Programming with Uncertainties

Eva Darulová

Writing accurate numerical software is hard because of many sources of unavoidable **uncertainties**

- measurement uncertainties
- finite precision arithmetic
- limited resources

What if you could write your program in reals?

- programmer can reason in real arithmetic
- verification algorithm can reason in reals
- this approach enables the developers and tools to quantify the deviation of implementation outputs to ideal ones
- compiler can perform sound optimizations (example: associativity)
- this approach supports external uncertainties, not only roundoff errors

```
def sineTaylor(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def sineOrder3(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  0.954929658551372 * x -  0.1290061773279798*(x*x*x)
 } ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def checkApproximation(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
 } ensuring(res => ~res <= 0.1)
```

ranges for inputs

```
def sineTaylor(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def sineOrder3(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  0.954929658551372 * x -  0.1290061773279798*(x*x*x)
 } ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def checkApproximation(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
 } ensuring(res => ~res <= 0.1)
```

ranges for inputs

```
def sineTaylor(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

specification of uncertainties

```
def sineOrder3(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  0.954929658551372 * x -  0.1290061773279798*(x*x*x)
 } ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def checkApproximation(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
 } ensuring(res => ~res <= 0.1)
```

ranges for inputs

```
def sineTaylor(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def sineOrder3(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  0.954929658551372 * x -  0.1290061773279798*(x*x*x)
 } ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

specification of uncertainties

```
def checkApproximation(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
 } ensuring(res => ~res <= 0.1)
```

value actually computed

ranges for inputs

```
def sineTaylor(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def sineOrder3(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  0.954929658551372 * x -  0.1290061773279798*(x*x*x)
 } ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

specification of uncertainties

```
def checkApproximation(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
 } ensuring(res => ~res <= 0.1)
```

compiler selects data type automatically
(floating-point or fixed-point)

value actually computed

ranges for inputs

```
def sineTaylor(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

```
def sineOrder3(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  0.954929658551372 * x -  0.1290061773279798*(x*x*x)
 } ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

specification of uncertainties

```
def checkApproximation(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
 } ensuring(res => ~res <= 0.1)
```

compiler selects data type automatically
(floating-point or fixed-point)

value actually computed

key enabling technology is precise reasoning about ranges of variables in the presence of non-linear arithmetic