Automatic Detection of Floating-Point Exceptions

Earl T. Barr
Thanh Vo, Vu Le, and Zhendong Su

Presented by Saheel

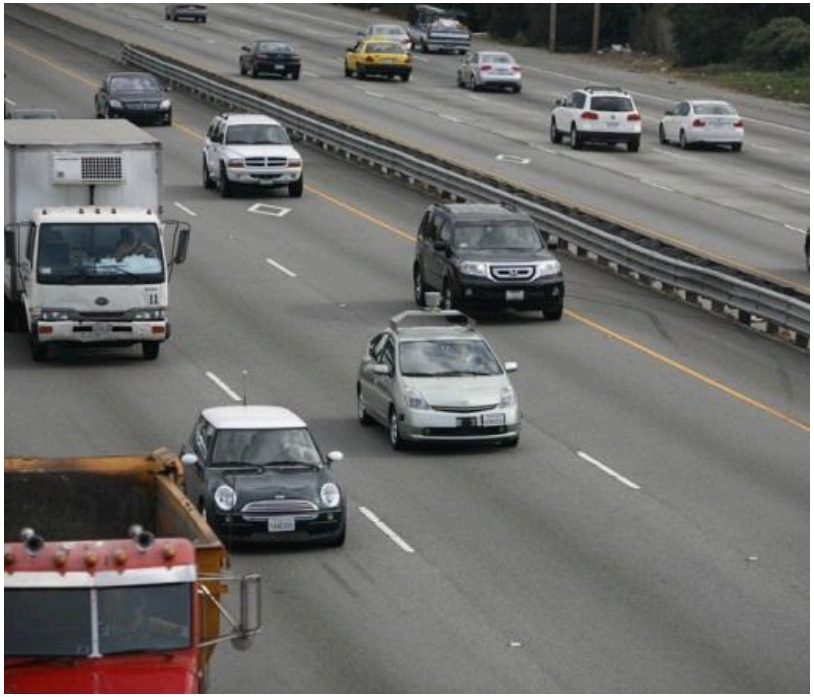
Numerical Code

- Ubiquitous
- Safety critical

Google's Driverless Car

Numerical Code

- Ubiquitous
- Safety critical



Google's Driverless Car

Numerical Code

- Ubiquitous
- Safety critical
- **Hard to get right**

Contribution

Ariadne: A practical symbolic analysis engine that finds witnesses for floating-point exceptions

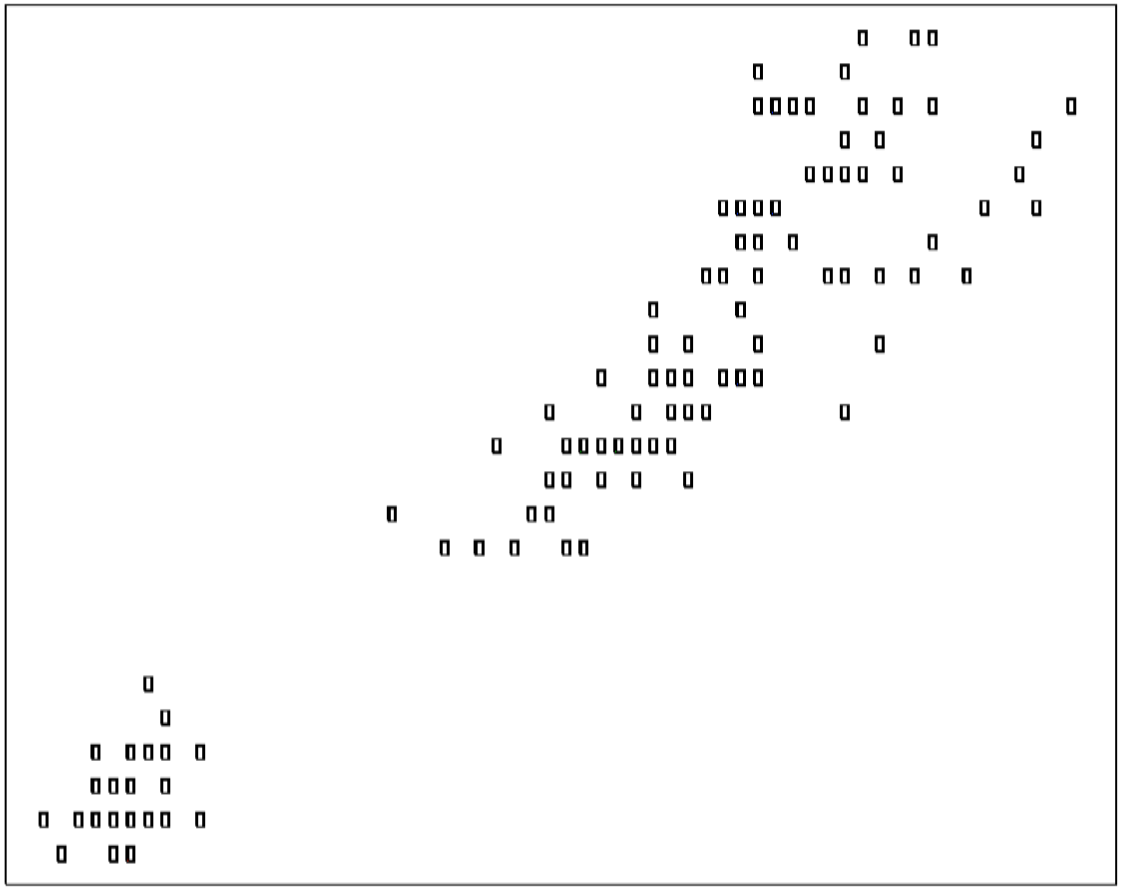
- Realizes a sweet spot in balance of analysis cost and effectiveness
- Finds over 2,000 **real**, non-rounding, FP exceptions in the GNU Scientific Library

Key Insight

Exception-triggering inputs form dense clusters

- If one input triggers an exception, many do, and one will do so over both real and floating-point arithmetic.

Exception Triggering Inputs

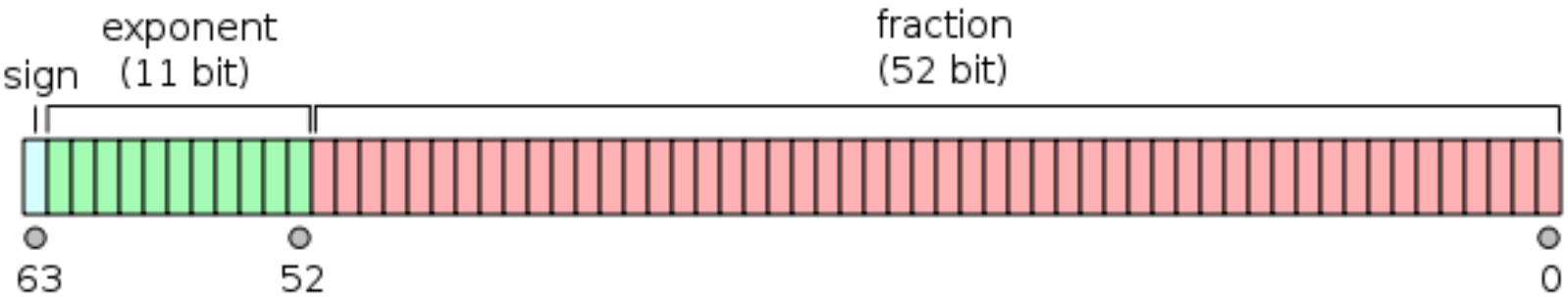


Outline

- **Floating-Point Primer**
- Ariadne Frontend
- Ariadne Backend
- Evaluation

Floating-Point Format

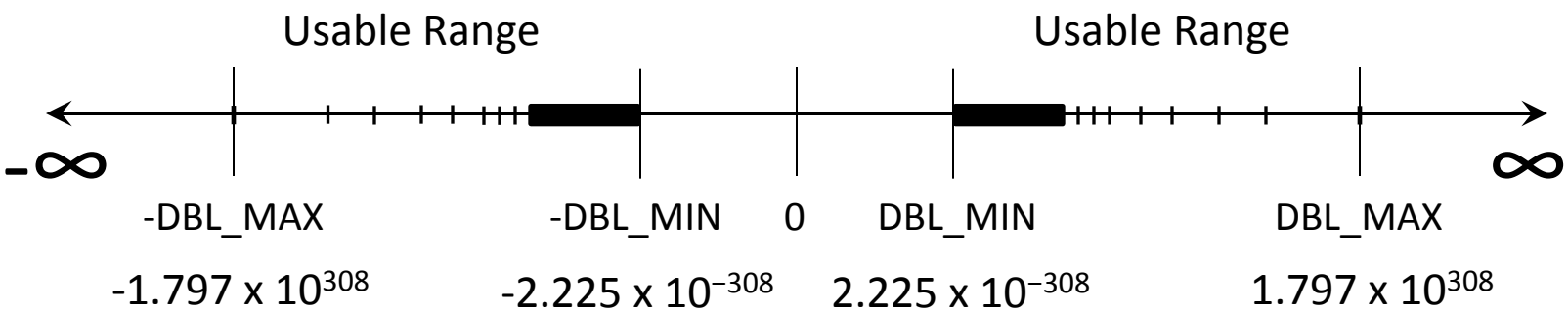
$$\pm b_0.b_1b_2\ldots b_{p-1} \times 2^E$$



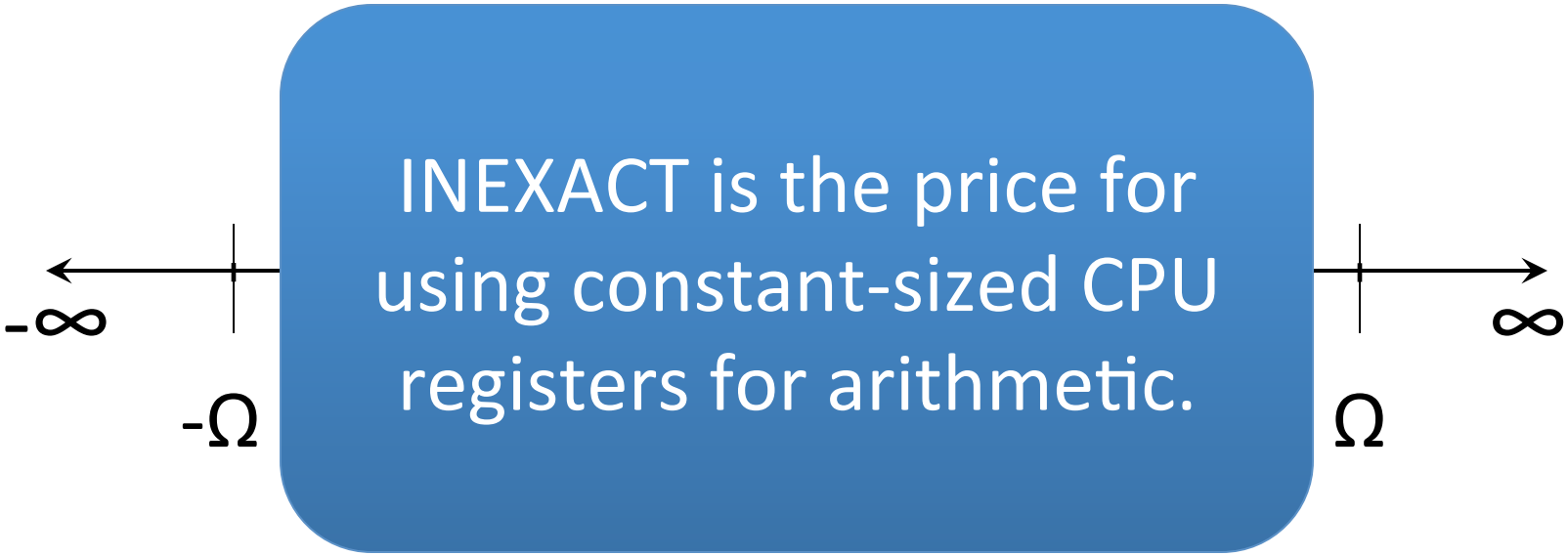
$$-1^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exponent} - \text{bias}}$$

