ARITH 18

Montpellier, France
June 25-27, 2007

18th IEEE Symposium on Computer Arithmetic

Formal Verification of Floating-Point programs

Sylvie Boldo and Jean-Christophe Filliâtre

Montpellier – June, 26th 2007

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

INRIA Futurs

CNRS, LRI

Motivations

Goal: reliability in numerical software

Motivations

Goal: reliability in numerical software

Tool: formal proofs

Motivations

Goal: reliability in numerical software

Tool: formal proofs

Drawback: we were not checking the real program

Motivations

Goal: reliability in numerical software

Tool: formal proofs

Drawback: we were not checking the real program

⇒ put together existing tools
⇒ check what is really written by programmers

Outline

Existing tools
- Caduceus
- Formalization of floats

Model and specification of FP numbers

Examples

Conclusion

What is Caduceus?

The method is to annotate the C program

What is Caducеus?

The method is to annotate the C program

We add pre-conditions and post-conditions to functions

We add variants, invariants, assertions

What is Caduceus?

The method is to annotate the C program

We add pre-conditions and post-conditions to functions

We add variants, invariants, assertions

The tool generates proof obligations (such as Coq theorems) associated to the user annotations

The proof of the verification conditions ensures that the program meets its specification

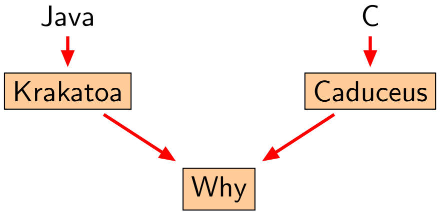
Caduceus

Java

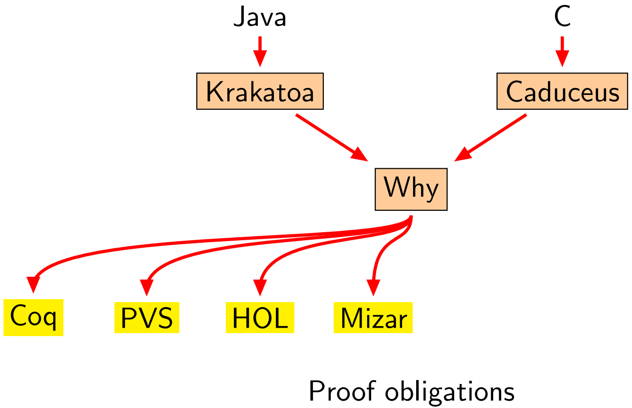
C

Caduceus

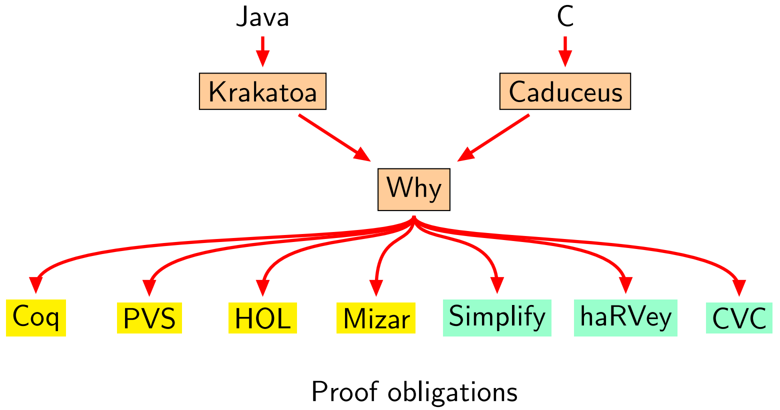
Caduceus



Caduceus


Java
C
Krakatoa
Caduceus
Why
Coq
PVS
HOL
Mizar
Proof obligations

Caduceus


Java
C
Krakatoa
Caduceus
Why
Coq
PVS
HOL
Mizar
Simplify
haRVey
CVC
Proof obligations

Example: search in an array

```
int index(int t[], int n, int v) {
  int i = 0;
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

Example: search in an array

```
/*@ requires \valid_range(t,0,n-1)
  @ ensures
  @   (0 <= \result < n => t[\result] == v) &&
  @   (\result == n =>
  @        \forall int i; 0 <= i < n => t[i] != v) */

int index(int t[], int n, int v) {
  int i = 0;
  /*@ invariant 0 <= i &&
    @      \forall int k; 0 <= k < i => t[k] != v
    @ variant n - i */
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

Coq formalization (by Daumas, Rideau, Théry)

Float = pair of signed integers (mantissa, exponent)

$$(n, e) \in \mathbb{Z}^2$$

Coq formalization (by Daumas, Rideau, Théry)

Float = pair of signed integers (mantissa, exponent)
associated to a real value

$$(n, e) \in \mathbb{Z}^2 \hookrightarrow n \times \beta^e \in \mathbb{R}$$

Coq formalization (by Daumas, Rideau, Théry)

Float = pair of signed integers (mantissa, exponent)
associated to a real value

$$(n, e) \in \mathbb{Z}^2 \hookrightarrow n \times \beta^e \in \mathbb{R}$$

1.00010_2 E 4	$\mapsto$	$(100010_2, -1)_2$	$\hookrightarrow$	17
IEEE-754		significant of 754R		real value

$\Rightarrow$ normal floats, subnormal floats, cohorts, ~~overflow~~

Partial Conclusion

- We have all the needed tools
  - program → formal theorem (obligations)
  - formal float, formal rounding...

