Lecture 2

Core Library and Precision-Driven Computation

Chee Yap
Courant Institute of Mathematical Sciences
New York University

Overview

We introduce the Core Library and the underlying mechanism for achieving its basic properties. Two key concepts are Precision-Driven Computation and Conditional Zero Bounds.

- I. Core Library
- II. Precision-Driven Computation
- III. Conditional Zero Bounds

I. CORE LIBRARY

Modes of Numerical Computing

- Landscape of Numerical Modes
  - Why there is not ONE number type, $\mathbb{C}$?
  - Diversity of number types and applications
  - $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R} \subseteq \mathbb{C}$
- 1. Symbolic Mode (e.g., Maple)
  - $\sqrt{2}$ is represented exactly, symbolically
- 2. FP Mode (e.g., IEEE Arithmetic)
  - Fixed Precision, Floating Point
- 3. Arbitrary Precision Mode
  - Brent's MP, Bailey's MPFUN, Muller's iRRAM

Modes of Numerical Computing

- Landscape of Numerical Modes
  - Why there is not ONE number type, $\mathbb{C}$?
  - Diversity of number types and applications
  - $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R} \subseteq \mathbb{C}$
- 1. Symbolic Mode (e.g., Maple)
  - $\sqrt{2}$ is represented exactly, symbolically
- 2. FP Mode (e.g., IEEE Arithmetic)
  - Fixed Precision, Floating Point
- 3. Arbitrary Precision Mode
  - Brent's MP, Bailey's MPFUN, Muller's iRRAM

Modes of Numerical Computing

- Landscape of Numerical Modes
  - ∗ Why there is not ONE number type, $\mathbb{C}$?
  - ∗ Diversity of number types and applications
  - ∗ $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R} \subseteq \mathbb{C}$
- 1. Symbolic Mode (e.g., Maple)
  - ∗ $\sqrt{2}$ is represented exactly, symbolically
- 2. FP Mode (e.g., IEEE Arithmetic)
  - ∗ Fixed Precision, Floating Point
- 3. Arbitrary Precision Mode
  - ∗ Brent's MP, Bailey's MPFUN, Muller's iRRAM

Modes of Numerical Computing

- Landscape of Numerical Modes
  - Why there is not ONE number type, $\mathbb{C}$?
  - Diversity of number types and applications
  - $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R} \subseteq \mathbb{C}$
- 1. Symbolic Mode (e.g., Maple)
  - $\sqrt{2}$ is represented exactly, symbolically
- 2. FP Mode (e.g., IEEE Arithmetic)
  - Fixed Precision, Floating Point
- 3. Arbitrary Precision Mode
  - Brent's MP, Bailey's MPFUN, Muller's iRRAM

Modes of Numerical Computing

- Landscape of Numerical Modes
  - Why there is not ONE number type, $\mathbb{C}$?
  - Diversity of number types and applications
  - $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R} \subseteq \mathbb{C}$
- 1. Symbolic Mode (e.g., Maple)
  - $\sqrt{2}$ is represented exactly, symbolically
- 2. FP Mode (e.g., IEEE Arithmetic)
  - Fixed Precision, Floating Point
- 3. Arbitrary Precision Mode
  - Brent's MP, Bailey's MPFUN, Muller's iRRAM

- 4. Interval Arithmetic or Enclosure Mode
  - Certified or validated computing
  - Automatic error tracking
- 5. Guaranteed Accuracy Mode
  - E.g., LEDA Real, Core Library
  - A priori precision bounds is given as input

- 4. Interval Arithmetic or Enclosure Mode
  - Certified or validated computing
  - Automatic error tracking
- 5. Guaranteed Accuracy Mode
  - E.g., LEDA Real, Core Library
  - A priori precision bounds is given as input

- 4. Interval Arithmetic or Enclosure Mode
  - Certified or validated computing
  - Automatic error tracking

- 5. Guaranteed Accuracy Mode
  - E.g., LEDA Real, Core Library
  - A priori precision bounds is given as input

Core Numerical Accuracy API

- Framework to unify some of the above modes
- CORE Levels:
  - Level I: IEEE Arithmetic
  - Level II: Arbitrary Accuracy
  - Level III: Guaranteed Accuracy
  - Level IV: Mixed Accuracy

Core Numerical Accuracy API

- Framework to unify some of the above modes
- CORE Levels:
  - ∗ Level I: IEEE Arithmetic
  - ∗ Level II: Arbitrary Accuracy
  - ∗ Level III: Guaranteed Accuracy
  - ∗ Level IV: Mixed Accuracy

Core Numerical Accuracy API

- Framework to unify some of the above modes
- CORE Levels:
  * Level I: IEEE Arithmetic
  * Level II: Arbitrary Accuracy
  * Level III: Guaranteed Accuracy
  * Level IV: Mixed Accuracy

Core Numerical Accuracy API

- Framework to unify some of the above modes
- CORE Levels:
  - Level I: IEEE Arithmetic
  - Level II: Arbitrary Accuracy
  - Level III: Guaranteed Accuracy
  - Level IV: Mixed Accuracy

Core Numerical Accuracy API

- Framework to unify some of the above modes
- CORE Levels:
  * Level I: IEEE Arithmetic
  * Level II: Arbitrary Accuracy
  * Level III: Guaranteed Accuracy
  * Level IV: Mixed Accuracy

Core Numerical Accuracy API

- Framework to unify some of the above modes
- CORE Levels:
  * Level I: IEEE Arithmetic
  * Level II: Arbitrary Accuracy
  * Level III: Guaranteed Accuracy
  * Level IV: Mixed Accuracy

Core Numerical Accuracy API

- Framework to unify some of the above modes
- CORE Levels:
  - Level I: IEEE Arithmetic
  - Level II: Arbitrary Accuracy
  - Level III: Guaranteed Accuracy
  - Level IV: Mixed Accuracy

- Delivery Mechanism (C++):


```
#define Core_Level 3
#include "CORE.h"

...  standard C++ Program here ...
```

- Default Level is 3

- Delivery Mechanism (C++):

```
#define Core_Level 3
#include "CORE.h"

...  standard C++ Program here ...
```

- Default Level is 3

Core Library for the Impatient

- Structure of CORE files
  - ∗ src, inc, lib, ext, progs
  - ∗ Makefile in every directory
- Go to $(COREPATH)/progs/
  - ∗ Create your own subdir myproj.
- Copy into myproj one of the Makefiles
  - ∗ Take from a sibling directory. E.g., progs/demos
- Write your first program, helloCore.cpp.
- Modify the Makefile: e.g., simply set "p = helloCore".
- Now, type "make".