b_0

Floating-Point Number Line



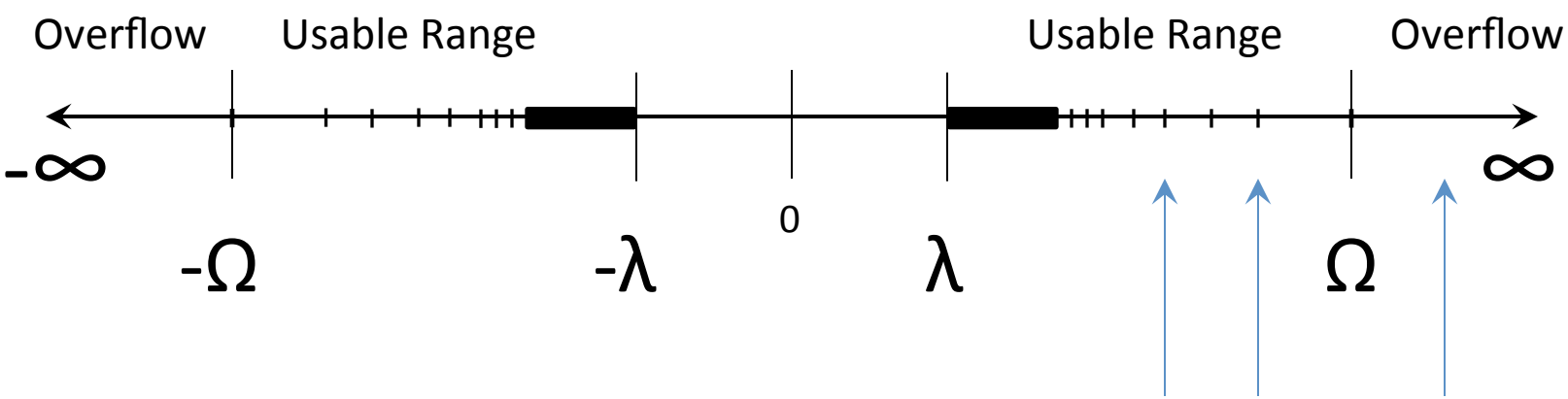
Floating-Point Number Line



x is rounded
INEXACT exception thrown

x

Floating-Point Number Line

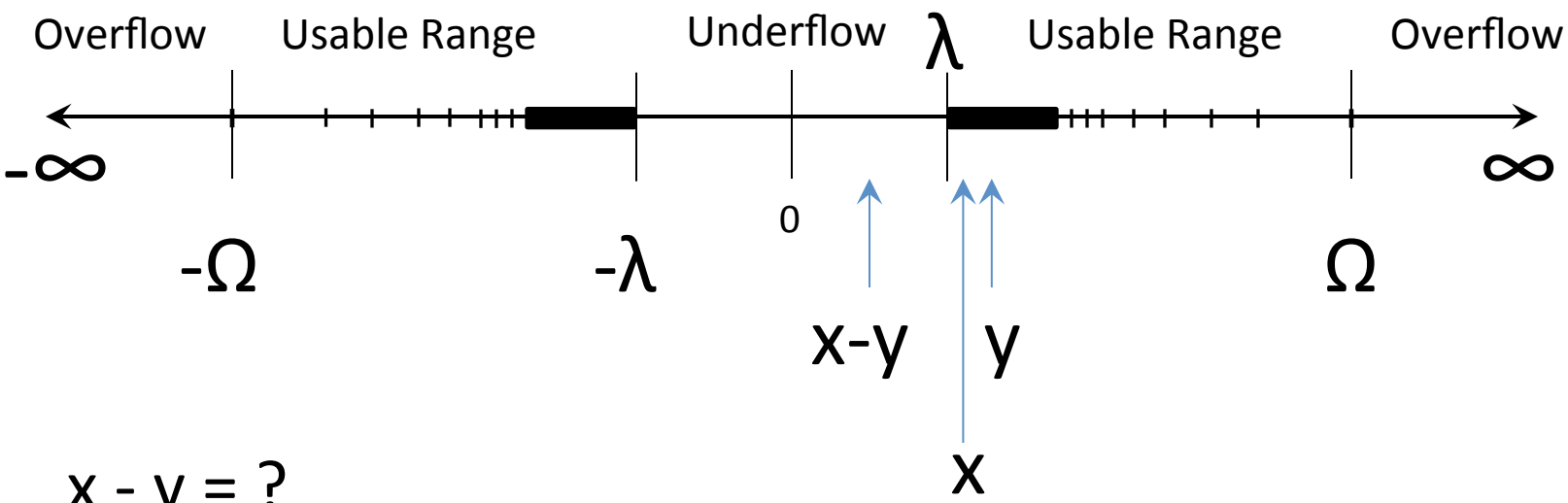


x + y = ∞

OVERFLOW exception thrown

x y x+y

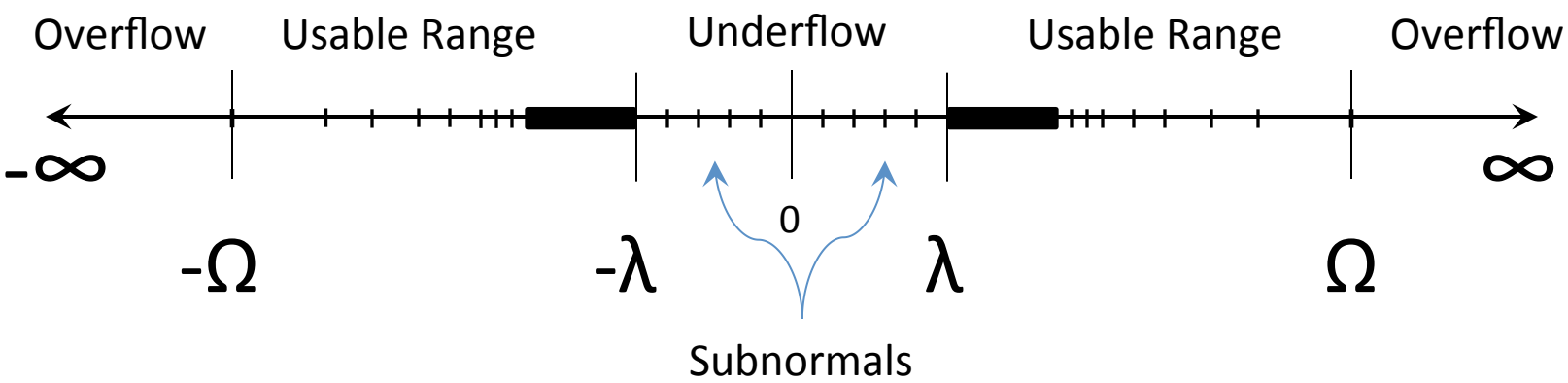
Floating-Point Number Line



x - y = ?

UNDERFLOW exception

Floating-Point Number Line



Floating-Point Exceptions

Exception	Example	Result
INVALID	0 / 0, 0 x ∞, $\sqrt{-1}$	NaN
DIVIDEBYZERO	Finite nonzero / 0	±∞
OVERFLOW	Ω + 1	±∞
UNDERFLOW	λ / 2	Subnormal
~~INEXACT~~	~~10 / 3~~	~~Rounded value~~

Floating-Point Numbers

- Floating-point numbers are a “leaky abstraction”
- The semantics of floating-point are intricate and unintuitive

Lack of Associativity

$$A + \alpha_1 + \alpha_2 + \ldots + \alpha_n$$

$$Assume \quad A \ll \alpha_i \quad and \quad A = \sum_{i=1}^{n} \alpha_i$$

$$((((A + \alpha_1) + \alpha_2) + \ldots) + \alpha_n) = A$$

$$A + (\ldots((\alpha_1 + \alpha_2) + (\alpha_3 + \alpha_4)) + \ldots + (\alpha_{n-1} + \alpha_n)\ldots) = 2A$$

Catastrophic Cancellation

a = 1.xxx xxx1

b = 1.xxx xxx0

a – b = 0.000 0001 uuu uuuu

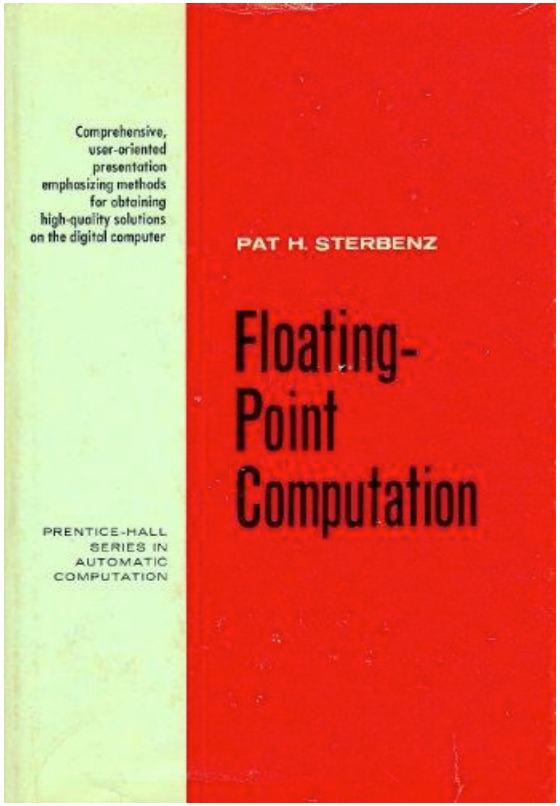
= 1.uuu uuuu x 10^{-7}

where each u is a random, unassigned digit

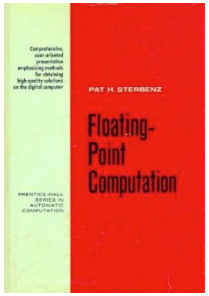
Adapted from slides by Gerald W. Recktenwald

Avoiding Floating-Point Exceptions

How hard is it to prevent floating-point exceptions via "carefully written programs"?

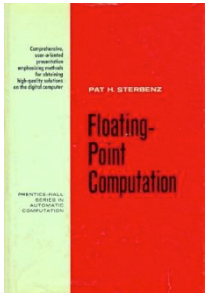


Averaging Two Numbers



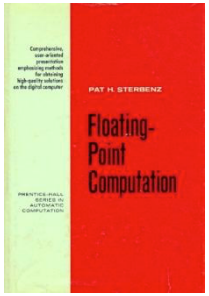
$$\frac{x+y}{2}$$

Averaging Two Numbers




```
double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}
```

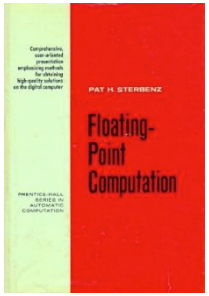
Averaging Two Numbers




```
double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}
```

$$\frac{x+y}{2}+\frac{\Omega+\Omega}{2}=\frac{2\Omega}{2}=\Omega$$

Averaging Two Numbers

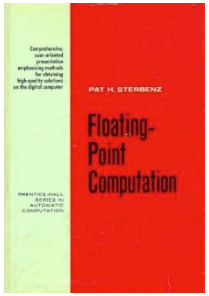



```
double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}
```

OVERFLOW

$$\frac{x+y}{2} + \frac{\Omega+\Omega}{2} = \frac{2\Omega}{2} = \Omega$$

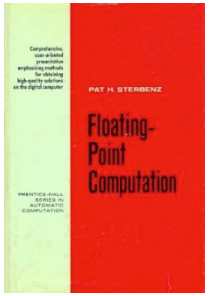
Averaging Two Numbers




```
double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}
```

$$\frac{x+y}{2} = \frac{\Omega}{2} + \frac{\Omega}{2} = \Omega$$

Averaging Two Numbers

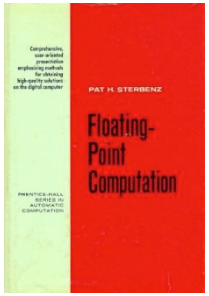



```
double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}
```

$$\frac{x+y}{2} = \frac{\lambda}{2} + \frac{\lambda}{2}$$

UNDERFLOW

Averaging Two Numbers



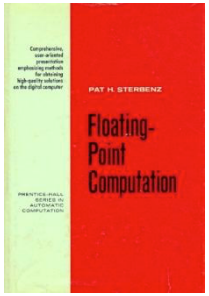

```
double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}
```

$\Rightarrow$ Overflows when x and y are large and have the same sign.

```
double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}
```

$\Rightarrow$ No overflow but obvious underflow.

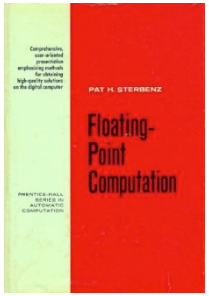
Averaging Two Numbers



$$\frac{x+y}{2}+\frac{x}{2}-\frac{x}{2}=\frac{x+x+y-x}{2}$$

$$=\frac{2x}{2}+\frac{y-x}{2}=x+\frac{y-x}{2}$$

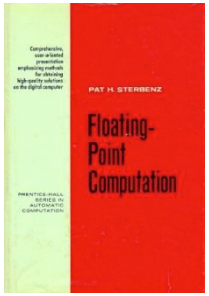
Averaging Two Numbers



Consider x = Ω, y = Ω

$$\Omega + \frac{\Omega - \Omega}{2} = \Omega + \frac{0}{2} = \Omega$$

Averaging Two Numbers

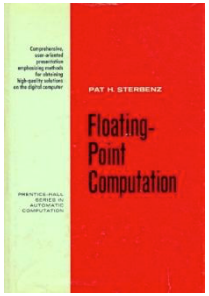


What about x = -Ω, y = Ω?

