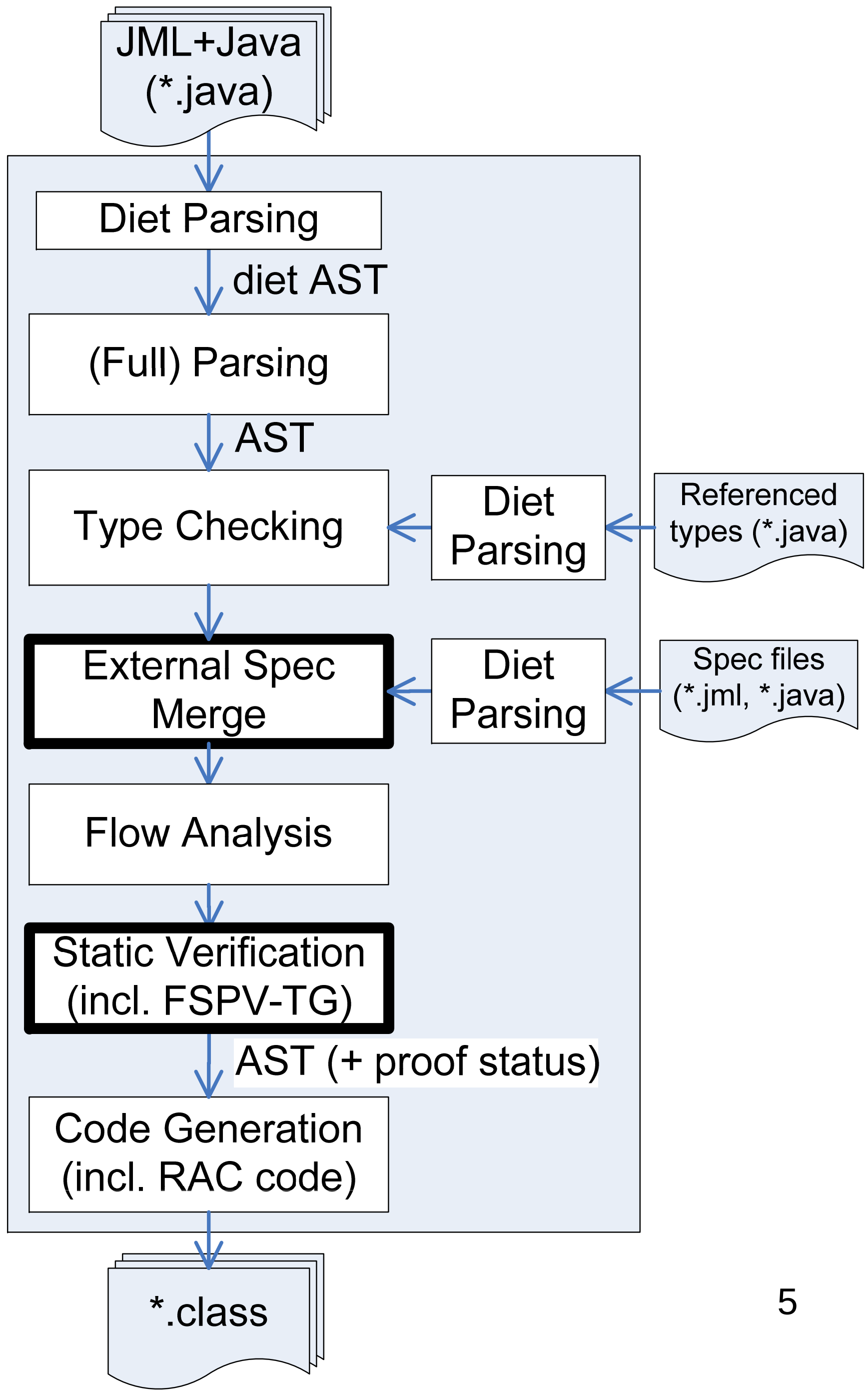
- Next-Generation Research Platform
- Integrates existing tools
  - RAC (jml & jmlc)
  - ESC (ESC/Java2)
- ESC4 & FSPV-TG



