

Jessie Plugin and ACSL Specifications

Virgile Prevosto

CEA List

March 31, 2009

long n
for 0 < i < n
C1) if (m
tmp2 =
of the

tmp2[i] = 0; if (i < (n-1)) else if (tmp1[i]) >= 1) << (n-1) - i; else tmp2[i] = tmp1[i]; /* Then the second part takes the first part...
tmp1[i] = 0; k = 5; k <= i) tmp1[i] += mc2[i][k] * tmp2[k]; /* The [i][k] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
i = 1; tmp1[i] >= 1; /* Final rounding: tmp2[i] is now represented on 9 bits. *if (tmp1[i] < -256) m2[i] = -256; else if (tmp1[i] > 255) m2[i] = 255; else m2[i] = tmp1[i];



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

long n;
for (i = 0; i < n; i++)
 C[i] = 0;
tmp2 = 0;
// ...

tmp2[j] = 0; k = 0; while (tmp1[j] >= 0) { tmp2[j] = tmp1[j]; k++; } Then the second part takes the first part as input.
tmp1[k] = 0; k = k + 1; tmp1[k] = m2[0][k] * tmp2[k]; } The [i][j] coefficient of the matrix product MC2 * TMP2, that is, *MC2*(TMP1) = MC2*(MC1 * M1) = MC2 * M1 * MC1.
i = 1; tmp1[0] >= 1; } Final rounding: tmp2[0] is now represented on 9 bits: *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

long n;
for (i = 0; i < n; i++)
 C[i] = 0;
tmp2 = ...
... of the

tmp2[i] = 0; i < (N-1) else if (tmp1[i]) >= 0; i < (N-1) tmp2[i] = (1 << (N-1)) - 1; else tmp2[i] = tmp1[i]; /* Then the second pass looks like the first one: */
tmp1[0] = 0; k = 0; k < n; k++) tmp1[k] += mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*[TMP2] = MC2*MC1*M1 = MC2*M1*MC1
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tm...



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)

long ra
for 0 <=
C1) if (m
tmp2 =
of the

tmp2[j] = 0; for (k = 0; k < n; k++) tmp2[j] += m2[j][k] * tmp1[k]; /* The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*[TMP1] = MC2*[M1] * MC1
l = 1; tmp1[l] += 1; */ Final rounding: tmp2[l] is now represented on 9 bits. *if (tmp1[l] < -256) m2[l] = -256; else if (tmp1[l] > 255) m2[l] = 255; else m2[l] = tm



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)

long ra
for 0 =>
C1) if m
tmp2 =
of the

tmp2[j] = 0; for (k = 0; k < n; k++) tmp1[j] += mc2[j][k] * tmp2[k]; /* The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*[TMP1] = MC2*[M1]*M1 = MC2*[M1]*M1 - 1. tmp1[j] >= 1. */ Final rounding: tmp2[j] is now represented on 9 bits. *if (tmp1[j] < -256) m2[j] = -256; else if (tmp1[j] > 255) m2[j] = 255; else m2[j] = tm



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)

(long ra
 for 0 <= i < n
 C1) if (m
 tmp2 =
 re of the

tmp2[i] = 0; for (k = 0; k < n; k++) tmp2[i] += m2[i][k] * tmp1[k]; /* The [i,j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(M1)*M1 = MC2*(M1)*M1 = 1 * tmp1[i][j] >= 1.*/ Final rounding: tmp2[i] is now represented on 9 bits. *if (tmp1[i][j] < -256) m2[i][j] = -256; else if (tmp1[i][j] > 255) m2[i][j] = 255; else m2[i][j] = tmp1[i][j];



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



(long ra
 t for 0 =>
 C1) if m
 tmp2 =
 re of the
 tmp2[0] = 1 << (nbl - 1) else if (tmp1[0]) >> 1 << (nbl - 1) tmp2[0] = (1 << (nbl - 1)) + tmp2[0] + tmp1[0]; /* Then the second part takes the first part of m
 tmp1[0] = 0; k = 0; k = k + 1; tmp1[0] = mc2[0][k] * tmp2[0][k]; /* The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*[TMP1] = MC2*[M1] = MC2*[M1] * MC1
 l = 1; tmp1[0] >> 1; /* Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];

- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



(long n)

 (for i <- 0

 (let (m)

 (tmp2

 (of the

 (tmp2[i]) = 1 <-> (n <-> i) ∧ (tmp1[i]) >= 1 <-> (n <-> i) ∧ (tmp2[i]) = 1 <-> (n <-> i) ∧ (tmp1[i]) >= 1)

 (tmp1[i]) = 0 <-> k <-> i ∧ (tmp1[i]) <= m2[i][k] * tmp2[i][j])

 (The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 * MC1

 (i = 1, tmp1[i]) >= 1.