Core Library for the Impatient

- Structure of CORE files
  - src, inc, lib, ext, progs
  - Makefile in every directory
- Go to $(COREPATH)/progs/
  - Create your own subdir myproj.
- Copy into myproj one of the Makefiles
  - Take from a sibling directory. E.g., progs/demos
- Write your first program, helloCore.cpp.
- Modify the Makefile: e.g., simply set "p = helloCore".
- Now, type "make".

Core Library for the Impatient

- Structure of CORE files
  - src, inc, lib, ext, progs
  - Makefile in every directory
- Go to $(COREPATH)/progs/
  - Create your own subdir myproj.
- Copy into myproj one of the Makefiles
  - Take from a sibling directory. E.g., progs/demos
- Write your first program, helloCore.cpp.
- Modify the Makefile: e.g., simply set "p = helloCore".
- Now, type "make".

Core Library for the Impatient

- Structure of CORE files
  - ∗ src, inc, lib, ext, progs
  - ∗ Makefile in every directory
- Go to $(COREPATH)/progs/
  - ∗ Create your own subdir myproj.
- Copy into myproj one of the Makefiles
  - ∗ Take from a sibling directory. E.g., progs/demos
- Write your first program, helloCore.cpp.
- Modify the Makefile: e.g., simply set "p = helloCore".
- Now, type "make".

Core Library for the Impatient

- Structure of CORE files
  - src, inc, lib, ext, progs
  - Makefile in every directory
- Go to $(COREPATH)/progs/
  - Create your own subdir myproj.
- Copy into myproj one of the Makefiles
  - Take from a sibling directory. E.g., progs/demos
- Write your first program, helloCore.cpp.
- Modify the Makefile: e.g., simply set "p = helloCore".
- Now, type "make".

Core Library for the Impatient

- Structure of CORE files
  - src, inc, lib, ext, progs
  - Makefile in every directory
- Go to $(COREPATH)/progs/
  - Create your own subdir myproj.
- Copy into myproj one of the Makefiles
  - Take from a sibling directory. E.g., progs/demos
- Write your first program, helloCore.cpp.
- Modify the Makefile: e.g., simply set "p = helloCore".
- Now, type "make".

Core Library for the Impatient

- Structure of CORE files
  * src, inc, lib, ext, progs
  * Makefile in every directory
- Go to $(COREPATH)/progs/
  * Create your own subdir myproj.
- Copy into myproj one of the Makefiles
  * Take from a sibling directory. E.g., progs/demos
- Write your first program, helloCore.cpp.
- Modify the Makefile: e.g., simply set "p = helloCore".
- Now, type "make".

Numerical I/O

- Assume: standard C++ program compiled in Level 3

- Key Principle: the internal rep is exact
  - ∗ Comparisons are exact
  - ∗ Input may be inexact
  - ∗ Printout can only be rational or bigfloat approximation

- Class of Extended Longs
  - ∗ Machine long, with special values
  - ∗ CORE_posInfty, CORE_negInfty, CORE_NaN
  - ∗ Main application: to specify precision

- Input will be exact if represented as strings
  - ∗ E.g., double x = 0.123; double y = "0.123"; double z

Numerical I/O

- Assume: standard C++ program compiled in Level 3
- Key Principle: the internal rep is exact
  - ∗ Comparisons are exact
  - ∗ Input may be inexact
  - ∗ Printout can only be rational or bigfloat approximation
- Class of Extended Longs
  - ∗ Machine long, with special values
  - ∗ CORE_posInfty, CORE_negInfty, CORE_NaN
  - ∗ Main application: to specify precision
- Input will be exact if represented as strings
  - ∗ E.g., double x = 0.123; double y = "0.123"; double z

Numerical I/O

- Assume: standard C++ program compiled in Level 3
- Key Principle: the internal rep is exact
  - ∗ Comparisons are exact
  - ∗ Input may be inexact
  - ∗ Printout can only be rational or bigfloat approximation
- Class of Extended Longs
  - ∗ Machine long, with special values
  - ∗ CORE_posInfty, CORE_negInfty, CORE_NaN
  - ∗ Main application: to specify precision
- Input will be exact if represented as strings
  - ∗ E.g., double x = 0.123; double y = "0.123"; double z

Numerical I/O

- Assume: standard C++ program compiled in Level 3

- Key Principle: the internal rep is exact
  - ∗ Comparisons are exact
  - ∗ Input may be inexact
  - ∗ Printout can only be rational or bigfloat approximation

- Class of Extended Longs
  - ∗ Machine long, with special values
  - ∗ CORE_posInfty, CORE_negInfty, CORE_NaN
  - ∗ Main application: to specify precision

- Input will be exact if represented as strings
  - ∗ E.g., double x = 0.123; double y = "0.123"; double z

= "123/100"; double w= "123e-3";

* Global Variable: defInputDigits

- Output: only see rational or bigfloat approximations
  * E.g., cout << x ;
- We never print garbage digits
  * The last digit is off by ± 1
  * So a printout of 1.99999 is OK for 2.0
  * To set output precision, e.g., cout << setprecision(15);
- Approximation: E.g., x.approx(rprec, aprec);
  * Global variable: defAbsPrec, defRelPrec
  * Composite Precision: [relprec, absprec]
- Facility for I/O of hugh numbers (in hexadecimal) in files
  * Can read any prefix of the file

= "123/100"; double w= "123e-3";

  * Global Variable: defInputDigits

- Output: only see rational or bigfloat approximations
  * E.g., cout << x ;

- We never print garbage digits
  * The last digit is off by ± 1
  * So a printout of 1.99999 is OK for 2.0
  * To set output precision, e.g., cout << setprecision(15);

- Approximation: E.g., x.approx(rprec, aprec);
  * Global variable: defAbsPrec, defRelPrec
  * Composite Precision: [relprec, absprec]

- Facility for I/O of hugh numbers (in hexadecimal) in files
  * Can read any prefix of the file

= "123/100"; double w= "123e-3";

  * Global Variable: defInputDigits

- Output: only see rational or bigfloat approximations
  * E.g., cout << x ;
- We never print garbage digits
  * The last digit is off by ± 1
  * So a printout of 1.99999 is OK for 2.0
  * To set output precision, e.g., cout << setprecision(15);
- Approximation: E.g., x.approx(rprec, aprec);
  * Global variable: defAbsPrec, defRelPrec
  * Composite Precision: [relprec, absprec]
- Facility for I/O of hugh numbers (in hexadecimal) in files
  * Can read any prefix of the file

= "123/100"; double w= "123e-3";