OVERFLOW

$$-\Omega + \frac{\Omega - (-\Omega)}{2} = \Omega + \frac{2\Omega}{2}$$

Averaging Two Numbers




```
double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}
```

$\Rightarrow$ Overflows when x and y are large and have the same sign.

```
double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}
```

$\Rightarrow$ No overflow but obvious underflow.

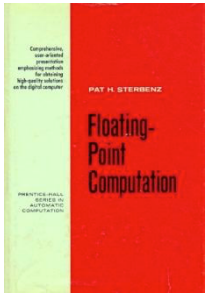
```
double av3 ( double x, double y ) {
    return ( x + ( y - x ) / 2.0 );
}
```

$\Rightarrow$ Overflows when x and y are large and differ in sign.

```
double av4 ( double x, double y ) {
    return ( y + ( x - y ) / 2.0 );
}
```

$\Rightarrow$ Overflows when x and y are large and differ in sign.

Sterbenz' Average

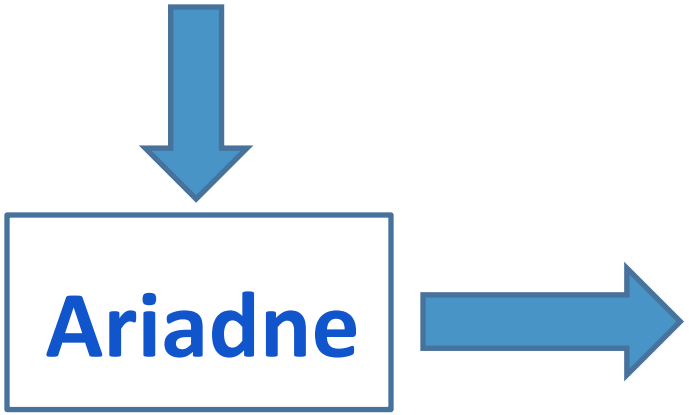



```
double average( double x, double y ) {
   int samesign;
   if ( x >= 0 ) {
      if ( y >= 0 )
         samesign = 1;
      else
         samesign = 0;
   } else {
      if ( y >= 0 )
         samesign = 0;
      else
         samesign = 1;
   }
   if ( samesign ) {
      if ( y >= x )
         return av3( x, y );
      else
         return av4( x, y );
   } else
         return av1( x, y );
   }
} //average
```

That's a lot of work and it only prevents overflow.

Tool Support

```
double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}
```

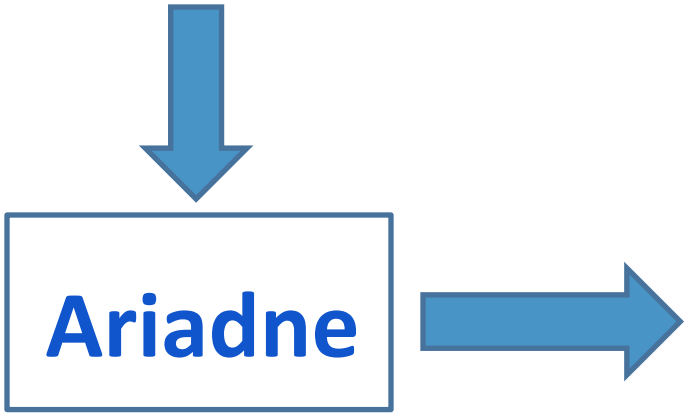


Overflow, line 2

$x = \Omega$, $y = \Omega$

Tool Support

```
double av3 ( double x, double y ) {
    return ( x + ( y - x ) / 2.0 );
}
```



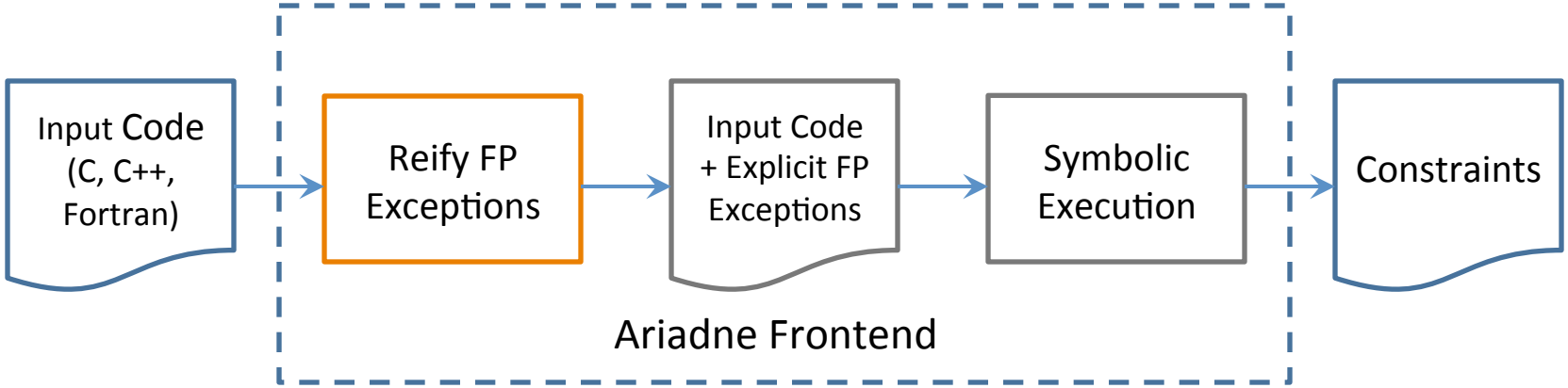
Overflow, line 2

$x = -\Omega$, $y = \Omega$

Ariadne's Design Principles

- Convert floating-point detection into a reachability problem
- Model floating-point arithmetic with real arithmetic

Ariadne Architecture



Rewriting Addition

```
double av1 ( double x, double y ) {
   return ( x + y ) / 2.0;
}
```

Rewriting Addition

```
double av1 ( double x, double y ) {
   double z = x + y;
   return z / 2.0;
}
```

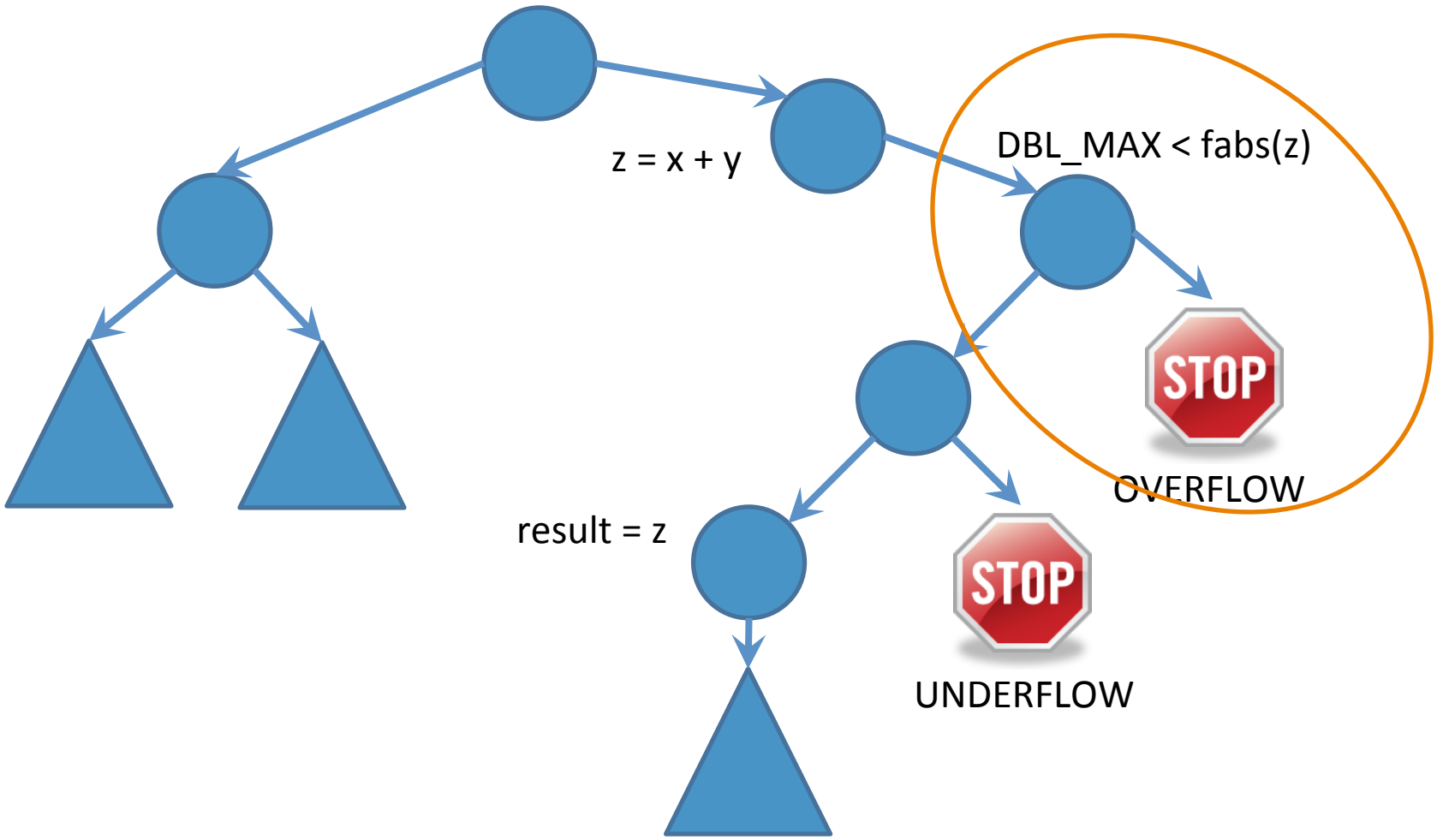
Rewriting Addition

```
double av1 ( double x, double y ) {
   double z = x + y;
   double abs1 = fabs( z );
   if ( DBL_MAX < abs1 ) {
      perror( "OVERFLOW!\n" );
      exit( 1 );
   }




   return z / 2.0;
}
```

