Icing
Supporting Fast-Math Style Optimizations in a Verified Compiler

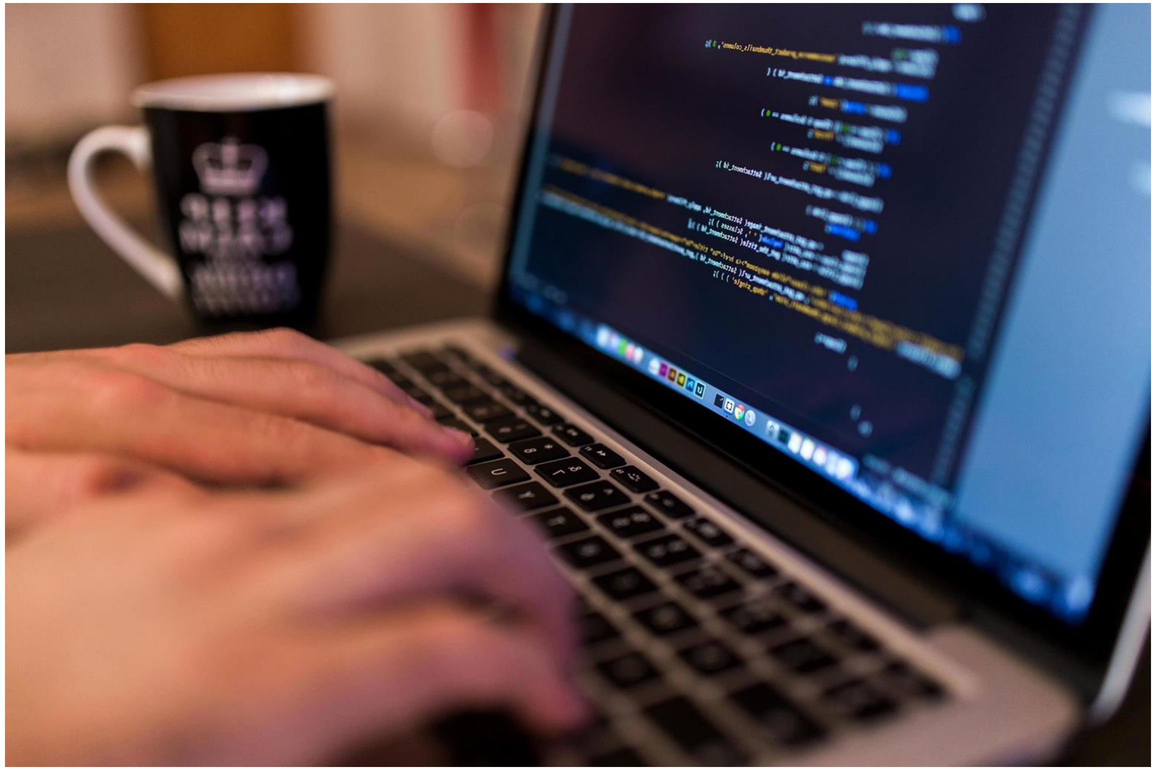
Heiko Becker, Eva Darulova,
Magnus Myreen, Zachary Tatlock