  * Global Variable: defInputDigits

- Output: only see rational or bigfloat approximations
  * E.g., cout << x ;
- We never print garbage digits
  * The last digit is off by ± 1
  * So a printout of 1.99999 is OK for 2.0
  * To set output precision, e.g., cout << setprecision(15);
- Approximation: E.g., x.approx(rprec, aprec);
  * Global variable: defAbsPrec, defRelPrec
  * Composite Precision: [relprec, absprec]
- Facility for I/O of hugh numbers (in hexadecimal) in files
  * Can read any prefix of the file

= "123/100"; double w= "123e-3";

- ∗ Global Variable: defInputDigits

- Output: only see rational or bigfloat approximations
  - ∗ E.g., cout << x ;

- We never print garbage digits
  - ∗ The last digit is off by ± 1
  - ∗ So a printout of 1.99999 is OK for 2.0
  - ∗ To set output precision, e.g., cout << setprecision(15);

- Approximation: E.g., x.approx(rprec, aprec);
  - ∗ Global variable: defAbsPrec, defRelPrec
  - ∗ Composite Precision: [relprec, absprec]

- Facility for I/O of hugh numbers (in hexadecimal) in files
  - ∗ Can read any prefix of the file

= "123/100"; double w= "123e-3";

- Global Variable: defInputDigits

- Output: only see rational or bigfloat approximations
  - E.g., cout << x ;

- We never print garbage digits
  - The last digit is off by ± 1
  - So a printout of 1.99999 is OK for 2.0
  - To set output precision, e.g., cout << setprecision(15);

- Approximation: E.g., x.approx(rprec, aprec);
  - Global variable: defAbsPrec, defRelPrec
  - Composite Precision: [relprec, absprec]

- Facility for I/O of hugh numbers (in hexadecimal) in files
  - Can read any prefix of the file

= "123/100"; double w= "123e-3";

  * Global Variable: defInputDigits

- Output: only see rational or bigfloat approximations
  * E.g., cout << x ;
- We never print garbage digits
  * The last digit is off by ± 1
  * So a printout of 1.99999 is OK for 2.0
  * To set output precision, e.g., cout << setprecision(15);
- Approximation: E.g., x.approx(rprec, aprec);
  * Global variable: defAbsPrec, defRelPrec
  * Composite Precision: [relprec, absprec]
- Facility for I/O of hugh numbers (in hexadecimal) in files
  * Can read any prefix of the file

- Question: my internal value is $\sqrt{2}$, but after setprecision(11), it still prints 1.414.
  - Why not 1.4142135624?
  - What is the solution?

- Question: my internal value is $\sqrt{2}$, but after setprecision(11), it still prints 1.414.
  - Why not 1.4142135624?
  - What is the solution?

How It Works in Core Library

- Level 1 Number Types
  - int, long, float, double
- Level 2 Number Types
  - BigInt, BigRational, BigFloat, Real
- Level 3 Number Types
  - Expr
- Promotion and Demotion
  - 1⇔3 : long, double ⇔Expr
  - 1⇔2 : long ⇔ BigInt; double ⇔ BigFloat, BigRat
  - Principle: any program must compile in each level

How It Works in Core Library

- Level 1 Number Types
  - int, long, float, double
- Level 2 Number Types
  - BigInt, BigRational, BigFloat, Real
- Level 3 Number Types
  - Expr
- Promotion and Demotion
  - 1⇔3 : long, double ⇔Expr
  - 1⇔2 : long ⇔ BigInt; double ⇔ BigFloat, BigRat
  - Principle: any program must compile in each level

How It Works in Core Library

- Level 1 Number Types
  - int, long, float, double
- Level 2 Number Types
  - BigInt, BigRational, BigFloat, Real
- Level 3 Number Types
  - Expr
- Promotion and Demotion
  - 1⇔3 : long, double ⇔Expr
  - 1⇔2 : long ⇔ BigInt; double ⇔ BigFloat, BigRat
  - Principle: any program must compile in each level

How It Works in Core Library

- Level 1 Number Types
  - int, long, float, double
- Level 2 Number Types
  - BigInt, BigRational, BigFloat, Real
- Level 3 Number Types
  - Expr
- Promotion and Demotion
  - 1⇔3 : long, double ⇔Expr
  - 1⇔2 : long ⇔ BigInt; double ⇔ BigFloat, BigRat
  - Principle: any program must compile in each level

How It Works in Core Library

- Level 1 Number Types
  - int, long, float, double
- Level 2 Number Types
  - BigInt, BigRational, BigFloat, Real
- Level 3 Number Types
  - Expr
- Promotion and Demotion
  - 1⇔3 : long, double ⇔Expr
  - 1⇔2 : long ⇔ BigInt; double ⇔ BigFloat, BigRat
  - Principle: any program must compile in each level

- What is Level 4?
  - ∗ Research Problem: Not fully defined
- Fundamental gap between Level 2 and Level 3
  - ∗ Role of zero bounds

- What is Level 4?
  - ∗ Research Problem: Not fully defined
- Fundamental gap between Level 2 and Level 3
  - ∗ Role of zero bounds

- What is Level 4?
  - ∗ Research Problem: Not fully defined
- Fundamental gap between Level 2 and Level 3
  - ∗ Role of zero bounds

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - User Specified precision (if any)
  - BigFloat approximation α
  - Error bound for α
  - Zero bound for α

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - User Specified precision (if any)
  - BigFloat approximation α
  - Error bound for α
  - Zero bound for α

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - User Specified precision (if any)
  - BigFloat approximation α
  - Error bound for α
  - Zero bound for α

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - User Specified precision (if any)
  - BigFloat approximation α
  - Error bound for α
  - Zero bound for α

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - User Specified precision (if any)
  - BigFloat approximation α
  - Error bound for α
  - Zero bound for α

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - ∗ E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - ∗ E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - ∗ User Specified precision (if any)
  - ∗ BigFloat approximation α
  - ∗ Error bound for α
  - ∗ Zero bound for α

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - User Specified precision (if any)
  - BigFloat approximation α
  - Error bound for α
  - Zero bound for α

Expressions in Core Library

- An expression is a DAG (directed acyclic graph
  - E.g. $E = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$
- Each operation constructs an expression
  - E.g., $x \leftarrow a + b$
- At each node of expression, store:
  - User Specified precision (if any)
  - BigFloat approximation α
  - Error bound for α
  - Zero bound for α

II. PRECISION-DRIVEN EVALUATION

Expression Evaluation Problem

- Ω be set of real operators (partial functions)
  - E.g., $\Omega = \{+, -, \times, \div\} \cup \mathbb{Z}$
- $Expr(\Omega)$ be the set of expressions over Ω
  - Evaluation: $Val : Expr(\Omega) \to \mathbb{R}$ (partial)
- Basic Problem: Given e and $p \in \mathbb{R}$
  - Compute a p-bit (rel/abs) approximation to $Val(e)$