 (Final rounding: tmp2[i]) is now represented on 9 bits. *if (tmp1[i]) < -255m2[i]) = -255 else if (tmp1[i]) > 255m2[i]) = 255 else tmp2[i] = tmp1[i]

- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



long n
for i <-
C1; if m
tmp2 =
of the

tmp2[i] := 1 + (n1 - i) * a + tmp1[i]; // i <- n1 - i; // a := tmp2[i] - tmp1[i]; // Then the second part takes for the first part
tmp1[i+k] := 0; k := 5; k := tmp1[i+k] := m2[i+k] * tmp2[i+k]; // The [i] coefficient of the matrix product MC2*TMP2, that is, *MC2*[i][TMP1] = MC2*[i][M1] * MC1
i := 1; tmp1[0] := 1; // Final rounding: tmp2[0] is now represented on 9 bits. if (tmp1[0] < -255) m2[0] := -255; else if (tmp1[0] > 255) m2[0] := 255; else m2[0] := tmp1[0];

- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



(long n) for 0 <= k < n { tmp2 = m[k]; for 0 <= j < n { tmp1 = m[j]; if (tmp1 < tmp2) { tmp2 = tmp1; } } } Then the second pass looks for the first pair (k, j) such that m[k] < m[j]. The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 * MC1 = 1 * tmp1[0] >= 1. *Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] <= 255) m[0] = 255; else if (tmp1[0] > 255) m[0] = 255; else m[0] = tmp1[0];

- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care

```

(long n)
for (i = 0; i < n; i++)
  C[i] = 0;
tmp2 = ...
// ...

```

```

tmp2[0] = 0; // (n-1) else if (tmp1[0]) >= 0; // (n-1) tmp2[0] = (1 << (n-1) - 1) else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one:
tmp1[0][k] = 0; k = 0; k++) tmp1[0][k] += mc2[0][k] * tmp2[0][k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
i = 1; tmp1[0][i] >>= 1; /* Final rounding: tmp2[0][i] is now represented on 9 bits. *if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tmp1[0][i];

```



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care

```

(long n)
for (i = 0; i < n; i++)
  C[i] = 0;
tmp2 = ...
... of the

```

```

tmp2[0] = 0; for (k = 0; k < n; k++) tmp2[k] = m2[0][k] * tmp2[k]; /* The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
l = 1; tmp1[0][l] >>= 1; */ Final rounding: tmp2[0][l] is now represented on 9 bits. *if (tmp1[0][l] < -256) m2[0][l] = -256; else if (tmp1[0][l] > 255) m2[0][l] = 255; else m2[0][l] =

```



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care

```

(long n)
for (i = 0; i < n; i++)
  C[i] = 0;
tmp2 = ...
... of the

```

```

tmp2[0] = 1; // (n-1) else if (tmp1[0]) >= 1 // (n-1) tmp2[0] = 1 // (n-1) // 1 else tmp2[0] = tmp1[0]; // Then the second part takes the first one
tmp1[0] = 0; k = 8; k-- tmp1[0] += mc2[0][k] * tmp2[k]; // The [j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
l = 1; tmp1[0] >= 1; // Final rounding: tmp2[0] is now represented on 9 bits. if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];

```



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



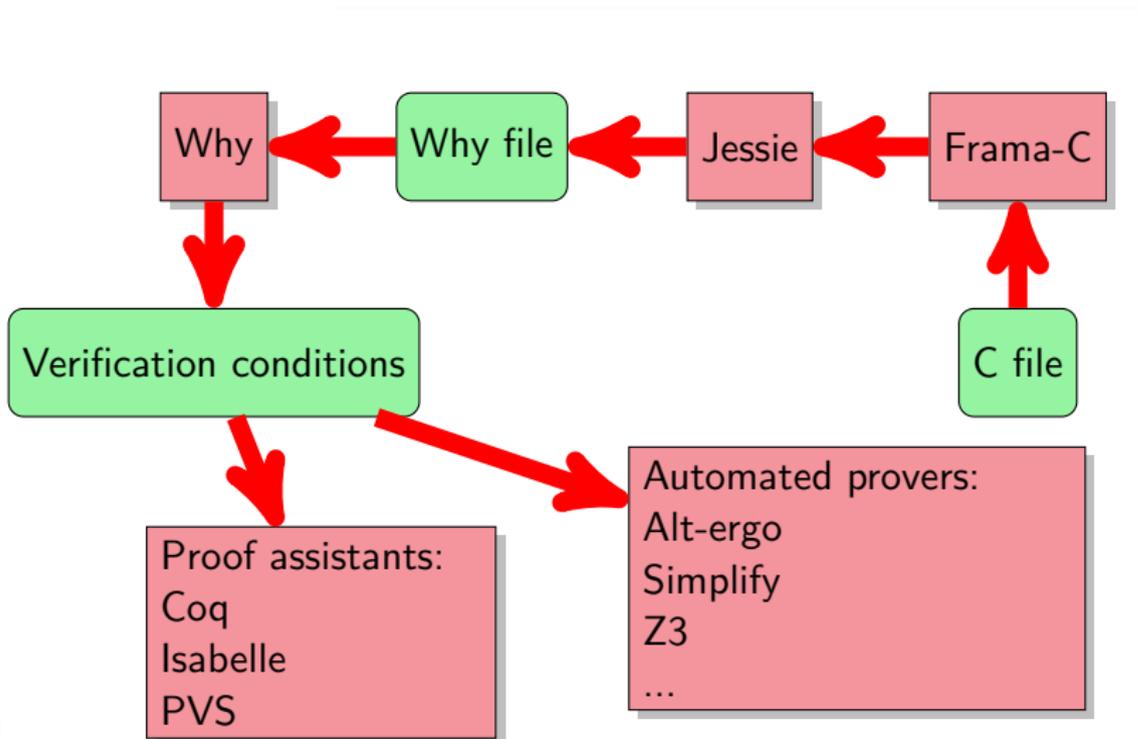
Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care