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

CHALMERS UNIVERSITY OF TECHNOLOGY

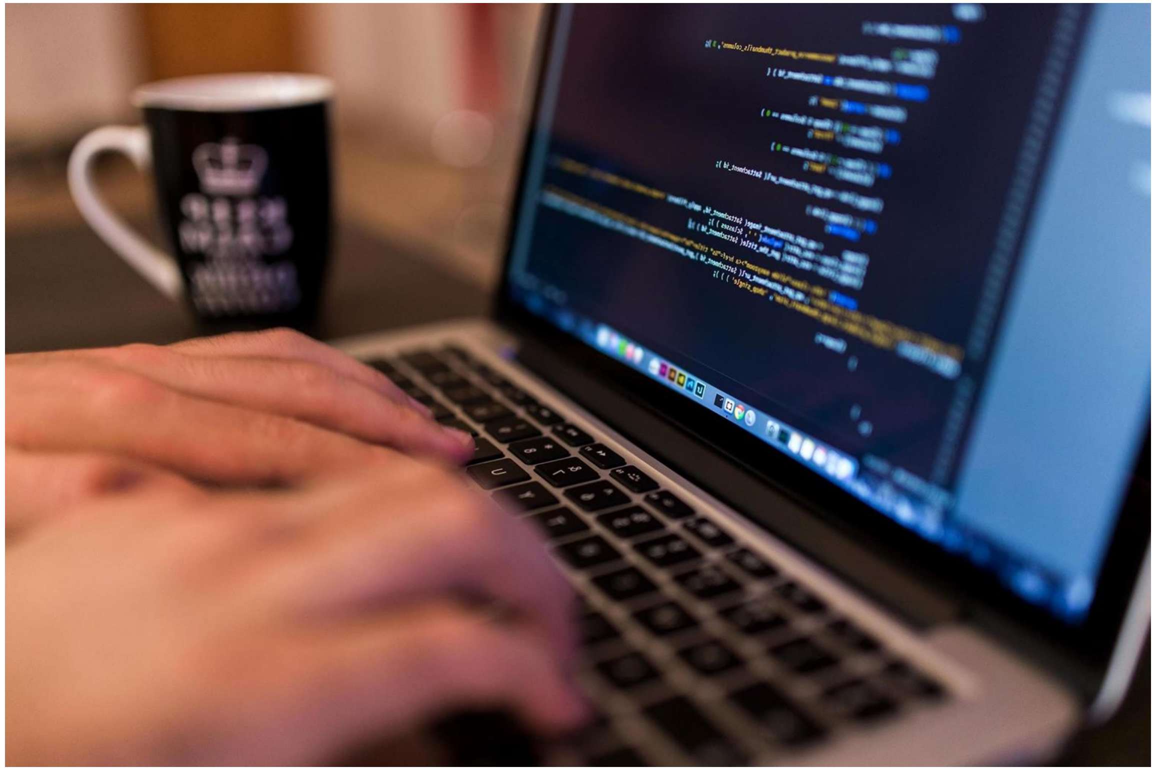
PLSE

How we develop programs



readability over performance

How we develop programs



readability over performance

compiler should make program fast

How we develop programs

readability over performance

make program fast

Compiler optimizations are

a vital part of

the development process

The need for understandable optimizations

What does gcc's ffast-math actually do?

Ask Question

130

I understand gcc's `--ffast-math` flag can greatly increase speed for float ops, and goes outside of IEEE standards, but I can't seem to find information on what is really happening when it's on. Can anyone please explain some of the details and maybe give a clear example of how something would change if the flag was on or off?

I did try digging through S.O. for similar questions but couldn't find anything explaining the workings of ffast-math.

44

performance math gcc floating-point fast-math

The need for understandable optimizations

What does gcc's ffast-math actually do?

Ask Question

130

I understand gcc's `--ffast-math` flag can greatly increase speed for float ops, and goes outside of IEEE standards, but I can't seem to find information on what is really happening when it's on. Can anyone please explain some of the details and maybe give a clear example of how something would change if the flag was on or off?

44

I did try digging through S.O. for similar questions but couldn't find anything explaining the workings of ffast-math.

performance math gcc floating-point fast-math

asked 7 years, 10 months ago

viewed 41,233 times

active 10 months ago

The need for understandable optimizations

What does gcc's ffast-math actually do?

Ask Question

130

I understand gcc's `--ffast-math` flag can greatly increase speed for float ops, and goes outside of IEEE standards, but I can't seem to find information on what is really happening when it's on. Can anyone please explain some of the details and maybe give a clear example of how <u>something would change if the flag was on or off</u>?

I did try digging through S.O. for similar questions but couldn't find anything explaining the workings of ffast-math.

44

performance | math | gcc | floating-point | fast-math

asked 7 years, 10 months ago

viewed 41,233 times

active 10 months ago

The long road of compiler verification

1967: *McCarthy, Painter*;

Correctness of a Compiler for Arithmetic Expressions (Integer's only)

2009: *Leroy*;

Formal Verification of a Realistic Compiler

2019: *Lööw et al*;

Verified Compilation on a Verified Processor

The long road of compiler verification

1967: *McCarthy, Painter*;

Correctness of a Compiler for Arithmetic Expressions (Integer's only)

2009: *Leroy*;

Formal Verification of a Realistic Compiler ← CompCert C compiler

2019: *Lööw et al*;

Verified Compilation on a Verified Processor

The long road of compiler verification

1967: *McCarthy, Painter*;

Correctness of a Compiler for Arithmetic Expressions (Integer's only)

2009: *Leroy*;

Formal Verification of a Realistic Compiler ← CompCert C compiler

2019: *Lööw et al*;

Verified Compilation on a Verified Processor ← CakeML & Silver

The long road of compiler verification

1967: *McCarthy, Painter*;

Correctness of a Compiler for Arithmetic Expressions (Integer's only)

2009: *Leroy*;

Formal Verification of a R

Where are floating-points?

2019: *Lööw et al*;

Verified Compilation on a Verified Processor

The long road of compiler verification

1967: *McCarthy, Painter*;

Correctness of a Compiler for Arithmetic Expressions (Integer’s only)

2009: *Leroy*;

Formal Verification of a Realistic Compiler

2015: *Boldo et al.*;

Verified Compilation of Floating-Point Programs

2019: *Lööw et al*;

Verified Compilation on a Verified Processor

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ….)

Verified Compilers (CakeML, ...)

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ….)

- apply aggressive optimizations

Verified Compilers (CakeML, ...)

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ….)

- apply aggressive optimizations
- do not preserve IEEE754 semantics

Verified Compilers (CakeML, ...)

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ...)

- apply aggressive optimizations
- do not preserve IEEE754 semantics
- give no guarantees on the result

Verified Compilers (CakeML, ...)

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ….)

- apply aggressive optimizations
- do not preserve IEEE754 semantics
- give no guarantees on the result

Verified Compilers (CakeML, ...)

- apply no floating-point optimizations

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ...)

- apply aggressive optimizations
- do not preserve IEEE754 semantics
- give no guarantees on the result

Verified Compilers (CakeML, ...)

- apply no floating-point optimizations
- fully preserve IEEE754 semantics

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ...)

- apply aggressive optimizations
- do not preserve IEEE754 semantics
- give no guarantees on the result

Verified Compilers (CakeML, ...)

- apply no floating-point optimizations
- fully preserve IEEE754 semantics
- guarantee preserving literal meaning

The state-of-the-art for fast-math

Unverified Compilers (gcc, clang, ….)

- apply aggressive o
- do not preserve IE
- give no guarantees

Verified Compilers (CakeML, ...)