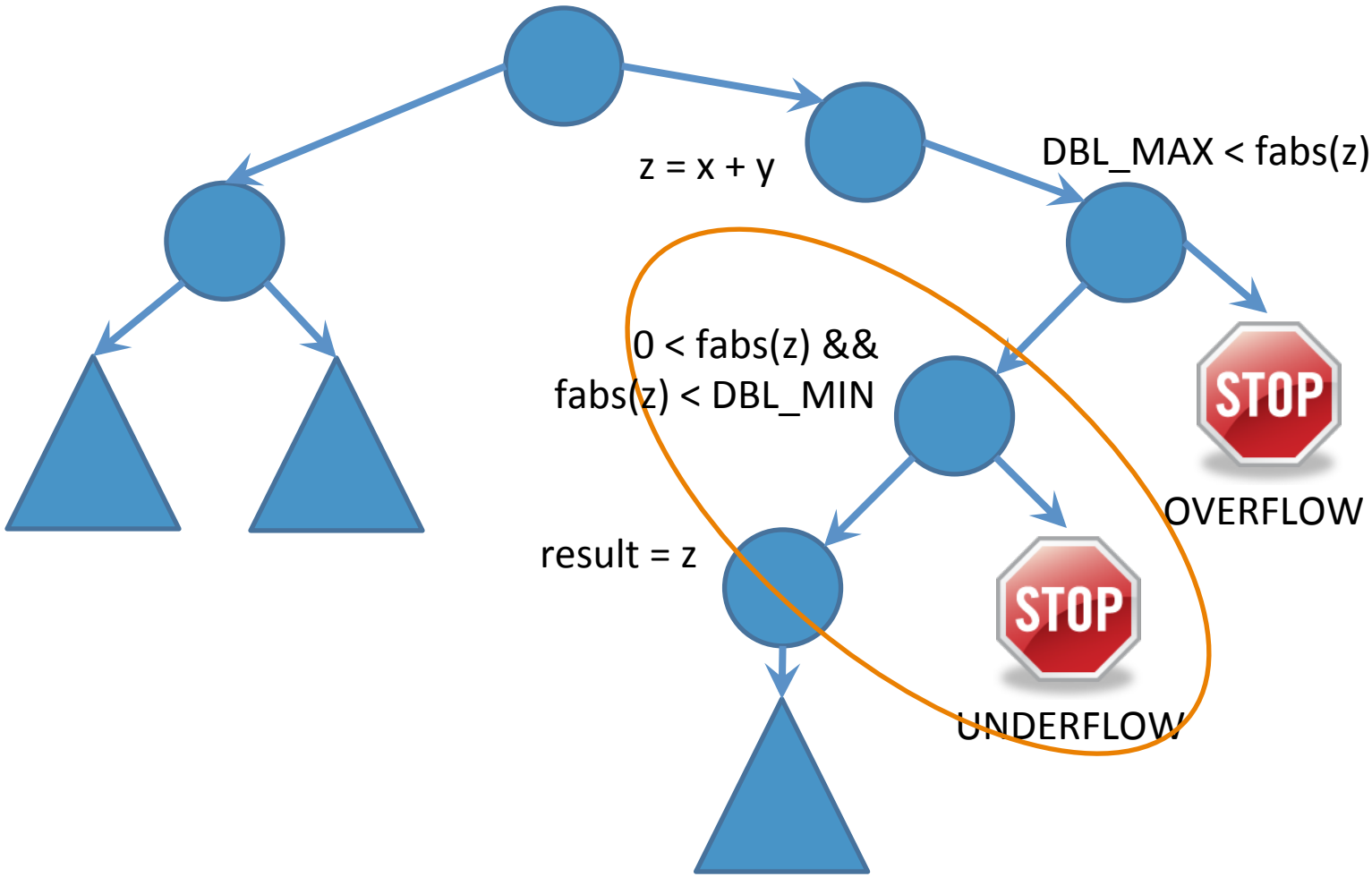
Rewritten Program's CFG



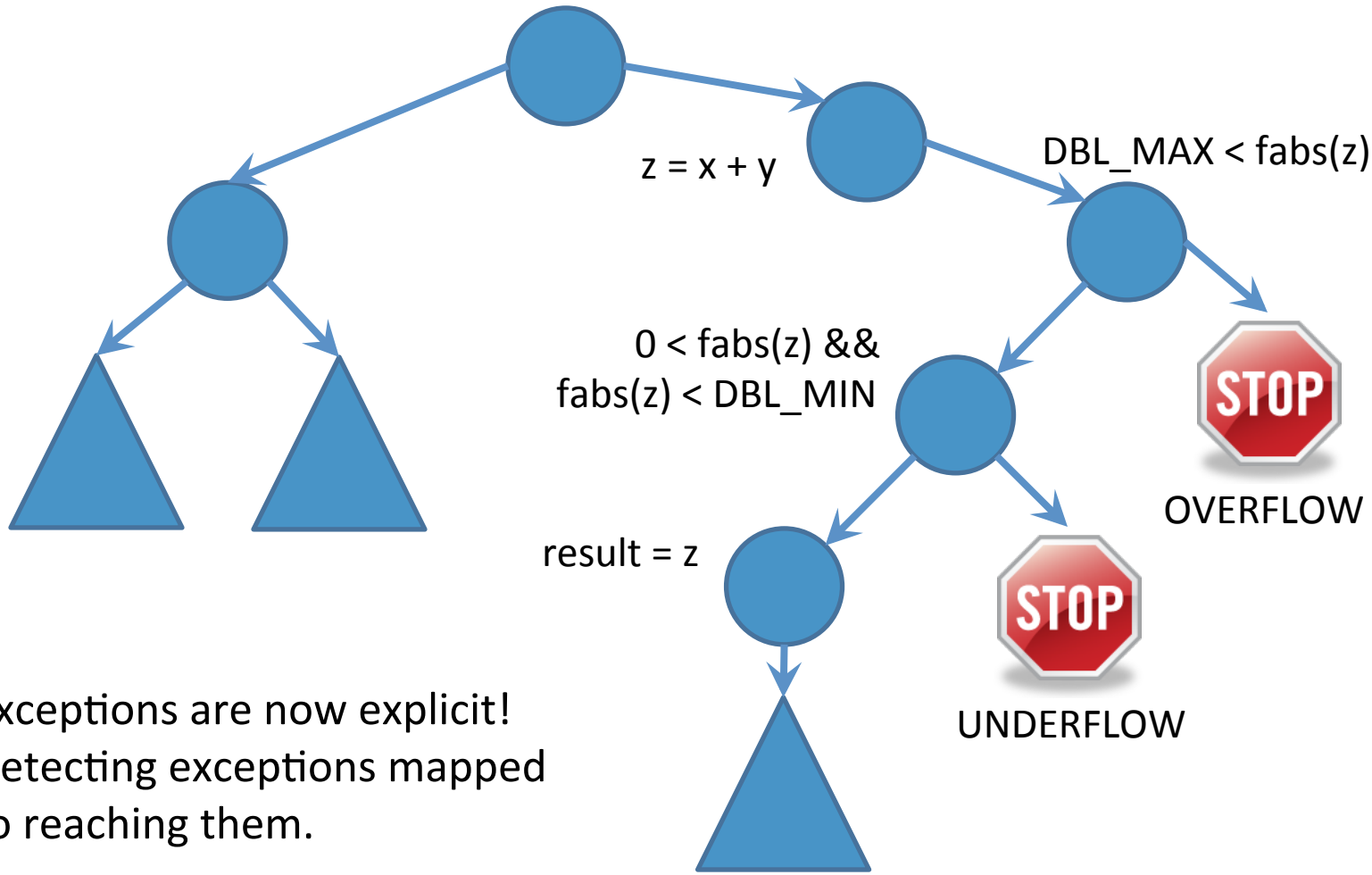
Rewriting Addition

```
double av1 ( double x, double y ) {
   double z = x + y;
   double abs1 = fabs( z );
   if (DBL_MAX < abs1 ) {
      perror( "OVERFLOW!\n" );
      exit( 1 );
   }
   if (0 < abs1 && abs1 < DBL_MIN) {
      perror( "UNDERFLOW!\n" );
      exit( 1 );
   }
   return z / 2.0;
}
```

Rewritten Program's CFG



Reifying Floating-Point Exceptions



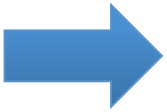
Exceptions are now explicit!
Detecting exceptions mapped to reaching them.

Elementary Functions

- `log, sqrt, cos, sin, pow, …`
- "External functions"
  - Some handled in hardware

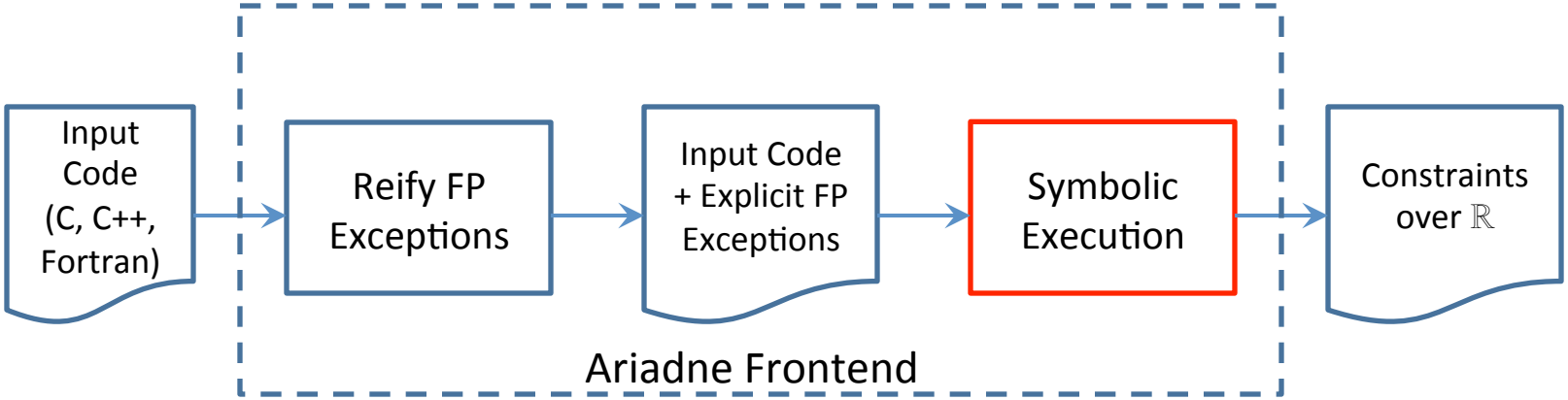
Elementary Function Handling

```
double y = log(x);
...
if ( x > 1 ) {
   ...
}
...
if ( y > 0 ) {
   ...
}
```

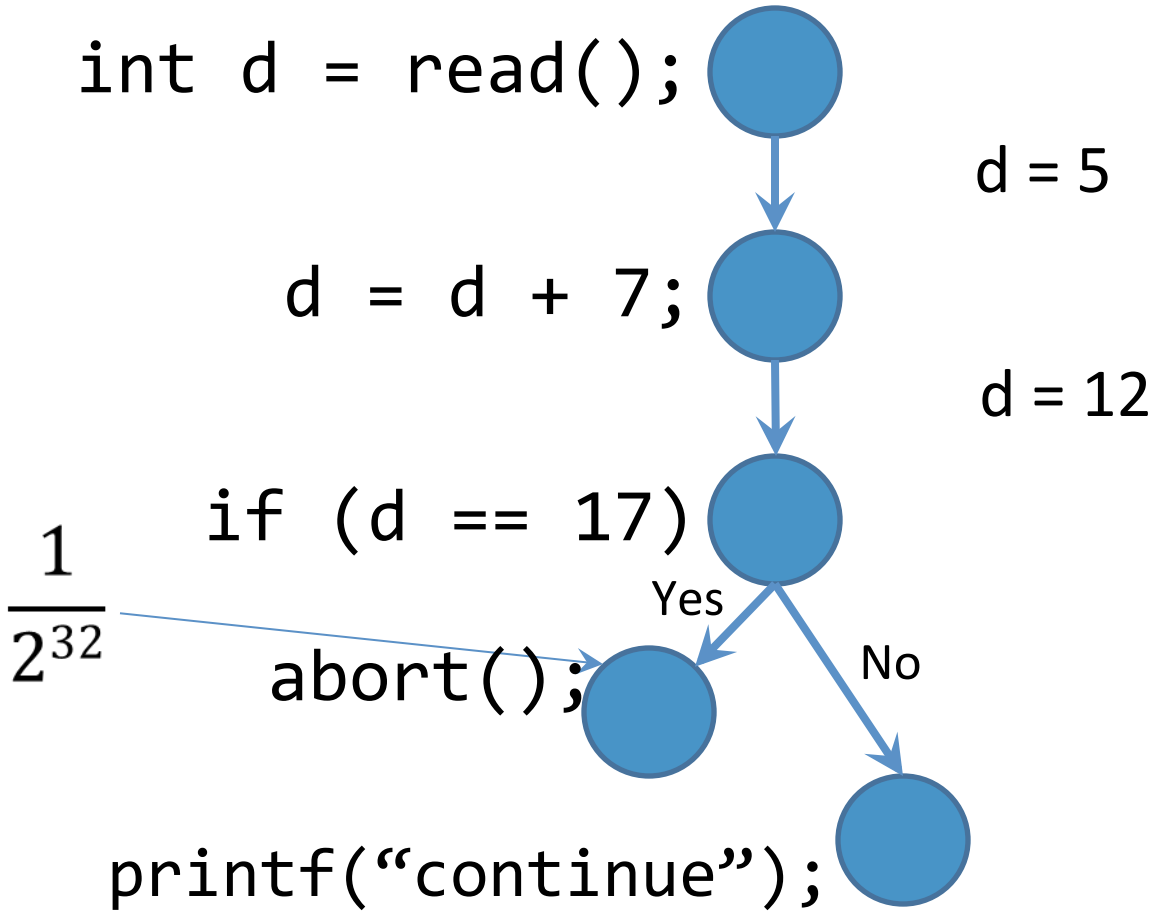


```
...
if ( x <= 0 ) {
  perror( "INVALID!\n" );
  exit( 1 );
}
double y = d;
...
```