Expression Evaluation Problem

- Ω be set of real operators (partial functions)
  - E.g., $\Omega = \{+, -, \times, \div\} \cup \mathbb{Z}$
- $Expr(\Omega)$ be the set of expressions over Ω
  - Evaluation: $Val : Expr(\Omega) \to \mathbb{R}$ (partial)
- Basic Problem: Given e and $p \in \mathbb{R}$
  - Compute a p-bit (rel/abs) approximation to $Val(e)$

Expression Evaluation Problem

- Ω be set of real operators (partial functions)
  - ∗ E.g., $\Omega = \{+, -, \times, \div\} \cup \mathbb{Z}$
- $Expr(\Omega)$ be the set of expressions over Ω
  - ∗ Evaluation: $Val : Expr(\Omega) \to \mathbb{R}$ (partial)
- Basic Problem: Given e and $p \in \mathbb{R}$
  - ∗ Compute a p-bit (rel/abs) approximation to $Val(e)$

Expression Evaluation Problem

- Ω be set of real operators (partial functions)
  - ∗ E.g., $\Omega = \{+, -, \times, \div\} \cup \mathbb{Z}$
- $Expr(\Omega)$ be the set of expressions over Ω
  - ∗ Evaluation: $Val : Expr(\Omega) \to \mathbb{R}$ (partial)
- Basic Problem: Given e and $p \in \mathbb{R}$
  - ∗ Compute a p-bit (rel/abs) approximation to $Val(e)$

Precision-Driven Expression Evaluation

- Precision Bound versus Error Bound
- Up-Down Propagation:
  - ∗ Downward propagation of precision
  - ∗ Upward propagation of error
- Assume problem is solved at the leaves
- This is NOT lazy evaluation

Precision-Driven Expression Evaluation

- Precision Bound versus Error Bound
- Up-Down Propagation:
  - Downward propagation of precision
  - Upward propagation of error
- Assume problem is solved at the leaves
- This is NOT lazy evaluation

Precision-Driven Expression Evaluation

- Precision Bound versus Error Bound
- Up-Down Propagation:
  - Downward propagation of precision
  - Upward propagation of error
- Assume problem is solved at the leaves
- This is NOT lazy evaluation

Precision-Driven Expression Evaluation

- Precision Bound versus Error Bound
- Up-Down Propagation:
  - Downward propagation of precision
  - Upward propagation of error
- Assume problem is solved at the leaves
- This is NOT lazy evaluation

Precision-Driven Expression Evaluation

- Precision Bound versus Error Bound
- Up-Down Propagation:
  - Downward propagation of precision
  - Upward propagation of error
- Assume problem is solved at the leaves
- This is NOT lazy evaluation

Basic Lemmas

- Let $\mu(x) := \lg |x|$. ($\mu(0) = -\infty$)
  - $*$ We may need estimates $\mu^-(x) \le \mu(x) \le \mu^+(x)$
- Let $x = y \circ z$ for some operation $\circ$
  - $*$ Compute $\widetilde{x} = \widetilde{y} \circ \widetilde{z}$, to some absolute precision
- To guarantee k relative bits in $\widetilde{x}$, it suffices:

Oper.	Op.Prec.	Prec. in $\widetilde{y}$	Prec. in $\widetilde{z}$	Remark
$x = yz$	∞	$k+1$	$k+2$	
$x = y \pm z$	∞	$k+1-\mu^-(x)$	$k+1-\mu^-(x)$	
$x = y/z$	$k+2$	$k+2$	$k+2$	$(k \ge 2)$
$x = \sqrt{y}$	$k+1$	$k+1$		
$\exp(y)$	$k+2$	$k+2+\mu^+(y)$		
$\log(y)$				(not possible)

Basic Lemmas

- Let $\mu(x) := \lg |x|$. ($\mu(0) = -\infty$)
  - $*$ We may need estimates $\mu^-(x) \leq \mu(x) \leq \mu^+(x)$
- Let $x = y \circ z$ for some operation $\circ$
  - $*$ Compute $\widetilde{x} = \widetilde{y} \circ \widetilde{z}$, to some absolute precision
- To guarantee k relative bits in $\widetilde{x}$, it suffices:

Oper.	Op.Prec.	Prec. in $\widetilde{y}$	Prec. in $\widetilde{z}$	Remark
$x = yz$	∞	$k+1$	$k+2$	
$x = y \pm z$	∞	$k+1-\mu^-(x)$	$k+1-\mu^-(x)$	
$x = y/z$	$k+2$	$k+2$	$k+2$	$(k \geq 2)$
$x = \sqrt{y}$	$k+1$	$k+1$		
$\exp(y)$	$k+2$	$k+2+\mu^+(y)$		
$\log(y)$				(not possible)

Basic Lemmas

- Let $\mu(x) := \lg|x|$. $(\mu(0) = -\infty)$
  - $*$ We may need estimates $\mu^-(x) \le \mu(x) \le \mu^+(x)$
- Let $x = y \circ z$ for some operation $\circ$
  - $*$ Compute $\widetilde{x} = \widetilde{y} \circ \widetilde{z}$, to some absolute precision
- To guarantee k relative bits in $\widetilde{x}$, it suffices:

Oper.	Op.Prec.	Prec. in $\widetilde{y}$	Prec. in $\widetilde{z}$	Remark
$x = yz$	∞	$k+1$	$k+2$	
$x = y \pm z$	∞	$k+1-\mu^-(x)$	$k+1-\mu^-(x)$	
$x = y/z$	$k+2$	$k+2$	$k+2$	$(k \ge 2)$
$x = \sqrt{y}$	$k+1$	$k+1$		
$\exp(y)$	$k+2$	$k+2+\mu^+(y)$		
$\log(y)$				(not possible)

Basic Lemmas

- Let $\mu(x) := \lg |x|$. $(\mu(0) = -\infty)$
  - We may need estimates $\mu^{-}(x) \leq \mu(x) \leq \mu^{+}(x)$
- Let $x = y \circ z$ for some operation $\circ$
  - Compute $\widetilde{x} = \widetilde{y} \circ \widetilde{z}$, to some absolute precision
- To guarantee k relative bits in $\widetilde{x}$, it suffices:

Oper.	Op.Prec.	Prec. in $\widetilde{y}$	Prec. in $\widetilde{z}$	Remark
$x = yz$	∞	$k+1$	$k+2$	
$x = y \pm z$	∞	$k+1-\mu^{-}(x)$	$k+1-\mu^{-}(x)$	
$x = y/z$	$k+2$	$k+2$	$k+2$	$(k \geq 2)$
$x = \sqrt{y}$	$k+1$	$k+1$		
$\exp(y)$	$k+2$	$k+2+\mu^{+}(y)$		
$\log(y)$				(not possible)