JML4 Extends Eclipse JDT

Java Development Tooling (JDT)

- JDT/JML4 Core Phases



Isabelle

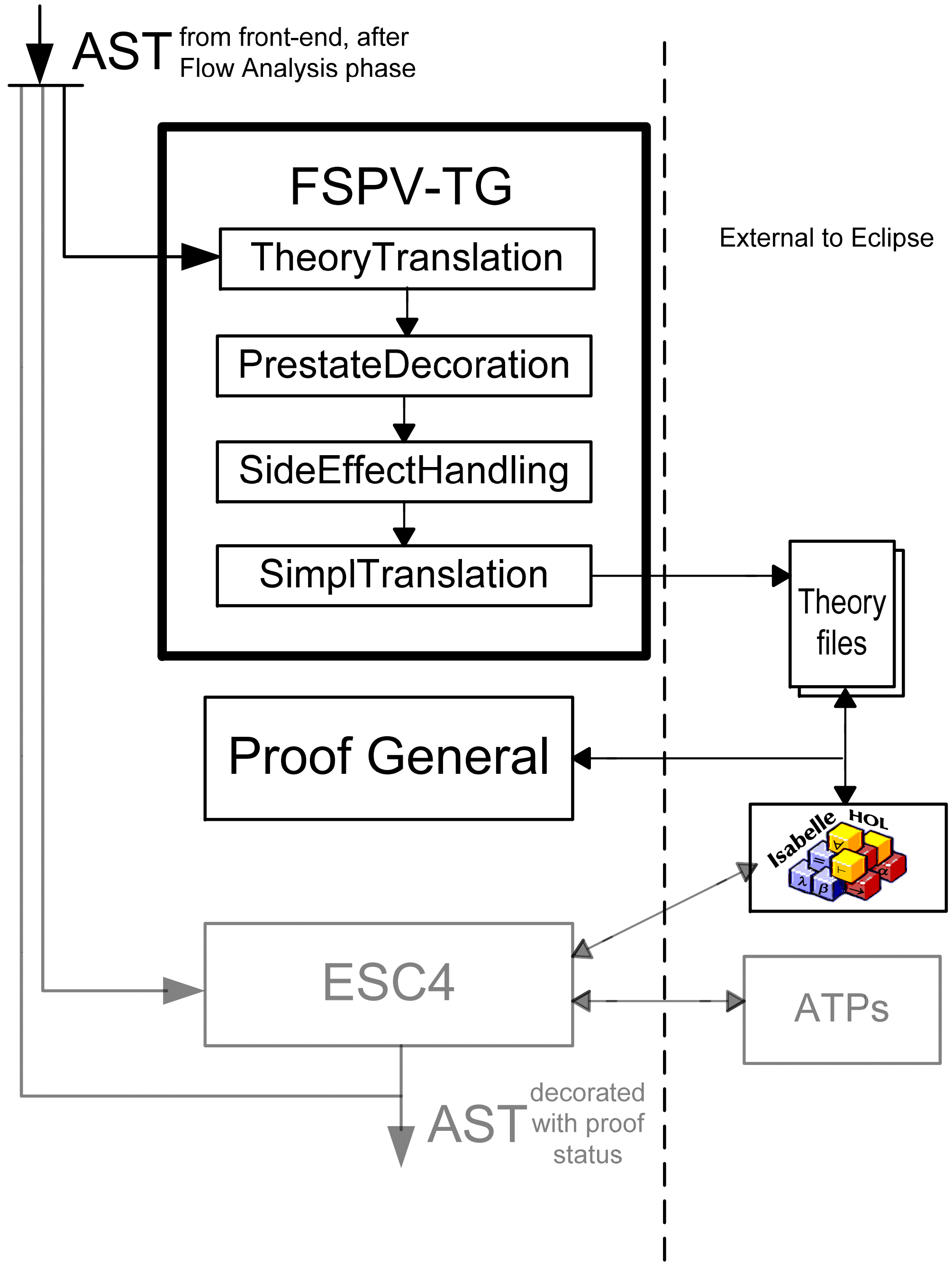
- Theorem prover framework
  - Isabelle/Pure (meta-logic)
  - classical reasoner
  - simplifier
- Provides machinery for defining new logics
  - existing logics can be extended
  - syntax can be added as sugar