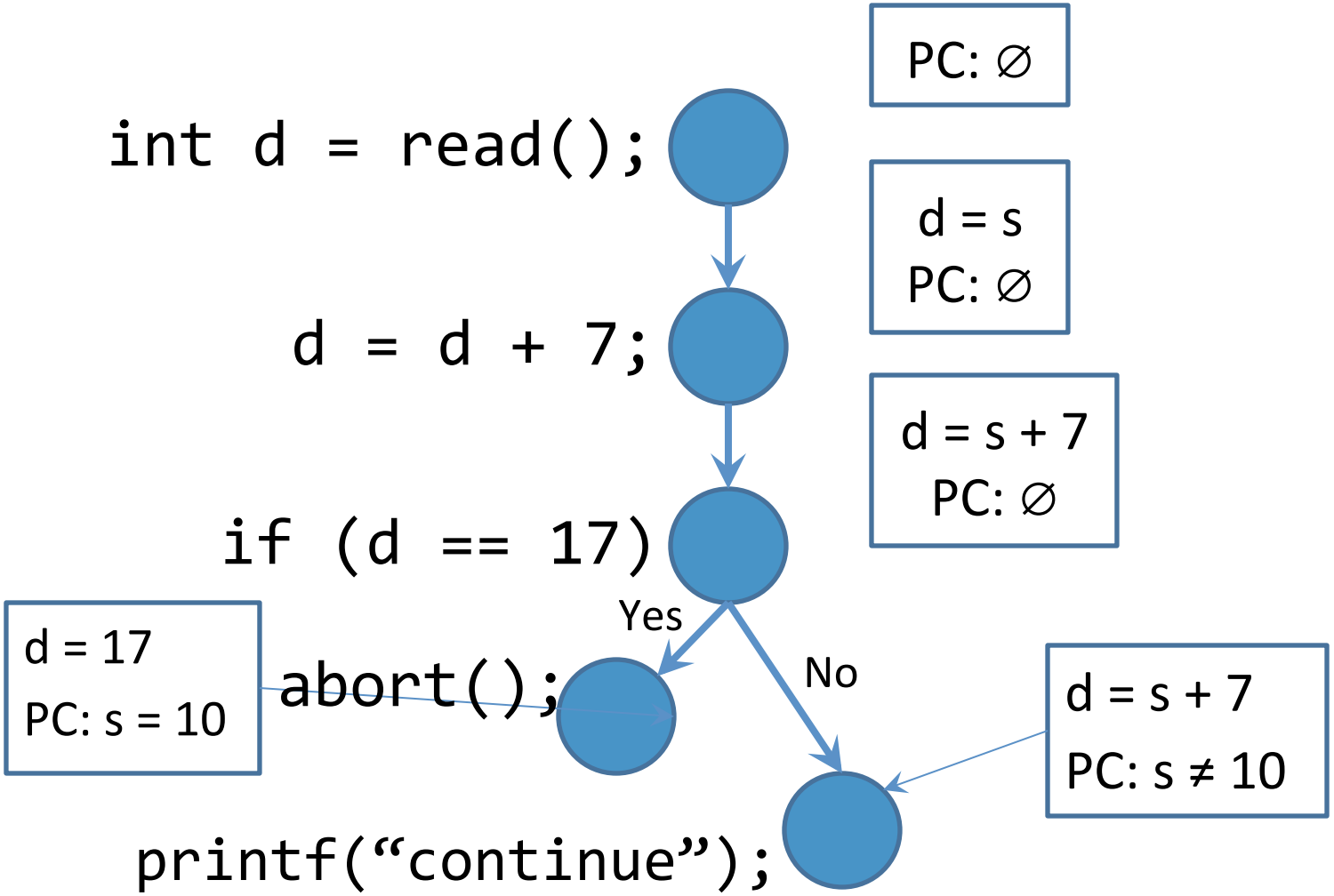
Generating Path Constraints



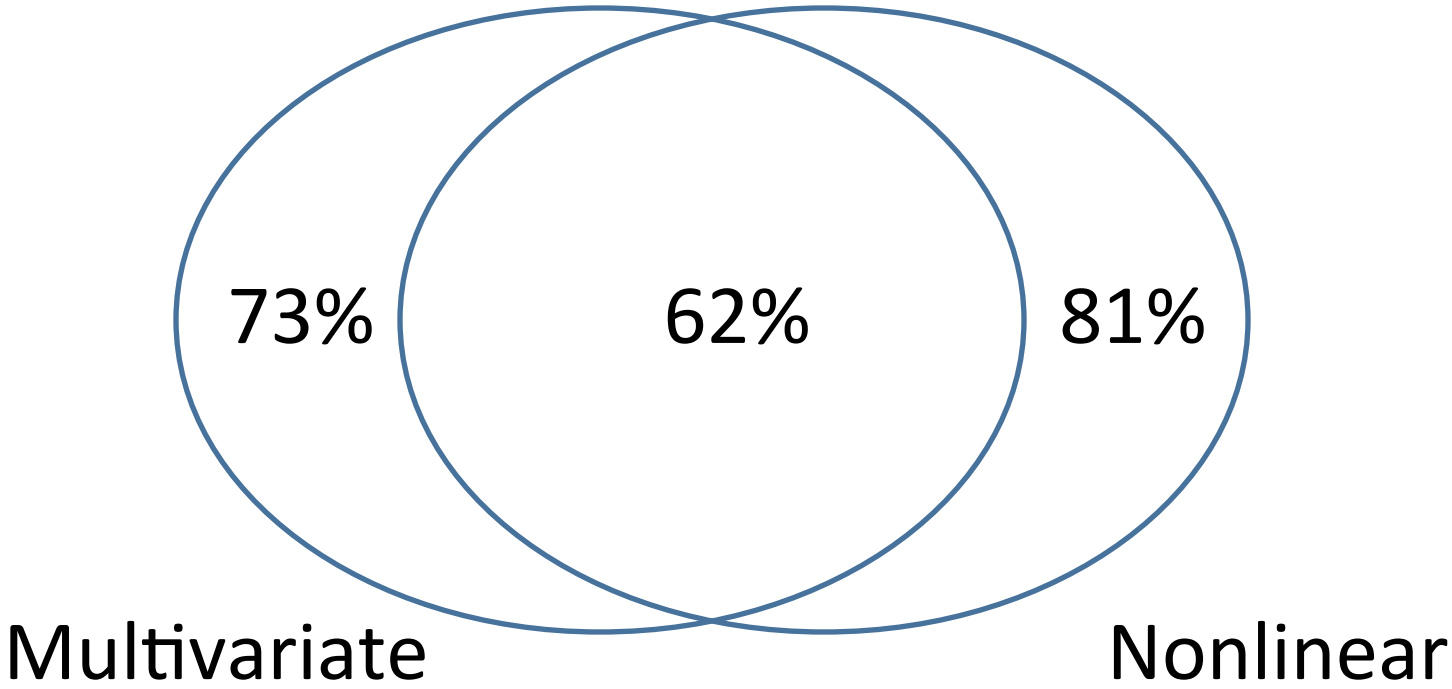
“Concrete” (Normal) Execution



Symbolic Execution -- using KLEE



Ariadne Path Constraints (PCs)



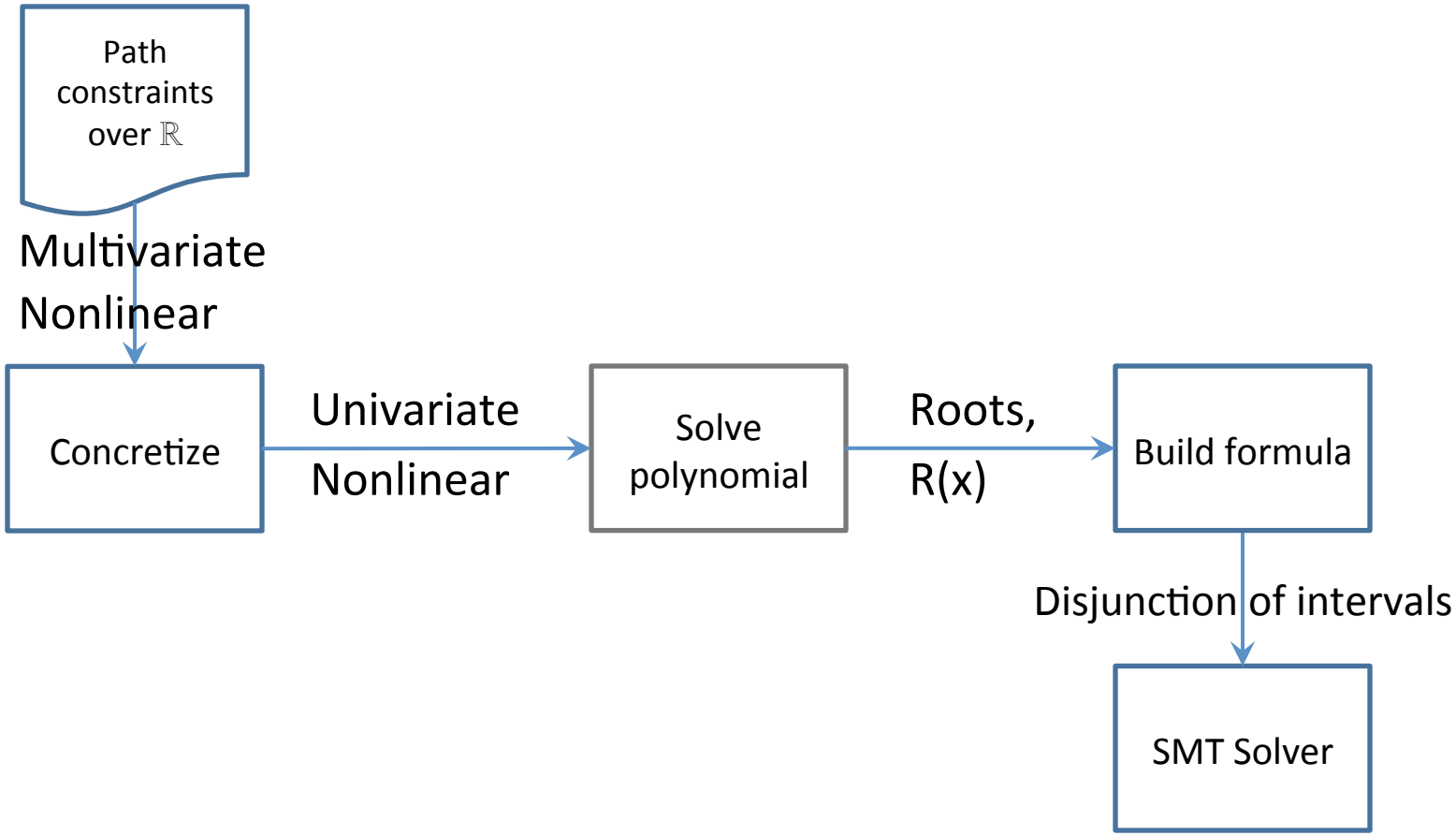
Satisfiability Modulo Theories

- Boolean satisfiability (SAT)
  - $(x \vee y) \wedge z$
- Satisfiability Modulo Theories (SMT)
  - SMT generalizes Boolean SAT to functions and predicates
  - Theories include real numbers, arrays, bit vectors, and uninterpreted functions
    - $5x + 7y - z \leq 17$
    - select(store(store(a,1,−1),2,−2),1)