long no
for 0 =>
C1) if m
tmp2
of the

tmp2[0] = 1 -> (nbl - 1) else if tmp1[0] == 1 -> (nbl - 1) else tmp2[0] = 1
Then the second part looks like the first one
tmp1[0] = 0 k = 5 k-1 tmp1[0] = mc2[0]k * tmp2[0]k / The [k] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
= 1 tmp1[0] <= 1. Final spending tmp2[0] is now represented on 9 bits. if tmp1[0] < 256m2[0] = 256 else if tmp1[0] > 255m2[0] = 255 else tmp1[0] = 255

Check safety of a function

- ▶ Pointer accesses
- ▶ Arithmetic overflow
- ▶ Division

```
unsigned int M;
```

```
void mean(unsigned int* p, unsigned int* q) {
    M = (*p + *q) / 2;
}
```



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

long n;
for (i = 0; i < n; i++)
C[i] = 0;
tmp2 = ...
... of the

tmp2[i] = 0; i < (N-1) else if (tmp1[i]) >= 0; i < (N-1) tmp2[i] = (1 << (N-1)) - 1; else tmp2[i] = tmp1[i]; /* Then the second pass looks like the first one: */
tmp1[0] = 0; k = 0; k < n; k++) tmp1[k] += mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*(M1)*MC1
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tm...



- ▶ Functional specification
- ▶ Pre-conditions (requires)
- ▶ Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



- ▶ Functional specification
- ▶ Pre-conditions (requires)
- ▶ Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



- ▶ Functional specification
- ▶ Pre-conditions (requires)
- ▶ Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



The specification:

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
  assigns M;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q);
```



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

long n;
for (i = 0; i < n; i++)
C[i] = 0;
tmp2 = ...
... of the

tmp2[i] = 0; i < (n-1); else if (tmp1[i] >= 0) i < (n-1); tmp2[i] = (i < (n-1) ? 1 : 0); else tmp2[i] = tmp1[i]; /* Then the second pass looks like the first one: */
tmp1[0] = 0; k = 0; k < n; k++) tmp1[k] += mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
i = 1; tmp1[0] >= 1; */ Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tm...



- ▶ Introduced by Floyd and Hoare (70s)
- ▶ Hoare triple: $\{P\}s\{Q\}$, meaning: *If P holds, then Q will hold after the execution of statement s*
- ▶ Deduction rules on Hoare triples: *Axiomatic semantic*

long n
for i in
c1: if m
tmp2
of the

tmp2[i] = 1 << (n-1) else if tmp1[i] >= 1 << (n-1) tmp2[i] = 1 << (n-1) + tmp1[i] else tmp2[i] = tmp1[i] ? Then the second part takes for the first part
tmp1[i] = 0 k = 5 k => tmp1[i] = mc2[i][k] * tmp2[k] // The [i][j] coefficient of the matrix product MC2*TMP2, that is *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
i = 1 tmp1[i] >= 1 // Final rounding tmp2[i] is now represented on 9 bits *if (tmp1[i] < -255) tmp2[i] = -255 else if (tmp1[i] > 255) tmp2[i] = 255 else tmp2[i] = tmp1[i]



$$\frac{\{P\}\{\}\{P\}}{\{P\} \Rightarrow P'} \quad \frac{\{P'\}s\{Q'\} \quad Q' \Rightarrow Q}{\{P\}s\{Q\}}$$

$$\frac{\{P\}s_1\{R\} \quad \{R\}s_2\{Q\}}{\{P\}s_1;s_2\{Q\}} \quad \frac{e \text{ evaluates without error}}{\{P[x \leftarrow e]\}x=e;\{P\}}$$

$$\frac{\{P \wedge e\}s_1\{Q\} \quad \{P \wedge !e\}s_2\{Q\}}{\{P\}\mathbf{if} (e) \{ s_1 \} \mathbf{else} \{ s_2 \}\{Q\}}$$

$$\frac{\{I \wedge e\}s\{I\}}{\{I\}\mathbf{while} (e) \{ s \}\{I \wedge !e\}}$$



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.

long ra
for 0 ->
C1) if (m
tmp2 =
of the

tmp2[0] = 1 && (n&l - 1) else if (tmp1[0]) >= 1 && (n&l - 1) tmp2[0] = 1 && (n&l - 1) else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one: "for
tmp1[0] = 0; k = 5; k <= 5; k++) tmp1[0][k] += m2[0][k] * tmp2[0][k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp1[0][i] >>= 1; /* Final rounding: tmp2[0][i] is now represented on 9 bits. *if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tm



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.

long ra
for 0 <=
C1) if (m
tmp2 =
se of the

tmp2[0] = 1 << (n0 - 1) else if (tmp1[0]) >= 1 << (n0 - 1) tmp2[0] = 1 << (n0 - 1) + else tmp2[0] = tmp1[0]; /* Then the second part looks like the first one: "for (k = 1; k <= 5; k++) tmp1[0][k] += mc2[0][k] * tmp2[0][k];" The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1. i = 1; tmp1[0][i] >= 1; */ Final rounding: tmp2[0][i] is now represented on 9 bits. *if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tmp1[0][i];



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.

if (long n) for (i = 0; i < n; i++) tmp2 = ...

tmp2[i] = i * (n-1) else if (tmp1[i] >= 1) * (n-1) tmp2[i] = (i < (n-1) ? tmp2[i] + tmp1[i] : * (n-1) * tmp1[i]); /* Then the second part takes like the first one: tmp1[i] = 0; k = 5; k++ tmp1[i] += mc2[i][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1. i = 3; tmp1[0] >= 3; /* Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.



- ▶ Assignment

$$WP(x=e, Q) = Q[x \leftarrow e]$$

- ▶ Sequence

$$WP(s_1; s_2, Q) = WP(s_1, WP(s_2; Q))$$

- ▶ Conditional

$$WP(\text{if } (e) \{ s_1 \} \text{ else } \{ s_2 \}, Q) = e \Rightarrow WP(s_1, Q) \wedge !e \Rightarrow WP(s_2, Q)$$

- ▶ While

$$WP(\text{while } (e) \{ s \}, Q) = I \wedge \forall \omega. I \Rightarrow (e \Rightarrow WP(s, I) \wedge !e \Rightarrow Q)$$



Issue

The same memory location can be accessed through different means:

```
int y;
int* yptr = &y;
*yptr = 3;
/*@ assert y == 3; */
```

- ▶ Again, supposing that any two pointers can be aliases would lead to intractable proof obligations.
- ▶ Memory is separated in disjoint regions
- ▶ Some hypotheses are done (as additional pre-conditions)



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

long n;
for (i = 0; i < n; i++)
c[i] = 0;
tmp2 = ...
... of the

tmp2[i] = 0; // (i < (n-1) || also if (tmp1[i] >= 0) // (i < (n-1) || tmp2[i] = (i < (n-1) ? 0 : abs(tmp2[i]) + tmp1[i]); /* Then the second part takes the first one. */
tmp1[0][k] = 0; k = k+1; tmp1[0][k] += mc2[0][k] * tmp2[0][k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
i = i+1; tmp1[0][k] >= 0; /* Final rounding: tmp2[0][i] is now represented on 9 bits. *if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tmp1[0][i];



Informal spec

- ▶ Input: a **sorted** array and its length, an element to search.
- ▶ Output: index of the element or -1 if not found

Implementation

```
int find_array(int* arr, int length, int query) {
    int low = 0;
    int high = length - 1;
    while (low ≤ high) {
        int mean = low + (high - low) / 2;
        if (arr[mean] ≡ query) return mean;
        if (arr[mean] < query) low = mean + 1;
        else high = mean - 1;
    }
    return -1;
}
```



Informal specification

- ▶ Input: an array and its length
- ▶ Output: the array is sorted in ascending order

```
int index_min(int* a, int low, int high);
```

```
void swap(int* arr, int i, int j);
```

```
void min_sort(int* arr, int length) {
    for(int i = 0; i < length; i++) {
        int min = index_min(arr,i,length);
        swap(arr,i,min);
    }
}
```