- oint optimizations
- 754 semantics
- ing literal meaning

We need a semantics to handle fast-math optimizations in verified compilers

Contributions

Icing, a **nondeterministic** semantics for floating-points:

- Support for subset of **gcc's fast-math optimizations**
- Optimization with **fine-grained control**
- Implementation of **three optimizers**
- Verification in HOL4
- Connection to CakeML

Optimizations in Icing

$$
\begin{tikzcd}
\text{Source} \arrow[d] & \text{Target} \arrow[d] \\
\text{s} \arrow[r] & \text{t}
\end{tikzcd}
$$

Example Optimizations:

$$a + b \longrightarrow b + a$$

$$a \times b \longrightarrow b \times a$$

$$a + (b + c) \longrightarrow (a + b) + c$$

$$a \times (b \times c) \longrightarrow (a \times b) \times c$$

$$a \times b + c \longrightarrow FMA(a, b, c)$$

Optimizations in Icing

$$\begin{array}{ccc} \text{Source} & & \text{Target} \\ \downarrow & & \downarrow \\ s & \longrightarrow & t \end{array}$$

Commutativity
(preserves IEEE754)

Example Optimizations:

$$a + b \longrightarrow b + a$$

$$a \times b \longrightarrow b \times a$$

$$a + (b + c) \longrightarrow (a + b) + c$$

$$a \times (b \times c) \longrightarrow (a \times b) \times c$$

$$a \times b + c \longrightarrow FMA(a, b, c)$$

Optimizations in Icing

Source

Target

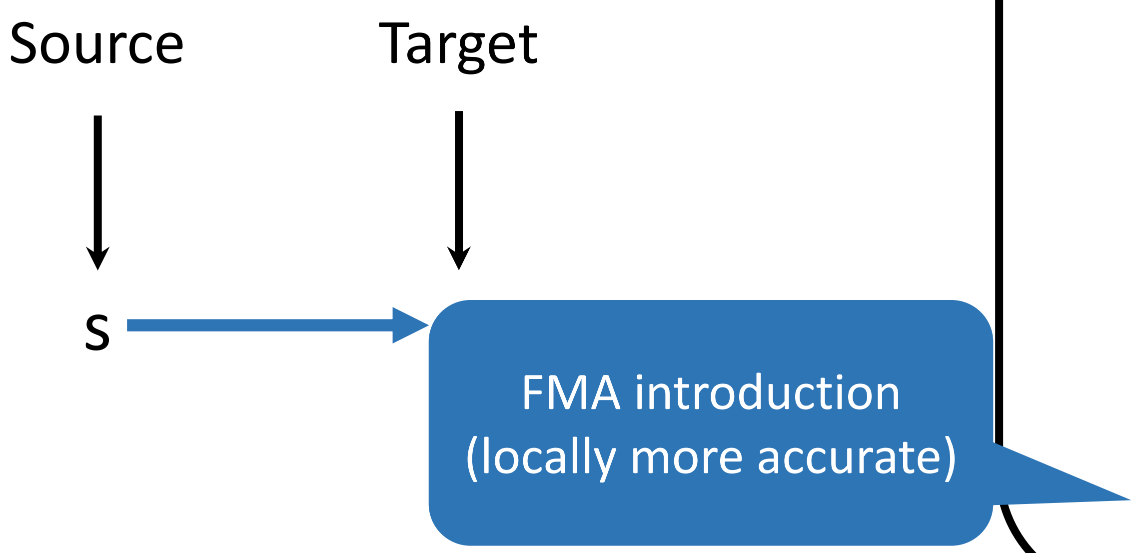
s → t

Associativity
(no IEEE754)

Example Optimizations:

$a + b$	$b + a$
$a \times b$	$b \times a$
$a + (b + c)$	$(a + b) + c$
$a \times (b \times c)$	$(a \times b) \times c$
$a \times b + c$	$FMA(a, b, c)$

Optimizations in Icing



Example Optimizations:

$a + b \rightarrow b + a$

$a \times b \rightarrow b \times a$

$a + (b + c) \rightarrow (a + b) + c$

$a \times (b \times c) \rightarrow (a \times b) \times c$

$a \times b + c \rightarrow FMA(a, b, c)$

Optimizations in Icing

$$
\begin{tikzcd}
\text{Source} \arrow[d] & \text{Target} \arrow[d] \\
s \arrow[r] & t
\end{tikzcd}
$$

Example Optimizations:

$$a + b \longrightarrow b + a$$

$$a \times b \longrightarrow b \times a$$

$$a + (b + c) \longrightarrow (a + b) + c$$

$$a \times (b \times c) \longrightarrow (a \times b) \times c$$

$$a \times b + c \longrightarrow FMA(a, b, c)$$

Floating-Point Values in Icing

IEEE754:

$$3.5 + 2.0 = 5.5$$

Icing:

Floating-Point Values in Icing

IEEE754:

$3.5 + 2.0 = 5.5 \longleftarrow$ floating-point word

Icing:

Floating-Point Values in Icing

IEEE754:

$3.5 + 2.0 = 5.5$ ⟵ floating-point word

Icing:

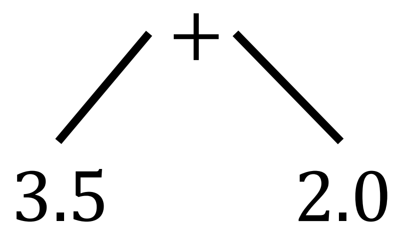
$3.5 + 2.0 =$

Floating-Point Values in Icing

IEEE754:

$3.5 + 2.0 = 5.5 \longleftarrow$ floating-point word

Icing:

$3.5 + 2.0 =$ 

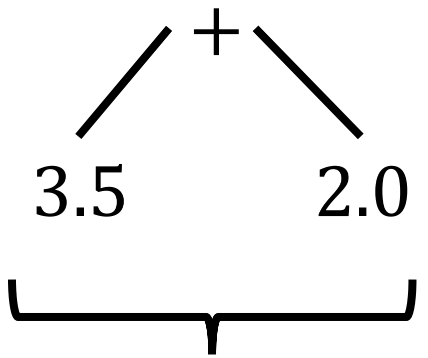
Floating-Point Values in Icing

IEEE754:

$3.5 + 2.0 = 5.5 \longleftarrow$ floating-point word

Icing:

$3.5 + 2.0 =$



value tree for addition

Floating-Point Values in Icing

IEEE754:

$3.5 + 2.0 = 5.5 \longleftarrow$ floating-point word

$3.5 + (2.0 + 1.5) = 12.25$

Icing:

$$3.5 + 2.0 = \underbrace{\begin{array}{ccc} & + & \\ / & & \backslash \\ 3.5 & & 2.0 \end{array}}_{\text{value tree for addition}}$$

Floating-Point Values in Icing

IEEE754:

$$3.5 + 2.0 = 5.5 \longleftarrow \text{floating-point word}$$

$$3.5 + (2.0 + 1.5) = 12.25$$

Icing:

$$3.5 + 2.0 = \underbrace{\begin{tikzcd}[column sep=small] & + \arrow[dl, no head] \arrow[dr, no head] & \\ 3.5 & & 2.0 \end{tikzcd}}_{\text{value tree for addition}}$$

$$3.5 + (2.0 + 1.5) =$$

Floating-Point Values in Icing

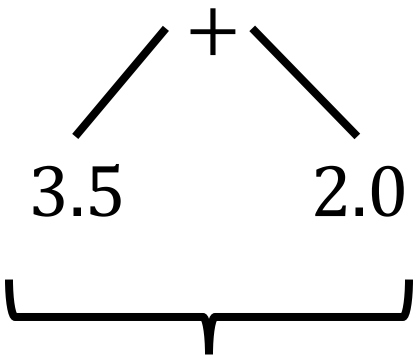
IEEE754:

$$3.5 + 2.0 = 5.5 \longleftarrow \text{floating-point word}$$

$$3.5 + (2.0 + 1.5) = 12.25$$

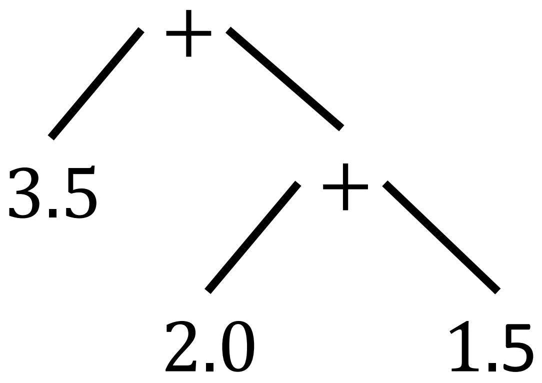
Icing:

$$3.5 + 2.0 =$$



value tree for addition

$$3.5 + (2.0 + 1.5) =$$



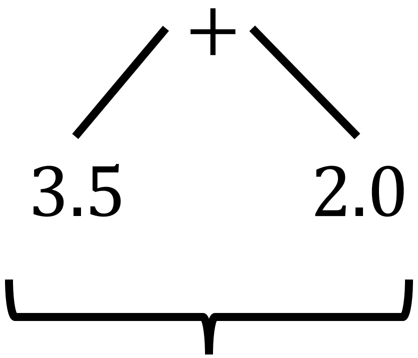
Floating-Point Values in Icing

IEEE754:

$3.5 + 2.0 = 5.5 \longleftarrow$ flo

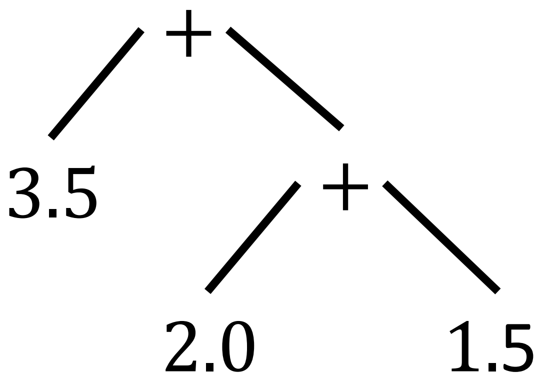
$3.5 + (2.0 + 1.5) = 12.$

Icing:



value tree for addition

$.5) =$



Icing's semantics

Allowed Optimization:

$$a \times b + c \longrightarrow FMA(a, b, c)$$

$$a \times b \longrightarrow b \times a$$

```
opt:(x * 2.4 + y)
```

Icing’s semantics

Allowed Optimization:

$a \times b + c \rightarrow FMA(a, b, c)$

$a \times b \rightarrow b \times a$

Included in the semantics

```
opt:(x * 2.4 + y)
```

Icing’s semantics

Allowed Optimization:

$$a \times b + c \longrightarrow FMA(a, b, c)$$

$$a \times b \longrightarrow b \times a$$

Included in the semantics

fine-grained control → `opt:(x * 2.4 + y)`

Icing’s semantics

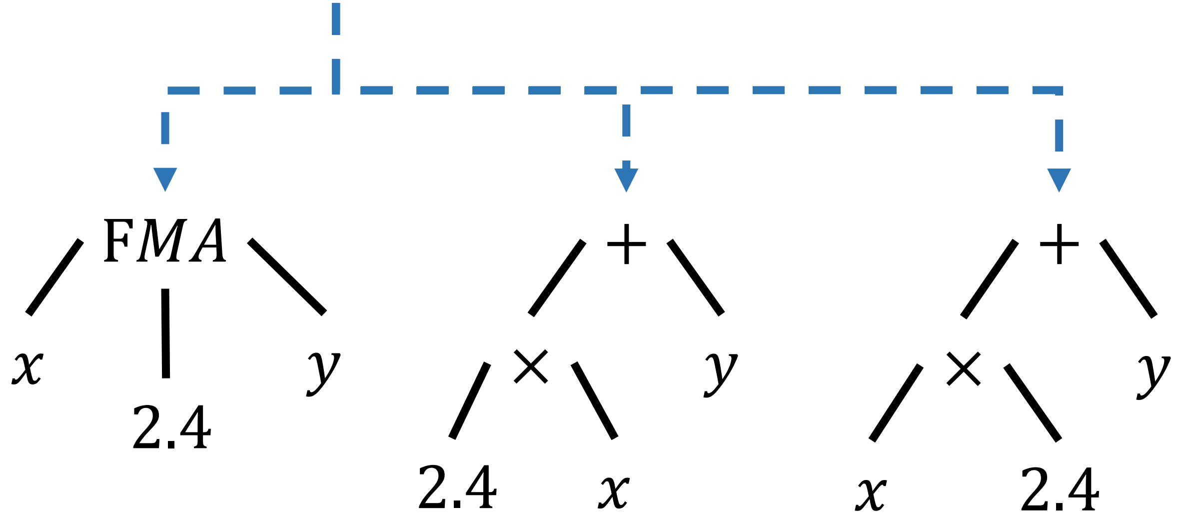
Allowed Optimization:

$a \times b + c \longrightarrow FMA(a, b, c)$

$a \times b \longrightarrow b \times a$

Included in the semantics

fine-grained control → `opt:(x * 2.4 + y)`



Icing's semantics

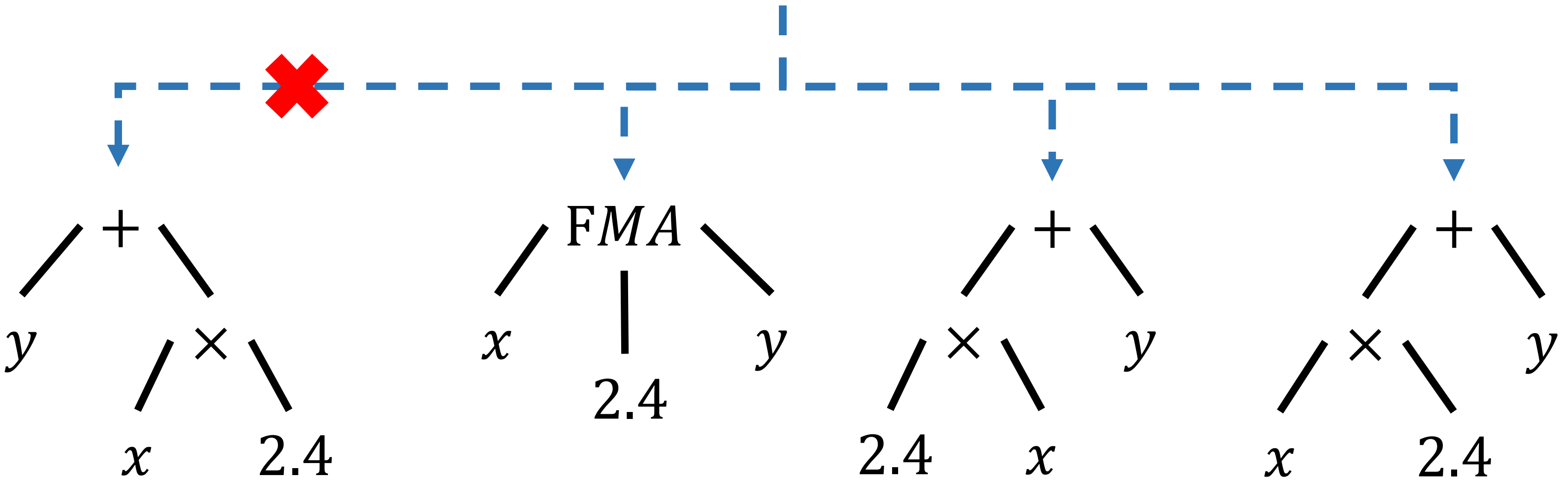
Allowed Optimization:

$$a \times b + c \longrightarrow FMA(a, b, c)$$

$$a \times b \longrightarrow b \times a$$

Included in the semantics

fine-grained control ⟶ `opt:(x * 2.4 + y)`



Icing's semantics

Allowed Optimization:

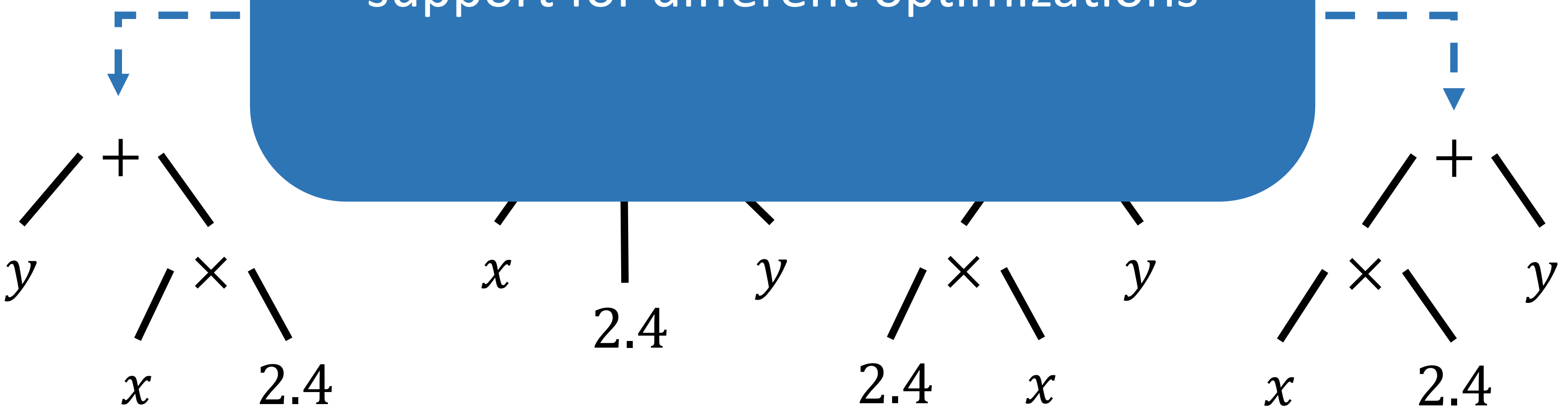
$a \times b + c \longrightarrow FMA(a, b, c)$

$a \times b \longrightarrow b \times$

Included in the semantics

fine-grained contr

Icing: a direct fit for fast-math with fine-grained control and support for different optimizations



Modelling the state-of-the-art

Unverified Compilers (gcc, clang, ….)

- aggressive optimizations
- no IEEE754 semantics
- no guarantees on the result

Verified Compilers (CakeML, ...)

- no floating-point optimizations
- IEEE754 semantics
- introduces no new behaviour

Icing provides:

greedy optimizer

IEEE754 Translator

Modelling the state-of-the-art

Unverified Compilers (gcc, clang, ….)

- aggressive optimizations
- no IEEE754 semantics
- no guarantees on the result

Verified Compilers (CakeML, ...)

- no floating-point optimizations
- IEEE754 semantics
- introduces no new behaviour

Icing provides:

greedy optimizer ✔

IEEE754 Translator ✔

What can we prove about the optimizers

Greedy optimizer:

$\vdash$ if evaluating the greedily optimized program p returns v then v is a possible result of evaluating p with the optimizations of the greedy optimizer

The greedy optimizer applies optimizations with respect to Icing semantics

IEEE754 translator:

$\vdash$ after running the IEEE754 translator on program p no optimizations can be applied by Icing semantics

$\vdash$ after running the IEEE754 translator on program p Icing semantics are deterministic no matter which optimizations are allowed

The IEEE754 translator preserves literal meaning (like CompCert/CakeML)

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

```
x * (y + z)
```

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

```
x * (y + z)
```

$\boxed{\text{Compiler}}$

```
x * y + x * z
```

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

```
x * (y + z)
```

Compiler

```
x * y + x * z
```

Semantics

```
x1 * y + x2 * z
```

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

```
x * (y + z)
```

Compiler

```
x * y + x * z
```

Semantics

```
x1 * y + x2 * z
```

What if these do not match?

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

```
x * (y + z)
```

Semantics

```
x‘ * (y + z)
```

Compiler

```
x * y + x * z
```

Semantics

```
x1 * y + x2 * z
```

What if these do not match?

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

`x * (y + z)`

Semantics

Compiler

`x' * (y + z)`

`x * y + x * z`

`x'` rewrites into `x1` and `x2`

Semantics

`x1 * y + x2 * z`

What if these do not match?

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$

x * (y + z)

Semantics

Compiler

x' * (y + z)