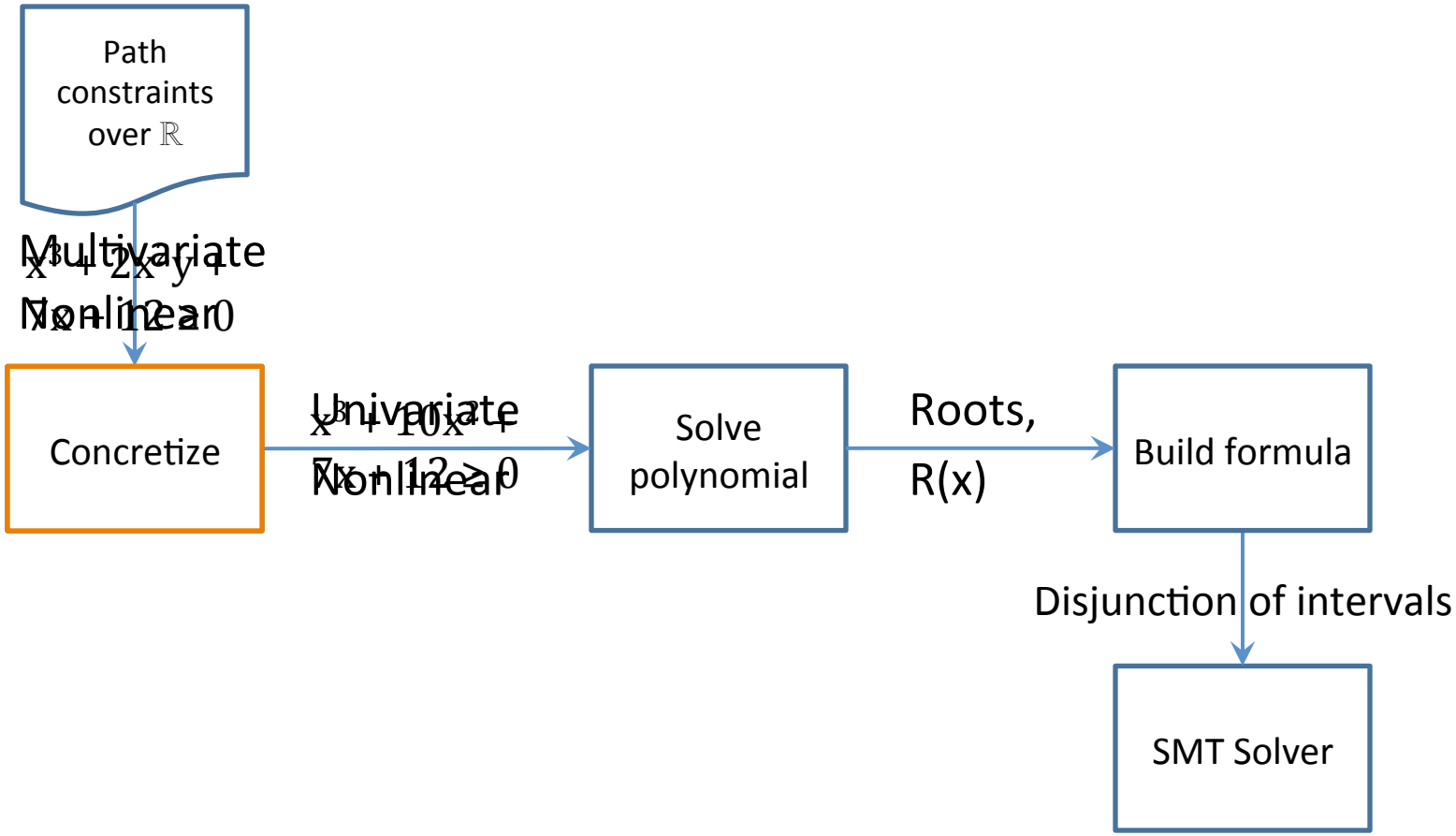
SMT solvers

- Z3 by Microsoft
  - SMT solver with nlsat (non-linear arithmetic) support
- Z3-Ariadne
  - custom multi-variate non-linear constraint solver

Z3-Ariadne pipeline

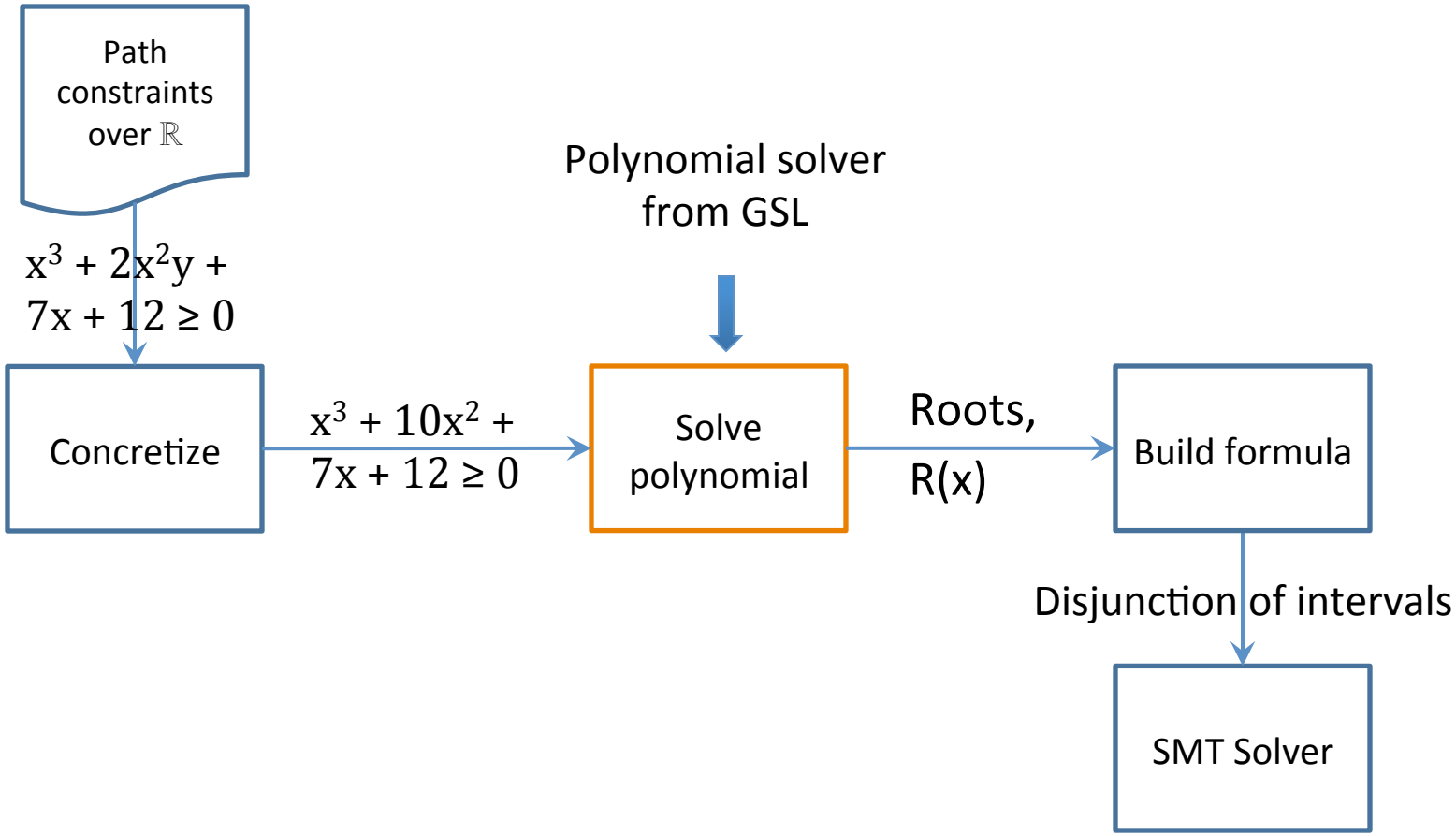


Concretizing Multi-variate PCs



Authors found that multiple guesses don't improve perf. by much

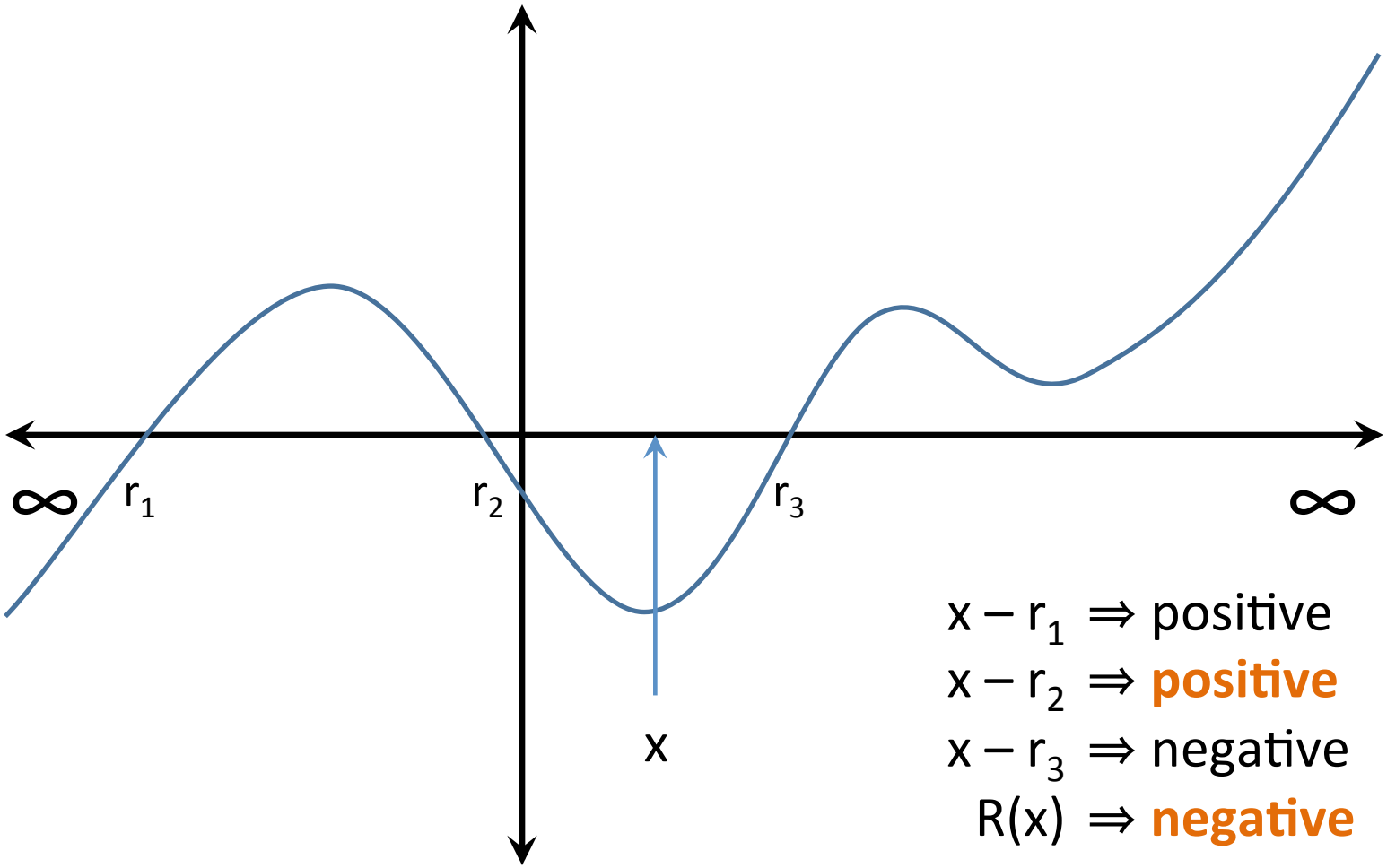
Nonlinearity remains...



Nonlinear Univariate Constraints

- Normalize nonlinear univariate constraints
  - $R(x) \bowtie 0$
- $R(x) = (x - r_1)(x - r_2)\ldots(x - r_n)$
  - Real roots determine the sign of $R(x)$