- To guarantee k absolute bits in $\widetilde{x}$, it suffices:

Oper.	Op.Prec.	Prec. in $\widetilde{y}$	Prec.in $\widetilde{z}$
yz	∞	$\max\{\frac{k+1}{2}, k+1+\mu^+(z)\}$	$\max\{\frac{k+1}{2}, k+1+\mu^+(z)\}$
$y+z$	∞	$k+1$	$k+1$
y/z	$k+1$	$k+2-\mu^-(z)$	$\max\{1-\mu^-(z), k+2-2\mu$
$\sqrt{y}$	$k+1$	$\max\{k+1, 1-\mu^-(y)/2\}$	
$\exp(y)$	$k+1$	$\max\{1, k+2+2^{\mu^+(y)+1}\}$	
$\log(y)$	$k+1$	$\max\{1-\mu^-(y), k+2-\mu^+(y)\}$	

- Three mutually recursive algorithms
  - $*$ Eval, Sign, Estimating $\mu^-(x), \mu^+(x)$
  - $*$ How to estimate $\mu^-(x)$?

- To guarantee k absolute bits in $\widetilde{x}$, it suffices:

Oper.	Op.Prec.	Prec. in $\widetilde{y}$	Prec.in $\widetilde{z}$
yz	∞	$\max\{\frac{k+1}{2}, k+1+\mu^+(z)\}$	$\max\{\frac{k+1}{2}, k+1+\mu^+(z)\}$
$y+z$	∞	$k+1$	$k+1$
y/z	$k+1$	$k+2-\mu^-(z)$	$\max\{1-\mu^-(z), k+2-2\mu$
$\sqrt{y}$	$k+1$	$\max\{k+1, 1-\mu^-(y)/2\}$	
$\exp(y)$	$k+1$	$\max\{1, k+2+2^{\mu^+(y)+1}\}$	
$\log(y)$	$k+1$	$\max\{1-\mu^-(y), k+2-\mu^+(y)\}$	

- Three mutually recursive algorithms
  - Eval, Sign, Estimating $\mu^-(x), \mu^+(x)$
  - How to estimate $\mu^-(x)$?

- To guarantee k absolute bits in $\widetilde{x}$, it suffices:

Oper.	Op.Prec.	Prec. in $\widetilde{y}$	Prec.in $\widetilde{z}$
yz	∞	$\max\{\frac{k+1}{2}, k+1+\mu^+(z)\}$	$\max\{\frac{k+1}{2}, k+1+\mu^+(z)\}$
$y+z$	∞	$k+1$	$k+1$
y/z	$k+1$	$k+2-\mu^-(z)$	$\max\{1-\mu^-(z), k+2-2$
$\sqrt{y}$	$k+1$	$\max\{k+1, 1-\mu^-(y)/2\}$	
$\exp(y)$	$k+1$	$\max\{1, k+2+2^{\mu^+(y)+1}\}$	
$\log(y)$	$k+1$	$\max\{1-\mu^-(y), k+2-\mu^+(y)\}$	

- Three mutually recursive algorithms
  - ∗ Eval, Sign, Estimating $\mu^-(x), \mu^+(x)$
  - ∗ How to estimate $\mu^-(x)$?

II. ZERO BOUNDS

Zero Bounds

- Let Ω be set of real operators (partial functions)
  - * E.g., $\Omega = \{+, -, \times, \div\} \cup \mathbb{Z}$
- Let $e \in Expr(\Omega)$ be an expression.
  - * Call $B > 0$ a **zero bound** for e if, whenever e is well-defined and not zero, then $|Val(e)| \geq B$.
- E.g., if $e = \sqrt{3} - \sqrt{2}$, then Cauchy's bound says $|e| \geq 1/11$ because e is the zero of $X^4 - 10x^2 + 1$.
- Classical bounds: not constructive or effective.

How to Use Zero Bounds

- Compute a numerical approximation $\widetilde{e}$ for e so that $|\widetilde{e} - e| < B/2$
    - If $|\widetilde{e}| \geq B$, then conclude that $\texttt{sign}(e)$ is the $\texttt{sign}(\widetilde{e})$
    - Otherwise, declare $e = 0$
- In practice, compute $\widetilde{e}$ incrementally
    - The zero bound is irrelevant unless $e = 0$
- This iteration is ONLY needed for $\pm$-nodes
    - Here is the CORE of Core Library!

How to Use Zero Bounds

- Compute a numerical approximation $\widetilde{e}$ for e so that $|\widetilde{e} - e| < B/2$
  - If $|\widetilde{e}| \geq B$, then conclude that $\mathtt{sign}(e)$ is the $\mathtt{sign}(\widetilde{e})$
  - Otherwise, declare $e = 0$
- In practice, compute $\widetilde{e}$ incrementally
  - The zero bound is irrelevant unless $e = 0$
- This iteration is ONLY needed for $\pm$-nodes
  - Here is the CORE of Core Library!

How to Use Zero Bounds

- Compute a numerical approximation $\widetilde{e}$ for e so that $|\widetilde{e} - e| < B/2$
  - * If $|\widetilde{e}| \geq B$, then conclude that $\texttt{sign}(e)$ is the $\texttt{sign}(\widetilde{e})$
  - * Otherwise, declare $e = 0$
- In practice, compute $\widetilde{e}$ incrementally
  - * The zero bound is irrelevant unless $e = 0$
- This iteration is ONLY needed for $\pm$-nodes
  - * Here is the CORE of Core Library!

How to Use Zero Bounds

- Compute a numerical approximation $\widetilde{e}$ for e so that $|\widetilde{e} - e| < B/2$
  - If $|\widetilde{e}| \geq B$, then conclude that $\mathtt{sign}(e)$ is the $\mathtt{sign}(\widetilde{e})$
  - Otherwise, declare $e = 0$
- In practice, compute $\widetilde{e}$ incrementally
  - The zero bound is irrelevant unless $e = 0$
- This iteration is ONLY needed for $\pm$-nodes
  - Here is the CORE of Core Library!

How to Use Zero Bounds

- Compute a numerical approximation $\widetilde{e}$ for e so that $|\widetilde{e} - e| < B/2$
  - * If $|\widetilde{e}| \geq B$, then conclude that $\mathtt{sign}(e)$ is the $\mathtt{sign}(\widetilde{e})$
  - * Otherwise, declare $e = 0$
- In practice, compute $\widetilde{e}$ incrementally
  - * The zero bound is irrelevant unless $e = 0$
- This iteration is ONLY needed for $\pm$-nodes
  - * Here is the CORE of Core Library!