x * y + x * z

x' rewrites into x1 and x2

Semantics

Semantics

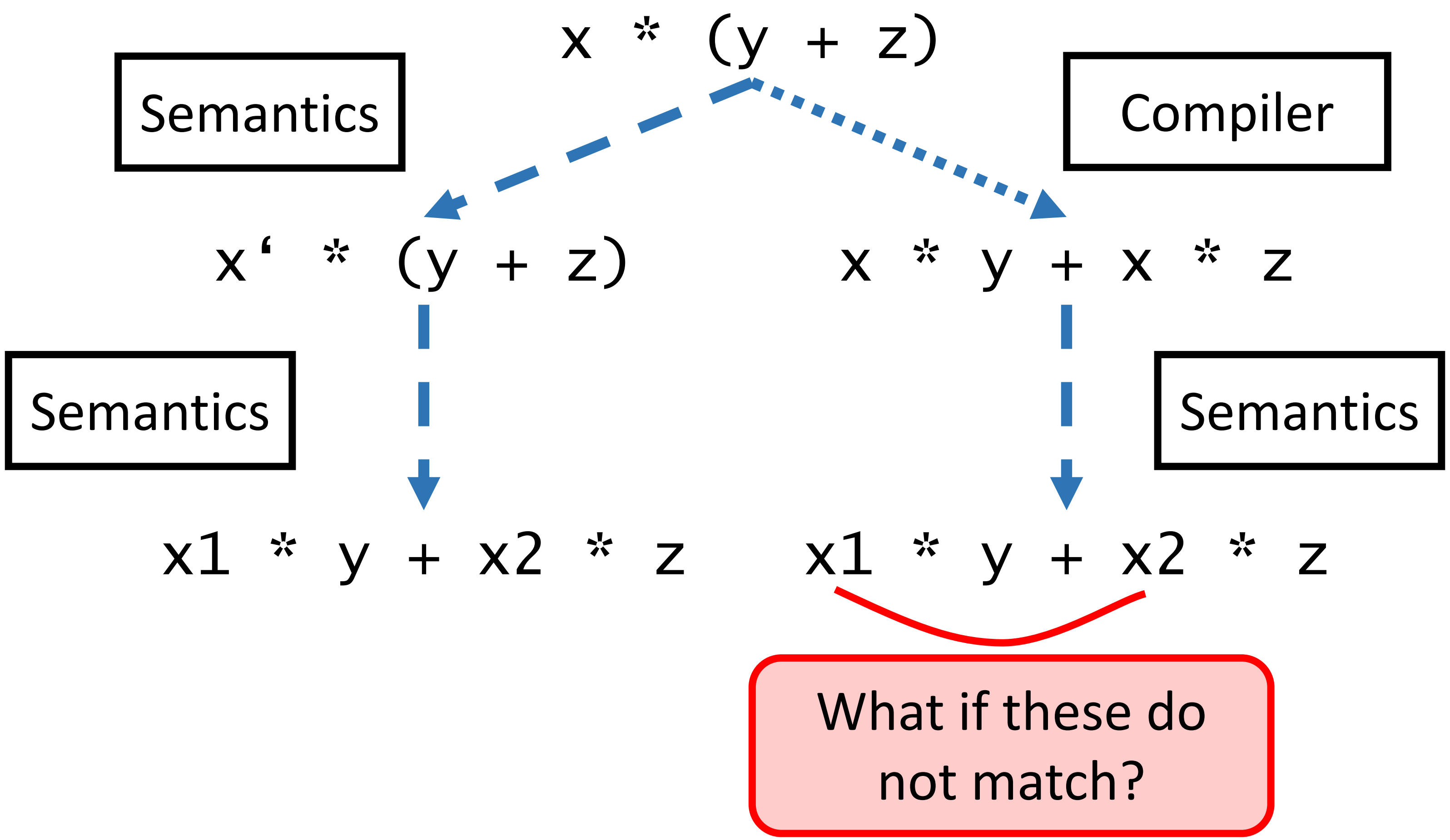
x1 * y + x2 * z

x1 * y + x2 * z

What if these do not match?

Distributivity in Icing

$$a \times (b + c) \longrightarrow a \times b + a \times c$$



x' rewrites into x1 and x2

Conditionals: Tricky!
(see paper)

Handling more of gcc's rewrites

Enable fast-math mode. This defines the `__FAST_MATH__` preprocessor mac
lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `* c == a * c + b * c`),
- operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

Official clang documentation

Handling more of gcc's rewrites

Enable fast-math mode. This defines the `__FAST_MATH__` preprocessor mac
lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g.
  `* c == a * c + b * c`),
- operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

Official clang documentation

Handling more of gcc's rewrites

$$\begin{array}{ccc} \text{Source} & & \text{Target} \\ \downarrow & & \downarrow \\ s & \longrightarrow & t \end{array}$$

Enable fast-math mode. This defines the `__FAST_MATH__` preprocessor mac
lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `* c == a * c + b * c`),
- operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

Official clang documentation

Handling more of gcc's rewrites

Source

Target

$s \longrightarrow t$

gcc:

$\text{isNaN}(c) \longrightarrow F$

Enable fast-math mode. This defines the `__FAST_MATH__` preprocessor mac
lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `* c == a * c + b * c`),
- operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

Official clang documentation

Handling more of gcc's rewrites

Precondition Source Target

$P \dashrightarrow s \longrightarrow t$

gcc:

isNaN (c) $\longrightarrow$ F

Precondition allows to check condition
before applying a rewrite

Enable fast-math mode. This defines the `__FAST_MATH__` preprocessor mac
lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `* c == a * c + b * c`),
- operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

Official clang documentation

Handling more of gcc's rewrites

Precondition Source Target

$P \dashrightarrow s \longrightarrow t$

Precondition allows to check condition before applying a rewrite

Icing:	gcc:
$c = c \dashrightarrow$	isNaN (c) $\longrightarrow$ F

Enable fast-math mode. This defines the `__FAST_MATH__` preprocessor mac
lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `* c == a * c + b * c`),
- operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