Isabelle/Simpl

- Extends Isabelle/HOL
- Hoare Rules for
  - Partial & Total Correctness
- VCs generated within Simpl
  - through vcg and vcg_step tactics
- Proven Sound and Complete
- Expressive
  - Global and local variables, exceptions, abrupt termination, procedures, breaks out of loops, procedures, references and heap

Dataflow in FSPV-TG

- Translate JML to Hoare triples
- Store info for \old
- Remove side effects from expressions
- Generate Simpl



McCarthy's 91 Function

```
public class McCarthy {
 //@ requires n >= 0;
 //@ ensures \result == (100 < n ? n-10 : 91);
 //@ measured_by 101 - n;
 public static int f91(int n) {
  if(100 < n)
   return n - 10;
  else
   return f91(f91(n + 11));
 }
}
```

McCarthy's 91 Simpl Theory

```
theory McCarthy imports Vcg begin
  procedures
    McCarthy_f91_int(n::int | result'::int)
  "IF 100 < ´n
   THEN
     ´result' :== ´n - 10
   ELSE
     CALL McCarthy_f91_int(´n + 11) ≫ n1.
       ´result' :== CALL McCarthy_f91_int(n1)
   FI"
  lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
    "∀n σ. Γ ⊢\<^sub>t
       {|σ. n=´n ∧ ´n≥0|}
       ´result' :== PROC McCarthy_f91_int(´n)
       {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
   apply(hoare_rule HoareTotal.ProcRec1
    [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
   apply(vcg)
   apply(auto)
  done
end
```

McCarthy’s 91 Simpl Theory

```
theory McCarthy imports Vcg begin
  procedures
    McCarthy_f91_int(n::int | result'::int)
  "IF 100 < ´n
   THEN
     ´result' :== ´n - 10
   ELSE
     CALL McCarthy_f91_int(´n + 11) ≫ n1.
       ´result' :== CALL McCarthy_f91_int(n1)
   FI"
  lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
    "∀n σ. Γ ⊢\<^sub>t
       {|σ. n=´n ∧ ´n≥0|}
       ´result' :== PROC McCarthy_f91_int(´n)
       {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
   apply(hoare_rule HoareTotal.ProcRec1
    [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
   apply(vcg)
   apply(auto)
  done
end
```

Method Translation

Total Correctness Proof

Components of an Isabelle/Simpl Theory

- States

```
hoarestate vars =
    x::τ1
    …
```

- Procedures

```
procedures
   N (x::τ1, y::τ2, …| z::τ3)
   where v::τ4 … in B
```

- Hoare Tuples

$\Gamma, \Theta \vdash$ {|*P*|} *B* {|*Q*|}, {|*R*|}

$\Gamma, \Theta \vdash_t$ {|*P*|} *B* {|*Q*|}, {|*R*|}

Hoarestate

- Define global and local variables
- Statespaces
  - Modeled as a function from abstract names to abstract values
  - Organizes
    - distinctness of names and
    - projection/injection of concrete values into the abstract one.
- Locales
  - Support modular reasoning
  - Allows multiple inheritance of other locales
  - Allows for renaming components

McCarthy 91 Function

```
public static int f91(int n) {
    if(100 < n)
        return n - 10;
    else
        return f91(f91(n + 11));
}
```

```
procedures
  McCarthy_f91_int(n::int | result'::int)
"IF 100 < ´n
 THEN
   ´result' :== ´n - 10
 ELSE
   CALL McCarthy_f91_int(´n + 11) ≫ n1.
     ´result' :== CALL McCarthy_f91_int(n1)
 FI"
```

McCarthy 91 Function

```
public static int f91(int n) {
    if(100 < n)
        return n - 10;
    else
        return f91(f91(n + 11));
}
```