Real Roots Determine Sign



$x - r_1 \Rightarrow$ positive

$x - r_2 \Rightarrow$ **positive**

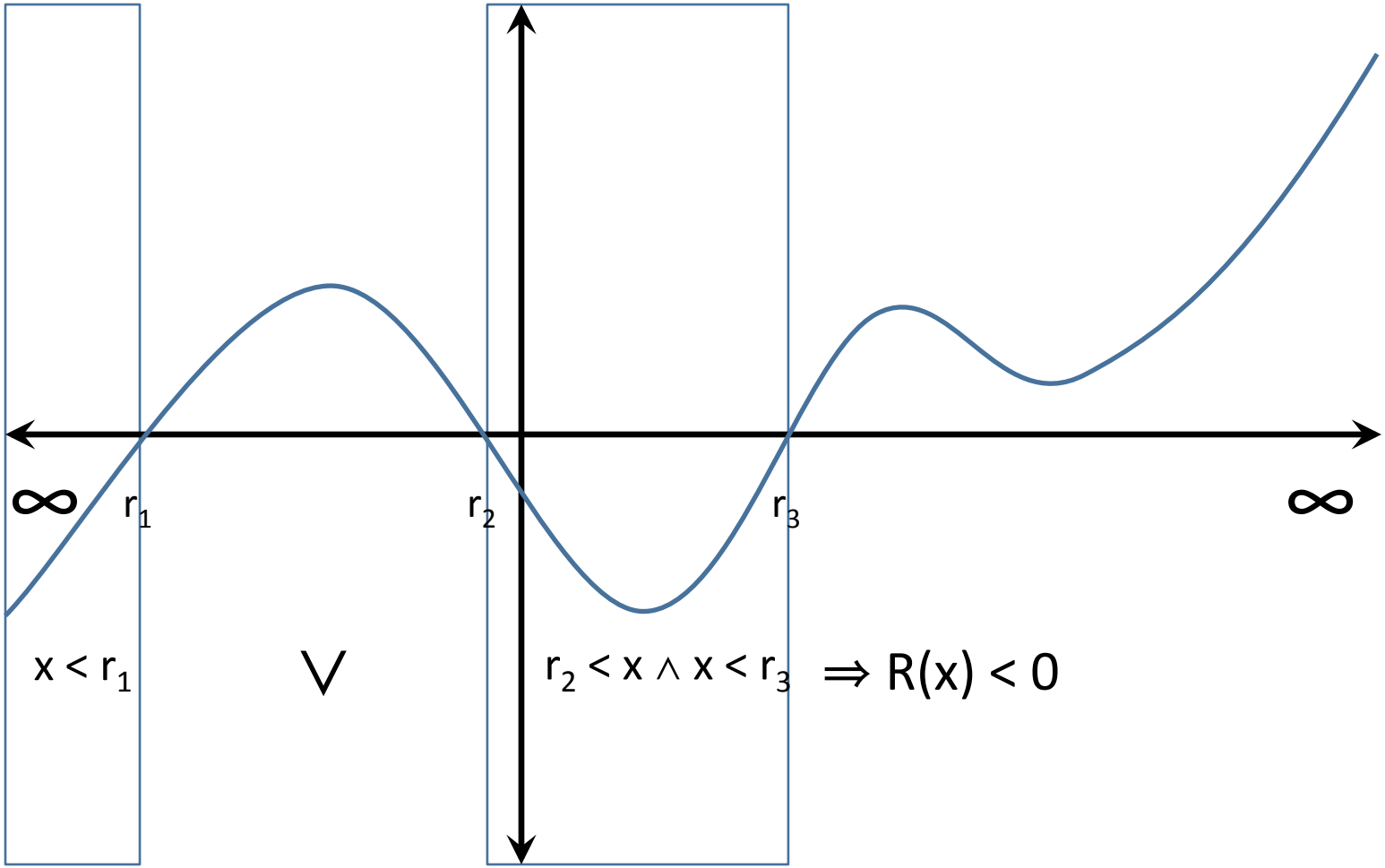
$x - r_3 \Rightarrow$ negative

$R(x) \Rightarrow$ **negative**

Nonlinear Univariate Constraints

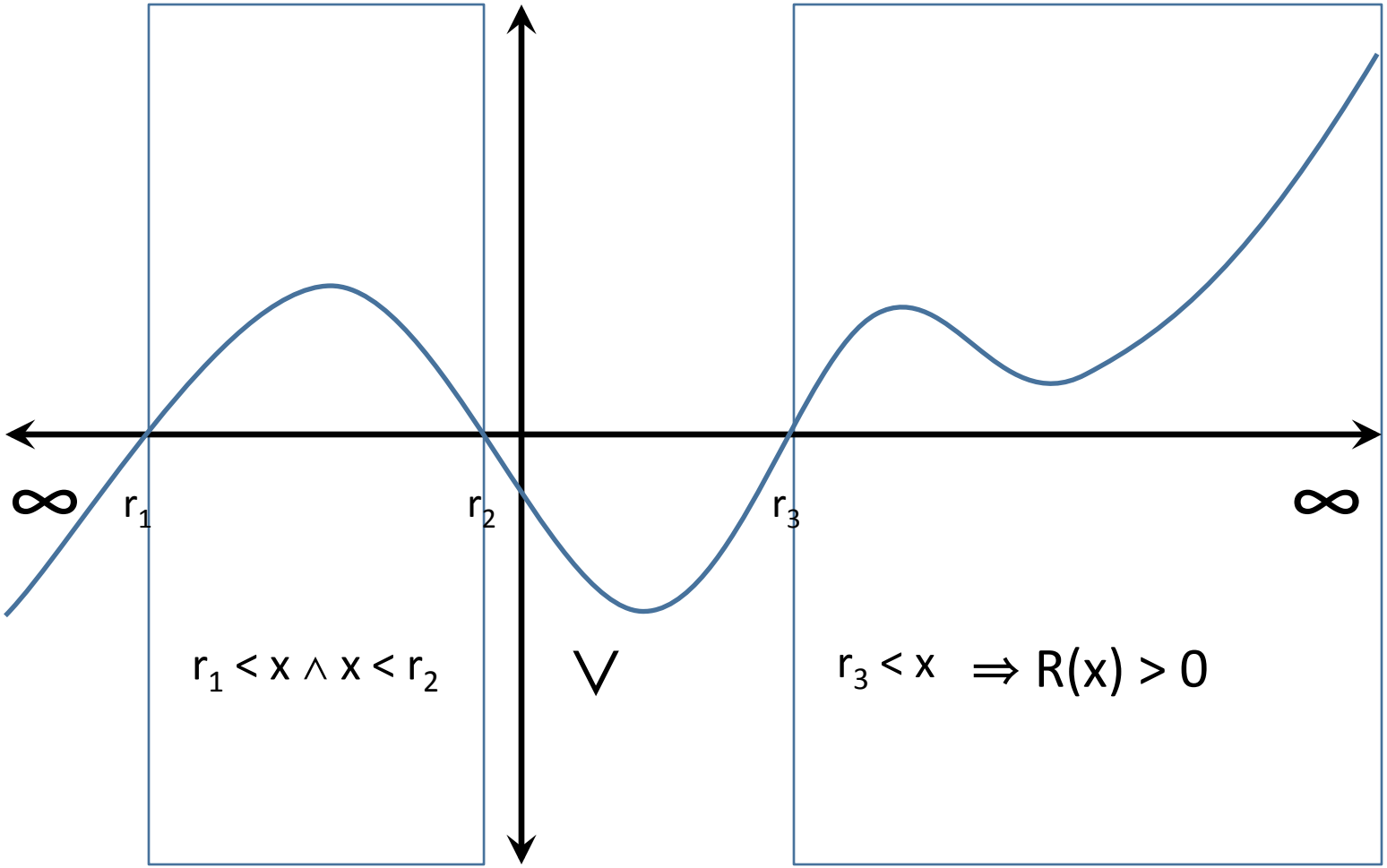
- In our example, $R(x) = x^3 + 10x^2 + 7x + 12$
  - and we want $R(x)$ to be >= 0