How to Use Zero Bounds

- Compute a numerical approximation $\widetilde{e}$ for e so that $|\widetilde{e} - e| < B/2$
  - ∗ If $|\widetilde{e}| \geq B$, then conclude that $\mathtt{sign}(e)$ is the $\mathtt{sign}(\widetilde{e})$
  - ∗ Otherwise, declare $e = 0$
- In practice, compute $\widetilde{e}$ incrementally
  - ∗ The zero bound is irrelevant unless $e = 0$
- This iteration is ONLY needed for $\pm$-nodes
  - ∗ Here is the CORE of Core Library!

How to Use Zero Bounds

- Compute a numerical approximation $\widetilde{e}$ for e so that $|\widetilde{e} - e| < B/2$
  - ∗ If $|\widetilde{e}| \geq B$, then conclude that $\texttt{sign}(e)$ is the $\texttt{sign}(\widetilde{e})$
  - ∗ Otherwise, declare $e = 0$
- In practice, compute $\widetilde{e}$ incrementally
  - ∗ The zero bound is irrelevant unless $e = 0$
- This iteration is ONLY needed for $\pm$-nodes
  - ∗ Here is the CORE of Core Library!

Some Constructive Bounds

- Degree-Measure Bounds [Mignotte (1982)]
- Degree-Height, Degree-Length [Yap-Dubé (1994)]
- BFMS Bound [Burnikel et al (1989)]
- Eigenvalue Bounds [Scheinerman (2000)]
- Conjugate Bounds [Li-Yap (2001)]
- BFMSS Bound [Burnikel et al (2001)]
- k-ary Method [Pion-Yap (2002)]

An Example

- Consider the $e = \sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$.
- Assume $x = a/b$ and $y = c/d$ where a, b, c, d are L-bit integers. Then Li-Yap Bound is $28L + 60$ bits, BFMSS is $96L + 30$ and Degree-Measure is $80L + 56$.

L	50	100	500	5000
BFMS	0.637	9.12	101.9	202.9
Measure	0.063	0.07	1.93	15.26
BFMSS	0.073	0.61	1.95	15.41
Li-Yap	0.013	0.07	1.88	1.89

New k-Ary Rational Bounds

- Division expressions is a bottle neck
  - Rational input numbers introduces division!
  - E.g., binary floating point, decimal numbers.
- Overwhelming majoring of "real inputs" are k-ary rationals $(k = 2, 10)$
- THEOREM (Pion-Yap 2003)
  - $BFMSS[k] \geq BFMSS$
  - $Measure[k] \geq Measure$
- Implemented in Core Library

- Example of 2-ary Version of BFMSS:

	Method	BFMSS	Li-Yap	BFMSS[2] (new)
1	Bit-Bound function	$96L + 30$	$28L + 60$	$8L + 30$
2	Bit-Bound Range ($L = 53$)	4926-5118	2085-2165	426-462
3	Timing ($L = 53$, 1000 times)	46.7 s	8.35 s	3.58 s

Applications of EGC

- Meshing Generation
  - Killer App?
- Theorem Proving
  - Proving geometric theorems by random tests [Yap et al]
  - Kepler's Conjecture [Hale]
- Producing Model Solutions
  - Table Maker's Dilemma [Mueller]
  - Verifying Simplex Programs [Mehlhorn et al]
  - Testing Statistical Packages [McCullough]
- Symbolic Perturbation

Applications of EGC

- Meshing Generation
  - Killer App?
- Theorem Proving
  - Proving geometric theorems by random tests [Yap et al]
  - Kepler's Conjecture [Hale]
- Producing Model Solutions
  - Table Maker's Dilemma [Mueller]
  - Verifying Simplex Programs [Mehlhorn et al]
  - Testing Statistical Packages [McCullough]
- Symbolic Perturbation

Applications of EGC

- Meshing Generation
  - Killer App?
- Theorem Proving
  - Proving geometric theorems by random tests [Yap et al]
  - Kepler's Conjecture [Hale]
- Producing Model Solutions
  - Table Maker's Dilemma [Mueller]
  - Verifying Simplex Programs [Mehlhorn et al]
  - Testing Statistical Packages [McCullough]
- Symbolic Perturbation

Applications of EGC

- Meshing Generation
  - Killer App?
- Theorem Proving
  - Proving geometric theorems by random tests [Yap et al]
  - Kepler's Conjecture [Hale]
- Producing Model Solutions
  - Table Maker's Dilemma [Mueller]
  - Verifying Simplex Programs [Mehlhorn et al]
  - Testing Statistical Packages [McCullough]
- Symbolic Perturbation

Applications of EGC

- Meshing Generation
  - Killer App?
- Theorem Proving
  - Proving geometric theorems by random tests [Yap et al]
  - Kepler's Conjecture [Hale]
- Producing Model Solutions
  - Table Maker's Dilemma [Mueller]
  - Verifying Simplex Programs [Mehlhorn et al]
  - Testing Statistical Packages [McCullough]
- Symbolic Perturbation

Applications of EGC

- Meshing Generation
  - Killer App?
- Theorem Proving
  - Proving geometric theorems by random tests [Yap et al]
  - Kepler's Conjecture [Hale]
- Producing Model Solutions
  - Table Maker's Dilemma [Mueller]
  - Verifying Simplex Programs [Mehlhorn et al]
  - Testing Statistical Packages [McCullough]
- Symbolic Perturbation

Applications of EGC

- Meshing Generation
  - Killer App?
- Theorem Proving
  - Proving geometric theorems by random tests [Yap et al]
  - Kepler's Conjecture [Hale]
- Producing Model Solutions
  - Table Maker's Dilemma [Mueller]
  - Verifying Simplex Programs [Mehlhorn et al]
  - Testing Statistical Packages [McCullough]
- Symbolic Perturbation

* Handling degenerate data automatically

* Handling degenerate data automatically

Challenge of EGC

- Internally, all numbers are exact
  - * How to round to lower precision?
  - * This is necessary for cascading algorithms
- Geometric Rounding Problems
  - * Very little is known
- Challenge
  - * Given planar triangulation T and $p > 0$, Round T to precision $\leq p$
  - * RULES: Degeneration is allowed but no inversion, preserve proximity

Challenge of EGC

- Internally, all numbers are exact
  - * How to round to lower precision?
  - * This is necessary for cascading algorithms
- Geometric Rounding Problems
  - * Very little is known
- Challenge
  - * Given planar triangulation T and $p > 0$, Round T to precision $\leq p$
  - * RULES: Degeneration is allowed but no inversion, preserve proximity

Challenge of EGC

- Internally, all numbers are exact
  - ∗ How to round to lower precision?
  - ∗ This is necessary for cascading algorithms
- Geometric Rounding Problems
  - ∗ Very little is known
- Challenge
  - ∗ Given planar triangulation T and $p > 0$, Round T to precision $\leq p$
  - ∗ RULES: Degeneration is allowed but no inversion, preserve proximity

Challenge of EGC

- Internally, all numbers are exact
  - How to round to lower precision?
  - This is necessary for cascading algorithms
- Geometric Rounding Problems
  - Very little is known
- Challenge
  - Given planar triangulation T and $p > 0$, Round T to precision $\leq p$
  - RULES: Degeneration is allowed but no inversion, preserve proximity

- Why Robust FP-Type Algorithms are hard
  * They must round and compute at same time!