Input

Name

Output

```
procedures
 McCarthy_f91_int(n::int | result'::int)
"IF 100 < ´n
 THEN
   ´result' :== ´n - 10
 ELSE
   CALL McCarthy_f91_int(´n + 11) ≫ n1.
     ´result' :== CALL McCarthy_f91_int(n1)
 FI"
```

McCarthy 91 Function

```
public static int f91(int n) {
    if(100 < n)
            return n - 10;
    else
            return f91(f91(n + 11));
}
```

```
procedures
 McCarthy_f91_int(n::int | result'::int)
"IF 100 < ´n
 THEN
   ´result' :== ´n - 10
 ELSE
   CALL McCarthy_f91_int(´n + 11) ≫ n1.
     ´result' :== CALL McCarthy_f91_int(n1)
 FI"
```

Input

Output

Name

Simpl Variable

Binder Variable

Inner Rec.

Outer Rec.

Generate for Simpl

- a hoarestate(`McCarthy_f91_int_impl`)
  - statespace, locale
- Functions
  - Copying the actual to formal parameters
  - Updating global variables
  - Copying the formal result parameter

McCarthy 91- Proving Correctness

```
//@ requires n >= 0;
//@ ensures \result == (100<n ? n-10 : 91);
//@ measured_by 101 - n;
public static int f91(int n)
```

```
lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
     {|σ. n=´n ∧ ´n≥0|}
     ´result' :== PROC McCarthy_f91_int(´n)
     {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
 apply(hoare_rule HoareTotal.ProcRec1
  [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
 apply(vcg)
 apply(auto)
done
```

McCarthy 91- Proving Correctness

```
//@ requires n >= 0;
//@ ensures \result == (100<n ? n-10 : 91);
//@ measured_by 101 - n;
public static int f91(int n)
```

locale

Name

```
lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
     {|σ. n=´n ∧ ´n≥0|}
     ´result' :== PROC McCarthy_f91_int(´n)
     {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
 apply(hoare_rule HoareTotal.ProcRec1
  [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
 apply(vcg)
 apply(auto)
done
```

McCarthy 91- Proving Correctness

```
//@ requires n >= 0;
//@ ensures \result == (100<n ? n-10 : 91);
//@ measured_by 101 - n;
public static int f91(int n)
```

locale

Name

```
lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
     {|σ. n=´n ∧ ´n≥0|}
     ´result' :== PROC McCarthy_f91_int(´n)
     {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
 apply(hoare_rule HoareTotal.ProcRec1
  [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
 apply(vcg)
 apply(auto)
done
```

Pre

Statement

Post

McCarthy 91- Proving Correctness

```
//@ requires n >= 0;
//@ ensures \result == (100<n ? n-10 : 91);
//@ measured_by 101 - n;
public static int f91(int n)
```

```
lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
     {|σ. n=´n ∧ ´n≥0|}
     ´result' :== PROC McCarthy_f91_int(´n)
     {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
 apply(hoare_rule HoareTotal.ProcRec1
  [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
 apply(vcg)
 apply(auto)
done
```

prestate

locale

Name

Pre

Statement

Post

McCarthy 91- Proving Correctness

```
//@ requires n >= 0;
//@ ensures \result == (100<n ? n-10 : 91);
//@ measured_by 101 - n;
public static int f91(int n)
```

prestate | locale | Name

```
lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
     {|σ. n=´n ∧ ´n≥0|}
     ´result' :== PROC McCarthy_f91_int(´n)
     {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
 apply(hoare_rule HoareTotal.ProcRec1
  [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
 apply(vcg)
 apply(auto)
done
```

Pre | Statement | Post | VC generator | measure | auto

Fibonacci

```
//@ public static native int fib_spec(int
  n);
//@ requires n>=0;
//@ ensures \result ==  fib_spec(n);
//@ measured_by n;
public static /*@ pure */ int fib(int n) {
  if(n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Fibonacci