Partial Conclusion

- We have all the needed tools
  - program $\longrightarrow$ formal theorem (obligations)
  - formal float, formal rounding...
- We have to merge them to get a tool:
  program $\longrightarrow$ formal theorem on FP arithmetic

Partial Conclusion

- We have all the needed tools
  - program $\longrightarrow$ formal theorem (obligations)
  - formal float, formal rounding...
- We have to merge them to get a tool:
  program $\longrightarrow$ formal theorem on FP arithmetic
- We have to decide how to specify a FP program!

Caduceus's model of FP numbers

A "program" float is a triple:

- the floating-point number, as computed by the program,
  $x \longrightarrow x_f$ floating-point part

Caduceus's model of FP numbers

A "program" float is a triple:

- the floating-point number, as computed by the program,
  $x \longrightarrow x_f$ floating-point part
- the value if all previous computations were exact,
  $x \longrightarrow x_e$ exact part

Caduceus's model of FP numbers

A "program" float is a triple:

- the floating-point number, as computed by the program,
  $x \longrightarrow x_f$ floating-point part
- the value if all previous computations were exact,
  $x \longrightarrow x_e$ exact part
- the ideally computed value
  $x \longrightarrow x_m$ model part

Caduceus's model of FP numbers (II)

Program features

- types for single and double precision floats
- roundings that may be switched
- basic operations
- . . .

Caduceus's model of FP numbers (II)

Program features

- types for single and double precision floats
- roundings that may be switched
- basic operations
- ...

Specification features

- computations are exact inside annotations
- access to the exact and model parts
- `round_error` and `total_error` macros

Example 1: exact subtraction

```
float Sterbenz(float x, float y){
  return x-y;
}
```

Example 1: exact subtraction

```
/*@ requires y/2 <= x <= 2*y
  @ ensures  \result == x-y
  @*/

float Sterbenz(float x,float y){
  return x-y;
}
```

(44 lines of Coq)

Example 2: Malcolm's Algorithm

```
double malcolm() {
  double A, B;
  A=2;
  while (A != (A+1))
      A*=2;

  B=1;
  while ((A+B)-A != B)
    B++;
  return B; }
```

(747 lines of Coq)

Example 2: Malcolm's Algorithm

```
/*@ ensures \result == 2 */
double malcolm() {
  double A, B;
  A=2;  /*@ assert A==2 */
  /*@ invariant A == 2 ^^ my_log(A)
    @     && 1 <= my_log(A) <= 53
    @ variant (53-my_log(A)) */
  while (A != (A+1))
      A*=2;
  /*@ assert A == 2 ^^ (53) */

  B=1;   /*@ assert B==1 */
  /*@ invariant B == IRNDD(B) && 1 <= B <= 2
    @ variant (2-IRNDD(B)) */
  while ((A+B)-A != B)
    B++;
  return B; }
```

Example 3: stupid exponential computation

```
double my_exp(double x) {
  double y=1+x*(1+x/2);
  return y;
}
```

Example 3: stupid exponential computation

```
/*@ requires |x| <= 2 ^^ (-3)
  @ ensures \model(\result)==exp(\model(x))
  @  && (\round_error(x)==0
  @        => \round_error(\result)
  @              <= 2 ^^ (-52))
  @  && \total_error(\result)
  @        <= \total_error(x)
  @              + 2 ^^ (-51)
*/
double my_exp(double x) {
  double y=1+x*(1+x/2);
  /*@ \set_model y exp(\model(x)) */
  return y;
}
```

(unproved)

Existing tools

- Caduceus
- Formalization of floats

Model and specification of FP numbers

Examples

Conclusion

Conclusion

Advantages

- a way to specify and formally prove a FP program
- includes all other aspects of program verification
- with or without Overflow
- intuitive specification

Conclusion

Advantages

- ⊕ a way to specify and formally prove a FP program
- ⊕ includes all other aspects of program verification
- ⊕ with or without Overflow
- ⊕ intuitive specification

Drawbacks

- ⊖ no NaNs, no $\pm\infty$
- ⊖ no exception, no flag
- ⊖ no way to detect compiler optimizations
- ⊖ fails on Intel architectures (no way to predict if 53 or 80 bits are used)