- Why Robust FP-Type Algorithms are hard
  * They must round and compute at same time!

Conclusions

- It is possible to provide a library to solve nonrobustness in general.
- Open Problem: Give a rounding algorithm for planar triangulations.
- Open Problem: Give a provably optimal precision-driven algorithm for the case of four arithmetic operations

Conclusions

- It is possible to provide a library to solve nonrobustness in general.
- Open Problem: Give a rounding algorithm for planar triangulations.
- Open Problem: Give a provably optimal precision-driven algorithm for the case of four arithmetic operations

Conclusions

- It is possible to provide a library to solve nonrobustness in general.
- Open Problem: Give a rounding algorithm for planar triangulations.
- Open Problem: Give a provably optimal precision-driven algorithm for the case of four arithmetic operations

Conclusions

- It is possible to provide a library to solve nonrobustness in general.
- Open Problem: Give a rounding algorithm for planar triangulations.
- Open Problem: Give a provably optimal precision-driven algorithm for the case of four arithmetic operations

EXERCISES

(1) Compute the BFMSS Bound for the expression $\sqrt{x}+\sqrt{y}-\sqrt{x+y+2\sqrt{xy}}$ when x, y are L-bit integers.
(2) Do the same as (1) when x, y are rational numbers whose numerator and denominator are L-bit integers.
(3) Do the same as (1) when x, y are L-bit binary floats. More precisely, I mean x and y have the form $B = m2^n$ (for some $m, n \in \mathbb{Z}$) where $|m| < 2^L$ and $2^n < 2^L$.

The BFMS and BFMSS bounds

FOR YOUR CONVENIENCE, I PUT SOME NOTES on THE BFMSS BOUND FROM [Mehlhorn-Yap] HERE.

We investigate the zero bound from Burnikel et al [**?**]. Call this the BFMSS Bound. But we begin with the older version known as the BFMS Bound [**?**]. In the absence of division, these two rules coincide.

Conceptually the BFMS approach first transforms a radical expression $e \in Expr(\Omega_2)$ to a quotient of two division-free expressions $U(e)$ and $L(e)$.

	e	$U(e)$	$L(e)$
1.	integer a	a	1
2.	$e_1 \pm e_2$	$U(e_1)L(e_2) \pm L(e_1)U(e_2)$	$L(e_1)L(e_2)$
3.	$e_1 \times e_2$	$U(e_1)U(e_2)$	$L(e_1)L(e_2)$
4.	$e_1 \div e_2$	$U(e_1)L(e_2)$	$L(e_1)U(e_2)$
5.	$\sqrt[k]{e_1}$	$\sqrt[k]{U(e_1)}$	$\sqrt[k]{L(e_1)}$

BFMS Rules for $U(e)$ and $L(e)$

If e is division-free, then $L(e) = 1$ and $Val(e)$ is an algebraic integer (i.e., a root of some monic integer polynomial). The following lemma is immediate from Table 1:

Lemma 1. $Val(e) = Val(U(e))/Val(L(e))$.

Table 1 should be viewed as transformation rules on expressions. We apply these rules recursive in a bottom-up fashion: suppose all the children v_i (say $i = 1, 2$) of a node v in the expression e has been transformed, and we now have the nodes $U(v_i), L(v_i)$ are available. Then we create the node $U(v), L(v)$ and construct the correspond subexpressions given by the table. The result is still a dag, but not rooted any more. The transformation $e \Rightarrow (U(e), L(e))$ is only conceptual – we do not really need to compute it. What we do compute are two real parameters $u(e)$ and $l(e)$ are maintained by the recursive rules in Table 2. The entries in this table are "shadows" of the corresponding entries in Table 1. (Where are they different?)

	e	$u(e)$	$l(e)$
1.	integer a	$\|a\|$	1
2.	$e_1 \pm e_2$	$u(e_1)l(e_2) + l(e_1)u(e_2)$	$l(e_1)l(e_2)$
3.	$e_1 \times e_2$	$u(e_1)u(e_2)$	$l(e_1)l(e_2)$
4.	$e_1 \div e_2$	$u(e_1)l(e_2)$	$l(e_1)u(e_2)$
5.	$\sqrt[k]{e_1}$	$\sqrt[k]{u(e_1)}$	$\sqrt[k]{l(e_1)}$
5'.	$\sqrt[k]{e_1}$	$\min\{\sqrt[k]{u(e_1)l(e_1)^{k-1}}, u(e_1)\}$	$\min\{l(e_1), \sqrt[k]{u(e_1)^{k-1}l(e_1)}\}$

BFMS (and BFMSS) Rules for $u(e)$ and $l(e)$

To explain the significance of $u(e)$ and $l(e)$, we define two useful quantities. If α is an algebraic number, define

$$MC(\alpha) := \max_{i=1}^{m} |\alpha_i| \tag{1}$$

where $\alpha_1, \ldots, \alpha_m$ are the conjugates of α. Thus $MC(\alpha)$ is the "maximum conjugate size" of α. In general, if $A(X)$ is any polynomial, we define $MC(A(X))$ to be the maximum of $|\alpha_i|$ where α_i range over the zeros of $A(X)$. For instance, $M(\alpha) \le M_0(\alpha)MC(\alpha)^d$ where $d = \deg(\alpha)$. Using $MC(\alpha)$ and $M_0(\alpha)$, we obtain an approach for obtaining zero bounds:

Lemma 2. *If $\alpha \neq 0$ and then*

$$|\alpha| \ge M_0(\alpha)^{-1} MC(\alpha)^{-d+1}$$

where $d = \deg(\alpha)$.

Proof. Let $d = \deg(\alpha)$. If the minimal polynomial of α is $a\prod_{i=1}^{m}(X - \alpha_i)$ then we have $a\prod_i |\alpha_i| \geq 1$. Thus, assuming $\alpha = \alpha_1$,

$$|\alpha| \geq \frac{1}{a\prod_{i=2}^{d}|\alpha_i|} \geq \frac{1}{aMC(\alpha)^{d-1}}.$$

Q.E.D.