```
function fib_spec :: "int ⇒ int" where
"fib_spec n =
  (if n = 0 then 0 else
    (if n=1 then 1 else
      (if n < 0 then arbitrary
        else (fib_spec (n - 1)) + (fib_spec (n - 2)))))"
by(pat_completeness, auto)
termination by (relation "measure (λn. nat n)",auto)

lemma (in Fibonacci_fib_int_impl) Fibonacci_fib_int_spec:
"∀n σ. Γ ⊢\<^sub>t
     {|σ. ´n=n ∧ ´n≥0|}
     ´result' :== PROC Fibonacci_fib_int(´n)
     {|´result'=fib_spec(n)|}"
apply(hoare_rule HoareTotal.ProcRec1
  [where r="measure (λ (s,p). nat \<^bsup>s\<^esup>n )"])
by(vcg,auto)
```

Fibonacci

```
class Fibonacci {
  //@ requires n>=0;
  //@ ensures \result == (n==0)? 0 : (n==1) ? 1
  //@  : fib_spec(n-1)+fib_spec(n-2);
  //@ measured_by n;
  //@ public static pure model
  //@                      int fib_spec(int n);

  //@ requires n>=0;
  //@ ensures \result == fib_spec(n);
  //@ measured_by n;
  public static /*@ pure */ int fib(int n) {
    ...
```

Ackermann

```
//@ public static native int ack_spec(int n);
//@ requires n >= 0 && m >= 0 ;
//@ ensures \result == ack_spec(n,m);
public static int ack(int n, int m) {
  if(n == 0)
    return m + 1;
  else if(m == 0)
    return ack(n-1, m);
  else
    return ack(n-1, ack(n, m-1));
}
```

Ackermann

```
lemma (in Ackermann_ack_int_int_impl) Ackermann_ack_int_int_spec:
  "∀n m σ. Γ ⊢\<^sub>t
     {|σ. n=´n ∧ ´n≥0 ∧ m=´m ∧ ´m≥0 |}
     ´result' :== PROC Ackermann_ack_int_int(´n, ´m)
     {|´result' =  (ack_spec n m) |}"
 apply(hoare_rule HoareTotal.ProcRec1
    [where r="measures [λ(s,p). nat \<^bsup>s\<^esup>n,
                              \<lambda>(s,p). nat \<^bsup>s\<^esup>m]"] )
 apply((auto|vcg)+,case_tac "nat n",auto,case_tac "nat n",auto)
 by (case_tac "nat m",auto)
```

Milestones

- FSPV TG
  - Supporting functional subset of JML+Java
- Study the adequacy of Isabelle/Simpl
  - Non-recursive Programs
    - Cube, Factorial, Summation
  - Total Correctness of Recursive Programs
    - Factorial, Fibonacci, McCarthy's 91, Ackermann
  - Classes
    - Point

Related Work

- LOOP
  - VerifiCard project (Industry)
- JACK
  - Banking Application (Academia)
- Krakatoa/Why

Comparison Table

	LOOP	JACK	Krakatoa/Why	FSPV-TG Simpl
Maintained	✗	✗	✓	✓
Open Source	✗	✓	✓	✓
Proven Sound	✓	✗	✓	✓
Proven Complete	✗	✗	✗	✓
Above two proofs done	in PVS	N/A	by hand	in Isabelle
VC generation done in prover	✗	✗	✗	✓
Termination of Recursive Functions	✗	✗	✗	✓

Future Work

- Update the translator to reflect current state.
- Case Study
- Object Oriented Support
  - Inheritance
  - Behavioral Subtyping
- Additional Language Elements
  - Exceptions
  - Loops with break and continue statements
- JML
  - Revise measured_by clause (see Ackermann)

Thank you!

George Karabotsos, Patrice Chalin, Perry R. James, Leveda Giannas
Dependable Software Research Group

Dept. of Computer Science and Software Engineering
Concordia University, Montréal, Canada
{g_karab, chalin, perry, leveda}@dsrg.org

Fields + Memory