Official clang documentation

Handling more of gcc's rewrites

Precondition Source Target

$P \dashrightarrow s \longrightarrow t$

Precondition allows to check condition before applying a rewrite

Only NaN unequal to itself

Icing:

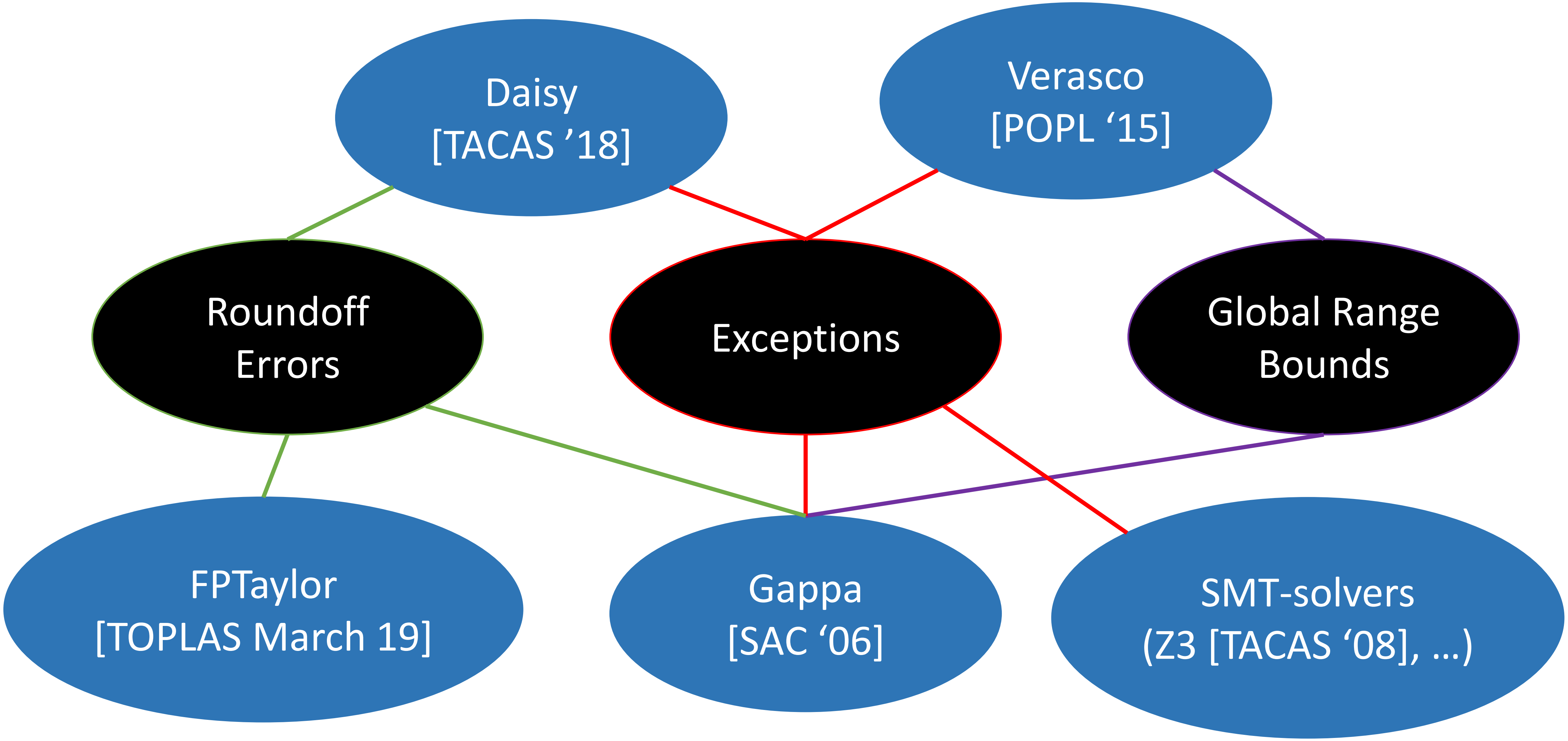
$c = c \dashrightarrow$ isNaN (c) $\longrightarrow$ F

Enable fast-math mode. This defines the `__FAST_MATH__` preprocessor mac lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `* c == a * c + b * c`),
- operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

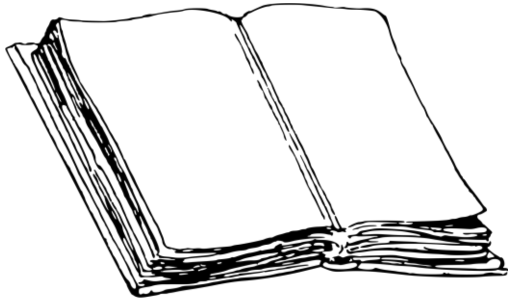
Official clang documentation

How can the preconditions be checked



Icings interface to external tools

Discharge checks in-place



Record assumed proposition

a, b, c variables $\dashrightarrow a \times (b + c) \rightarrow a \times b + a \times c$

simple local check

$\Longrightarrow$ checked before applying optimization

$c = c \dashrightarrow isNaN(c) \longrightarrow False$

complex global property

$\Longrightarrow$ checked offline after compiling

What does gcc's fast-math actually do?

Nondeterministic Icing
(with optimizations)

↑

deterministic Icing
(without optimizations)

↕

CakeML source

Outlook:

- integrate with external tools
- verify larger optimizations
- integrate into CakeML semantics