The following theorem shows the significance of $u(e), l(e)$.

Theorem 3. *Let $e \in Expr(\Omega_2)$. Then $u(e)$ and $l(e)$ are upper bounds on $MC(U(e))$ and $MC(L(e))$, respectively.*

Proof. The result is true in the base case where e is an integer. In general, $U(e)$ and $L(e)$ are formed by the rules in Table 1. These rules uses only the operations of $\pm, \times, \sqrt[k]{\cdot}$. Applying the previous lemma, we see that $u(e)$ and $l(e)$ are indeed upper bounds on $MC(Val(U(e)))$ and $MC(Val(L(e)))$. **Q.E.D.**

Finally, we show how the BFMS Rules gives us a zero bound. It is rather similar to Lemma 2, except that we do not need to invoke $M_0(e)$.

Theorem 4. *Let $e \in Expr(\Omega_2)$ and $Val(e) \neq 0$. Then*

$$(u(e)^{D(e)^2-1}l(e))^{-1} \leq |Val(e)| \leq u(e)l(e)^{D(e)^2-1}. \quad (2)$$

If e is division-free,

$$(u(e)^{D(e)-1})^{-1} \leq |Val(e)| \leq u(e). \quad (3)$$

Proof. First consider the division-free case. In this case, $Val(e) = Val(U(e))$. Then $|Val(e)| \leq u(e)$ follows from Theorem 3. The lower bound on $|Val(e)|$ follows from lemma 2, since $M_0(e) = 1$ in the division-free case.
In the general case, we apply the division-free result to $U(e)$ and $L(e)$ separately. However, we need to estimate the degree of $U(e)$ and $L(e)$. We see that in the transformation from e to $U(e), L(e)$, the number of radical nodes in the dag doubles: each $\sqrt[k]{\cdot}$ is duplicated. This means that $\deg(U(e)) \leq \deg(e)^2$ and $\deg(L(e)) \leq \deg(e)^2$. From the division-free case, we conclude that

$$(u(e)^{D(e)^2-1})^{-1} \leq |Val(U(e))| \leq u(e).$$

and

$$(l(e)^{D(e)^2-1})^{-1} \leq |Val(L(e))| \leq l(e).$$

Thus $|Val(e)| = |Val(U(e))/Val(L(e))| \geq (l(e)u(e)^{D(e)^2-1})^{-1}$. The upper bound on $|Val(e)|$ is similarly shown. **Q.E.D.**

Example. Consider the expression $e_k \in Expr(\Omega_2)$ whose value is

$$\alpha_k = Val(e_k) = (2^{2^k} + 1)^{1/2^k} - 2. \quad (4)$$

Note that e_k is not literally the expression shown, since we do not have exponentiation in Ω_2. Instead, the expression begins with the constant 2, squaring k times, plus 1, then taking square-roots k times, and finally minus 2. Thus $u(e_k) = (2^{2^k}+1)^{1/2^k}+2 \le 5$. The degree bound $D(e_k) = 2^k$. Hence the BFMS Bound says

$$|\alpha_k| \ge u(e_k)^{1-2^k} \ge 5^{1-2^k}.$$

How tight is this bound? We have

$$\begin{aligned}
(2^{2^k}+1)^{1/2^k} - 2 &= 2\left(1+2^{-2^k}\right)^{1/2^k} - 2 \\
&= 2 \cdot e^{2^{-k}\ln(1+2^{-2^k})} - 2 \\
&\le 2 \cdot e^{2^{-k}2^{-2^k}} - 2 \\
&\le 2\left(1+2 \cdot 2^{-k}2^{-2^k}\right) - 2 \\
&= 2^{2-k-2^k}
\end{aligned}$$

using $\ln(1+x) \le x$ if $x > -1$ and $e^2 \le 1+2x$ if $0 \le x \le 1/2$. We also have

$$(2^{2^k}+1)^{1/2^k} - 2 = 2 \cdot e^{2^{-k}\ln(1+2^{-2^k})} - 2$$

$$
\begin{aligned}
&\geq 2 \cdot e^{2^{-k}2^{-2^k-1}} - 2 \\
&\geq 2\left(1 + 2^{-k}2^{-2^k-1}\right) - 2 \\
&\geq 2^{-k-2^k}
\end{aligned}
$$

using $e^x \geq 1 + x$. Hence $\alpha_k = \Theta(2^{-k-2^k})$. This example shows that the BFMS bound is, in a certain sense, asymptotically tight for the class of division-free expressions over Ω_2.

Improvements on the BFMS bound

The root bit-bound in (2) is quadratic in $D(e)$, while in (3) it is linear in $D(e)$. This quadratic factor can become a serious efficiency issue. Consider a simple example: $e = (\sqrt{x} + \sqrt{y}) - \sqrt{x + y + 2\sqrt{xy}}$ where x, y are L-bit integers. Of course, this expression is identically 0 for any x, y. The BFMS bound yields a root bit-bound of $7.5L + \mathcal{O}(1)$ bits. But in case, x and y are viewed as rational numbers (with denominator 1), the bit-bound becomes $127.5L + \mathcal{O}(1)$. This example shows that introducing rational numbers at the leaves of expressions has a major impact on the BFMS bound. In this section, we introduce two techniques to overcome division.

The BFMSS Bound. Returning to the case of radical expressions, we introduce another way to improve on BFMS. To avoid the doubling of radical nodes in the $e \mapsto (U(e), L(e))$ transformation, we change the rule in the last row of Table 2 as follows. When $e = \sqrt[k]{e_1}$, we use the alternative rule

$$u(e) = \sqrt[k]{u(e_1)l(e_1)^{k-1}}, \quad l(e) = l(e_1). \tag{5}$$

But one could equally use

$$u(e) = u(e_1), \quad l(e) = \sqrt[k]{u(e_1)^{k-1}l(e_1)}.$$

Yap noted that by using the symmetrized rule

$$u(e) = \min\{\sqrt[k]{u(e_1)l(e_1)^{k-1}}, u(e_1)\}, \qquad l(e) = \min\{l(e_1), \sqrt[k]{u(e_1)^{k-1}l(e_1)}\},$$

the new bound is provably never worse than the BFMS bound.
The BFMSS Bound also extends the rules to support general algebraic expressions (Ω_4 expressions).

REFERENCE

- Chapter 2 (number types) and Chapter 12 (zero bounds) of [Mehlhorn-Yap]
- Paper "On Guaranteed Accuracy Computation": http://cs.nyu.edu/yap/papers/

"A rapacious monster lurks within every computer, and it dines exclusively on accurate digits."
– B.D. McCullough (2000)

THE END