```
hoarestate globals_memory =
 alloc::"ref list"
 free::nat
hoarestate globals_Point = globals_memory +
 XCoord :: "ref ⇒ int"
 YCoord :: "ref ⇒ int"
definition sz where "sz ≡ 2::nat"
```

Total Correctness Proof for the f91 Function

```
//@ requires n >= 0;
//@ ensures \result == (100<n ? n-10 : 91);
//@ measured_by 101 - n;
public static int f91(int n)
```

```
lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
     {|σ. n=´n ∧ ´n≥0|}
     ´result' :== PROC McCarthy_f91_int(´n)
     {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
```

Constructor

```
procedures (imports globals_Point)
 Point_Point_int_int(x::int, y::int | result'::ref)
 "´result':==NEW sz [´XCoord:==0,´YCoord:==0];;
  ´result'→´XCoord :== ´x;;
  ´result'→´YCoord :== ´y"
```

- Constructors defined as a regular procedure
- Allocate memory using the `NEW` operator
  - Providing a size (sz) and
  - A list of with the fields and their initialization
- Assignments of input values to fields

Method Calls

```
procedures (imports globals_Point)
 Point_tester_Point(P::ref|result'::ref)
 "´P :== CALL Point_Point_int_int(10,11);;
   CALL Point_move_int_int(´P,1,0);;
  ´result' :== ´P"
```

Point Class

```
Public class Point
 public int XCoord;
 public int YCoord;

 public Point(int x, int y) {
  XCoord = x;
  YCoord = y;
 }
 //@ensures XCoord==\old(XCoord)+dx;
 //@ensures YCoord==\old(YCoord)+dy;
 public void move(int dx, int dy) {
  XCoord += dx;
  YCoord += dy;
 }
 //@ requires P == null;
 //@ ensures \result != null;
 //@ ensures \result.XCoord == 11;
 //@ ensures \result.YCoord == 11;
 public static Point tester(Point P){
  P = new Point(10,11);
  P.move(1,0);
  return P;
 }
}
```

Point Class Simpl Theory

```
theory Point imports HeapList Vcg
begin
 hoarestate globals_memory =
  alloc::"ref list"
  free::nat
 hoarestate globals_Point = globals_memory +
  XCoord :: "ref ⇒ int"
  YCoord :: "ref ⇒ int"
 definition sz where "sz ≡ 2::nat"

 procedures (imports globals_Point)
  Point_Point_int_int(x::int, y::int | result'::ref)
  "´result':==NEW sz [´XCoord:==0,´YCoord:==0];;
   ´result'→´XCoord :== ´x;;
   ´result'→´YCoord :== ´y"
 lemma (in Point_Point_int_int_impl) Point_Point_int_int_spec:
  "∀x y. Γ⊢\<^sub>t
        {|´x = x ∧ ´y = y ∧ sz ≤ ´free|}
        ´result' :== PROC Point_Point_int_int(´x,´y)
        {|´result' ≠ Null ∧
          ´result'→´XCoord = x ∧ ´result'→´YCoord = y|}"
 by(vcg,auto)

 procedures (imports globals_Point)
  Point_move_int_int(this::ref, dx::int, dy::int)
  "´this→´XCoord :== ´this→´XCoord + ´dx;;
   ´this→´YCoord :== ´this→´YCoord + ´dy"
 lemma (in Point_move_int_int_impl) Point_move_int_int_spec:
  "∀dx dy x y σ. Γ⊢\<^sub>t
    {|σ.´this→´XCoord = x ∧ ´this→´YCoord = y ∧
        ´dx = dx ∧ ´dy = dy ∧ ´this ≠ Null |}
    PROC Point_move_int_int(´this, ´dx, ´dy)
    {|´this =  \<^bsup>σ\<^esup>this ∧
      ´this→´XCoord =  x + dx ∧ ´this→´YCoord =  y + dy|}"
 by(vcg, auto)

 procedures (imports globals_Point)
  Point_tester_Point(P::ref|result'::ref)
  "´P :== CALL Point_Point_int_int(10,11);;
    CALL Point_move_int_int(´P,1,0);;
   ´result' :== ´P"
 lemma (in Point_tester_Point_impl) Point_tester_Point_spec:
  "∀x y. Γ⊢\<^sub>t
    {| sz ≤ ´free ∧ ´P = Null |}
    ´result' :== PROC Point_tester_Point(´P)
    {| ´result' ≠ Null ∧
       ´result'→´XCoord = 11 ∧ ´result'→´YCoord = 11 |}"
 by(vcg, auto)
end
```

Point Class Simpl Theory

fields

Constructor

move

tester

```
theory Point imports HeapList Vcg
begin
 hoarestate globals_memory =
  alloc::"ref list"
  free::nat
 hoarestate globals_Point = globals_memory +
  XCoord :: "ref ⇒ int"
  YCoord :: "ref ⇒ int"
 definition sz where "sz ≡ 2::nat"

 procedures (imports globals_Point)
  Point_Point_int_int(x::int, y::int | result'::ref)
  "´result':==NEW sz [´XCoord:==0,´YCoord:==0];;
   ´result'→´XCoord :== ´x;;
   ´result'→´YCoord :== ´y"
 lemma (in Point_Point_int_int_impl) Point_Point_int_int_spec:
  "∀x y. Γ⊢\<^sub>t
        {|´x = x ∧ ´y = y ∧ sz ≤ ´free|}
        ´result' :== PROC Point_Point_int_int(´x,´y)
        {|´result' ≠ Null ∧
          ´result'→´XCoord = x ∧ ´result'→´YCoord = y|}"
 by(vcg,auto)

 procedures (imports globals_Point)
  Point_move_int_int(this::ref, dx::int, dy::int)
  "´this→´XCoord :== ´this→´XCoord + ´dx;;
   ´this→´YCoord :== ´this→´YCoord + ´dy"
 lemma (in Point_move_int_int_impl) Point_move_int_int_spec:
  "∀dx dy x y σ. Γ⊢\<^sub>t
    {|σ.´this→´XCoord = x ∧ ´this→´YCoord = y ∧
        ´dx = dx ∧ ´dy = dy ∧ ´this ≠ Null |}
    PROC Point_move_int_int(´this, ´dx, ´dy)
    {|´this = \<^bsup>σ\<^esup>this ∧
      ´this→´XCoord = x + dx ∧ ´this→´YCoord = y + dy|}"
 by(vcg, auto)

 procedures (imports globals_Point)
  Point_tester_Point(P::ref|result'::ref)
  "´P :== CALL Point_Point_int_int(10,11);;
    CALL Point_move_int_int(´P,1,0);;
   ´result' :== ´P"
 lemma (in Point_tester_Point_impl) Point_tester_Point_spec:
  "∀x y. Γ⊢\<^sub>t
    {| sz ≤ ´free ∧ ´P = Null |}
    ´result' :== PROC Point_tester_Point(´P)
    {| ´result' ≠ Null ∧
      ´result'→´XCoord = 11 ∧ ´result'→´YCoord = 11 |}"
 by(vcg, auto)
end
```

Memory in Simpl

- References and Heap
- Two components:
  - A list of allocated references
  - A natural number indicating the amount of available memory
- Expressed as a hoarestate

```
hoarestate  globals_memory =
   alloc = “ref list”
   free = nat
```

Fields

- Defined as maps from `ref => τ`

```
hoarestate globals_Point=globals_memory +
   XCoord :: “ref => int”
   YCoord :: “ref => int”
```

- Accessing a field:

```
   `P->`XCoord
```

Case Study—Benchmarks[1]

- Adding and Multiplying numbers
- Binary Search in an Array
- Sorting a Queue
- Layered Implementation of a Map ADT
- Linked-List Implementation of a Queue ADT
- Iterators
- Input/Output Streams
- An Integrated Application

[1] B. Weide et al., "Incremental Benchmarks for Software Verification Tools and Techniques," Verified Software: Theories, Tools, Experiments, 2008, pp. 84-98