Disjunction for R(x) < 0

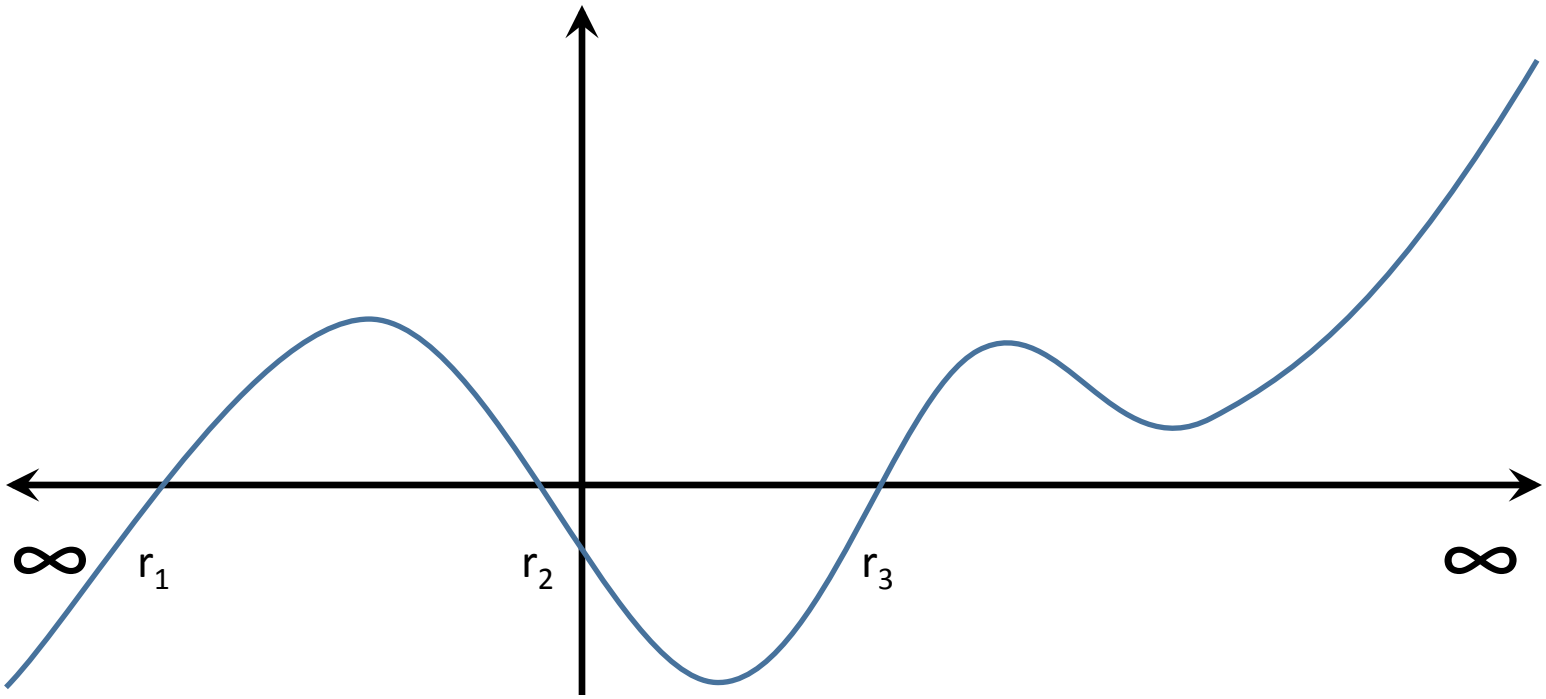


$x < r_1 \quad \vee \quad r_2 < x \wedge x < r_3 \Rightarrow R(x) < 0$

Disjunction for R(x) > 0

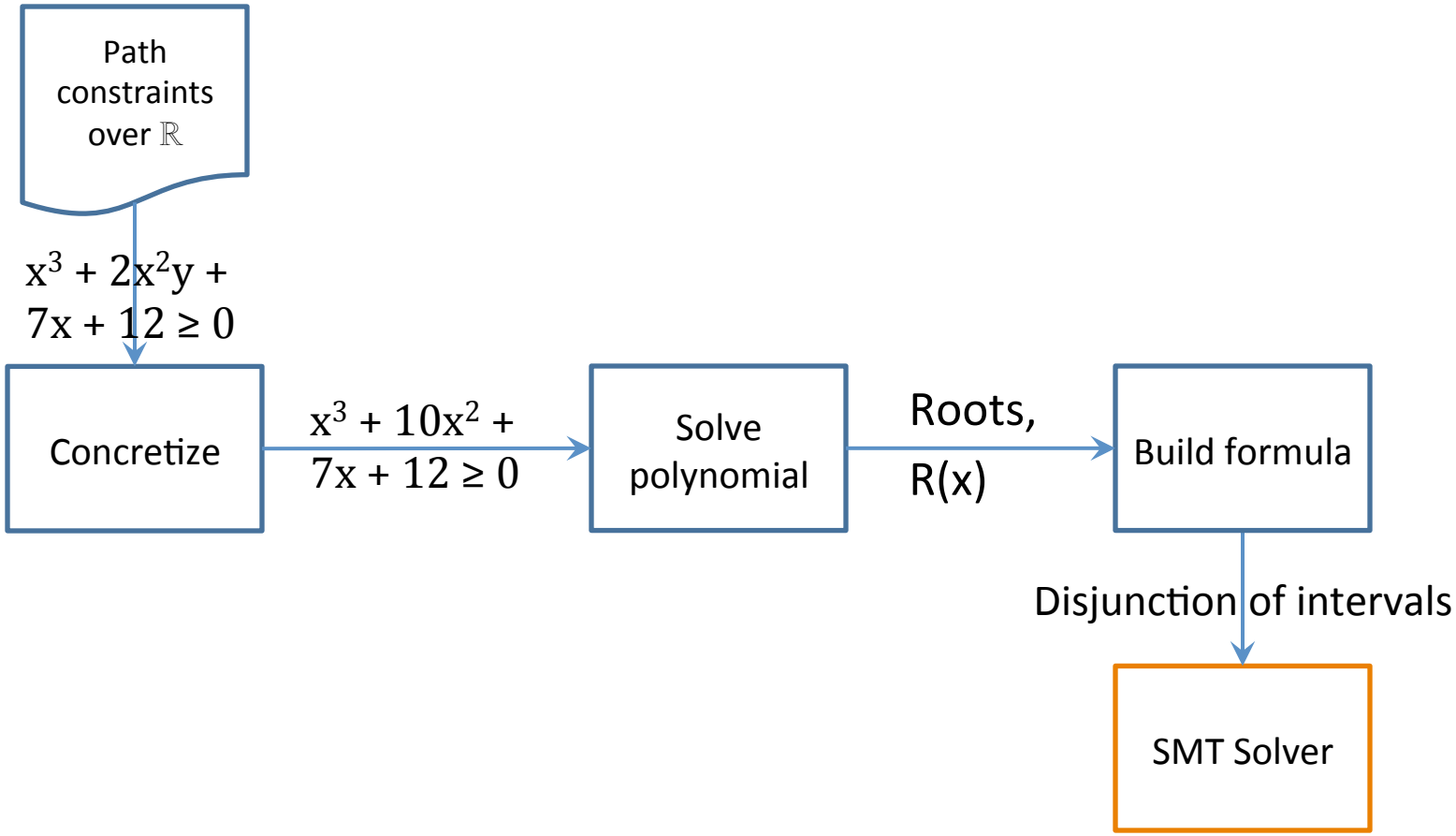


Disjunction for R(x) = 0

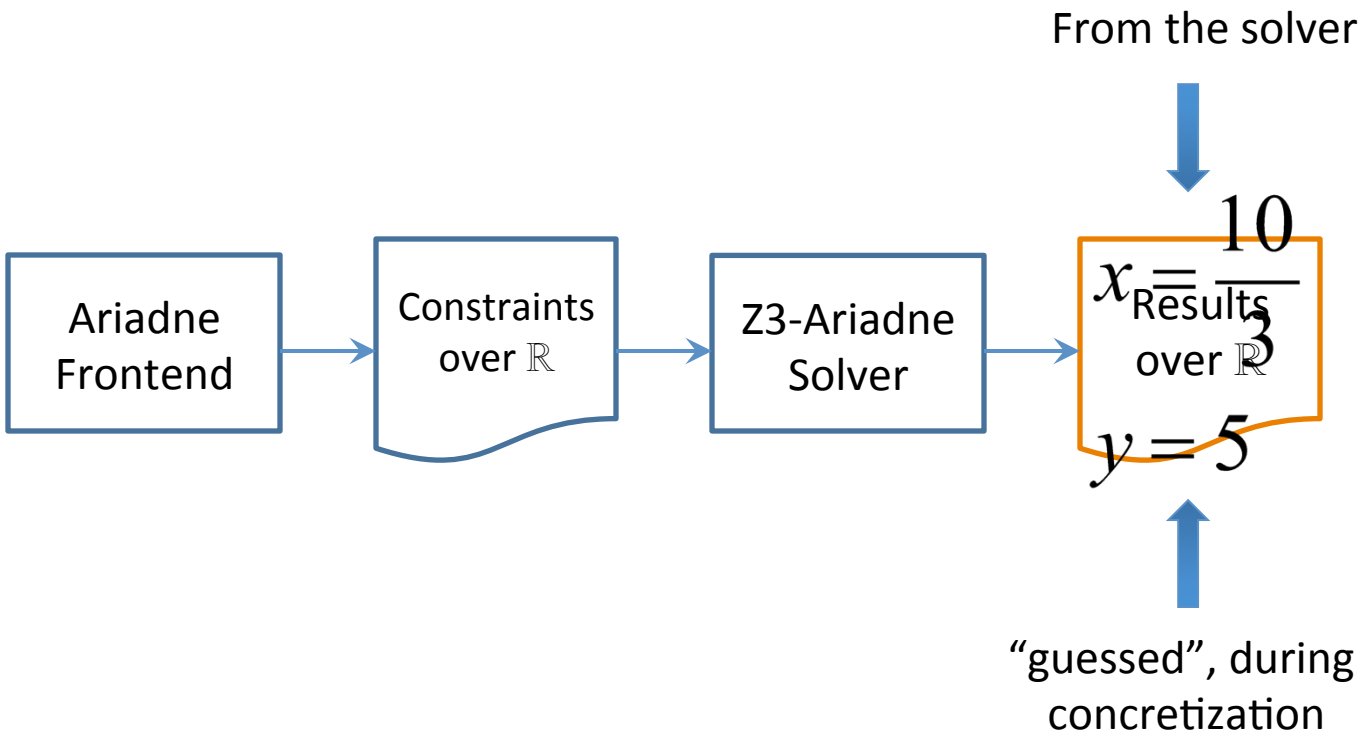


$$x = r_1 \quad \vee \quad x = r_1 \quad \vee \quad x = r_3 \quad \Rightarrow R(x) = 0$$

Feed intervals to SMT solver



Ariadne Architecture (revisited)



Result Handling

- The SMT solver returns a solution over $\mathbb{R}$
  - NOT floating-point number

$$x = \frac{10}{3}$$

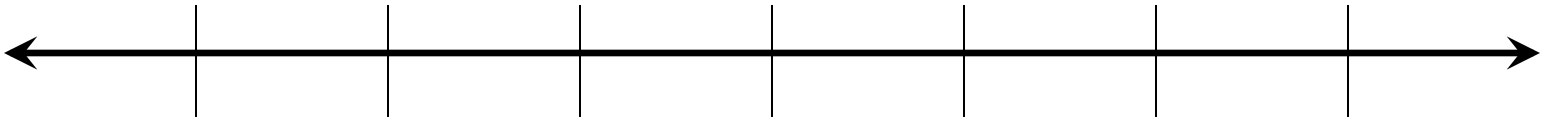
findFloat

$$x \in [l, h] \lor x \in [l', h'] \lor \quad \ldots$$

findFloat

$x \notin [l, h] \vee x \in [l', h'] \vee \quad \ldots$

Initially, $x = a \in [l, h]$
Eliminate "useless" intervals

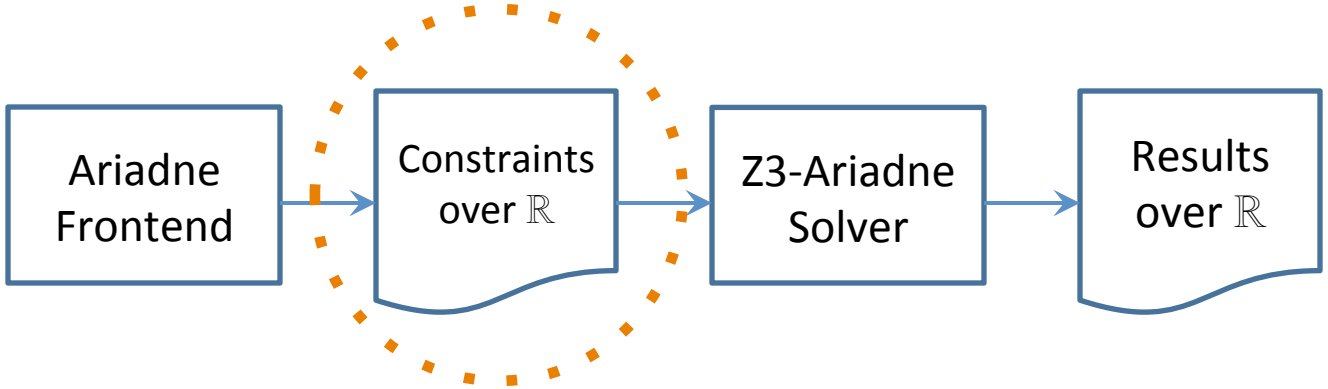


a_f $\quad l \quad a_f >= l$? $\quad a \quad next(a_f) \quad h \quad next(a_f)$
$<= h$?

Trapped in Real Arithmetic

At this point we have inputs that are

- normal floating-point numbers
- guaranteed to trigger an exception
- *when symbolically executed over arbitrary precision rationals*



Problems with Real Arithmetic

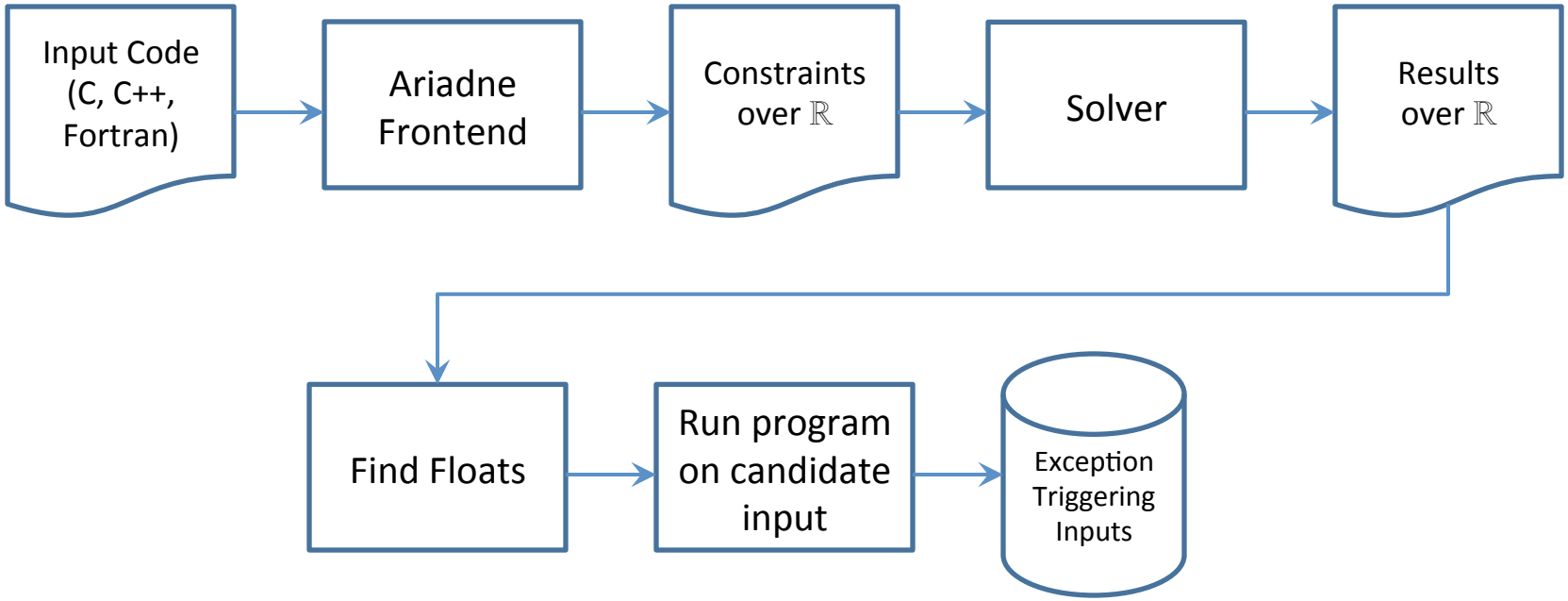
```
r = 10^54 + 42 – 10^54
if( r == 42 )
    printf(“Path under ℝ.\n”);
else
    printf(“Path under FPA.\n”);
```

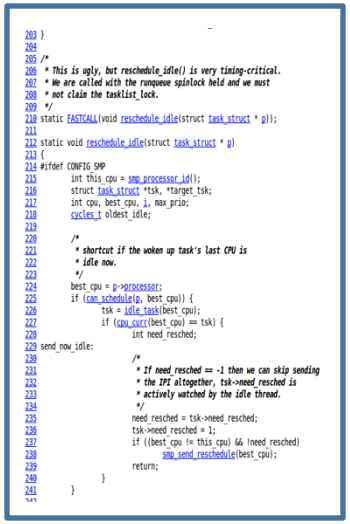
Escape from Real Arithmetic

Testing the trigger

- Ariadne concretely executes a candidate exception-trigger input over floating-point to confirm it triggers the specified exception.

Complete Ariadne Architecture





```
203 }
204
205 /*
206  * This is ugly, but reschedule_idle() is very timing-critical.
207  * We are called with the runqueue spinlock held and we must
208  * not claim the tasklist_lock.
209  */
210 static FASTCALL(void reschedule_idle(struct task_struct * p));
211
212 static void reschedule_idle(struct task_struct * p)
213 {
214 #ifdef CONFIG_SMP
215         int this_cpu = smp_processor_id();
216         struct task_struct *tsk, *target_tsk;
217         int cpu, best_cpu, i, max_prio;
218         cycles_t oldest_idle;
219
220         /*
221          * shortcut if the woken up task's last CPU is
222          * idle now.
223          */
224         best_cpu = p->processor;
225         if (can_schedule(p, best_cpu)) {
226                 tsk = idle_task(best_cpu);
227                 if (cpu_curr(best_cpu) == tsk) {
228                         int need_resched;
229 send_now_idle:
230                         /*
231                          * If need_resched == -1 then we can skip sending
232                          * the IPI altogether, tsk->need_resched is
233                          * actively watched by the idle thread.
234                          */
235                         need_resched = tsk->need_resched;
236                         tsk->need_resched = 1;
237                         if ((best_cpu != this_cpu) && !need_resched)
238                                 smp_send_reschedule(best_cpu);
239                         return;
240                 }
241         }
```

Ariadne

Overflow, line 2

$x = -\Omega$, $y = \Omega$

Underflow, line 11

$x = 0$, $y = 100$

DivideByZero, line 100

$x = -3$, $y = 12$

...

Ariadne is sound

But not complete, i.e. it cannot guarantee that a program is exception-free, because it:

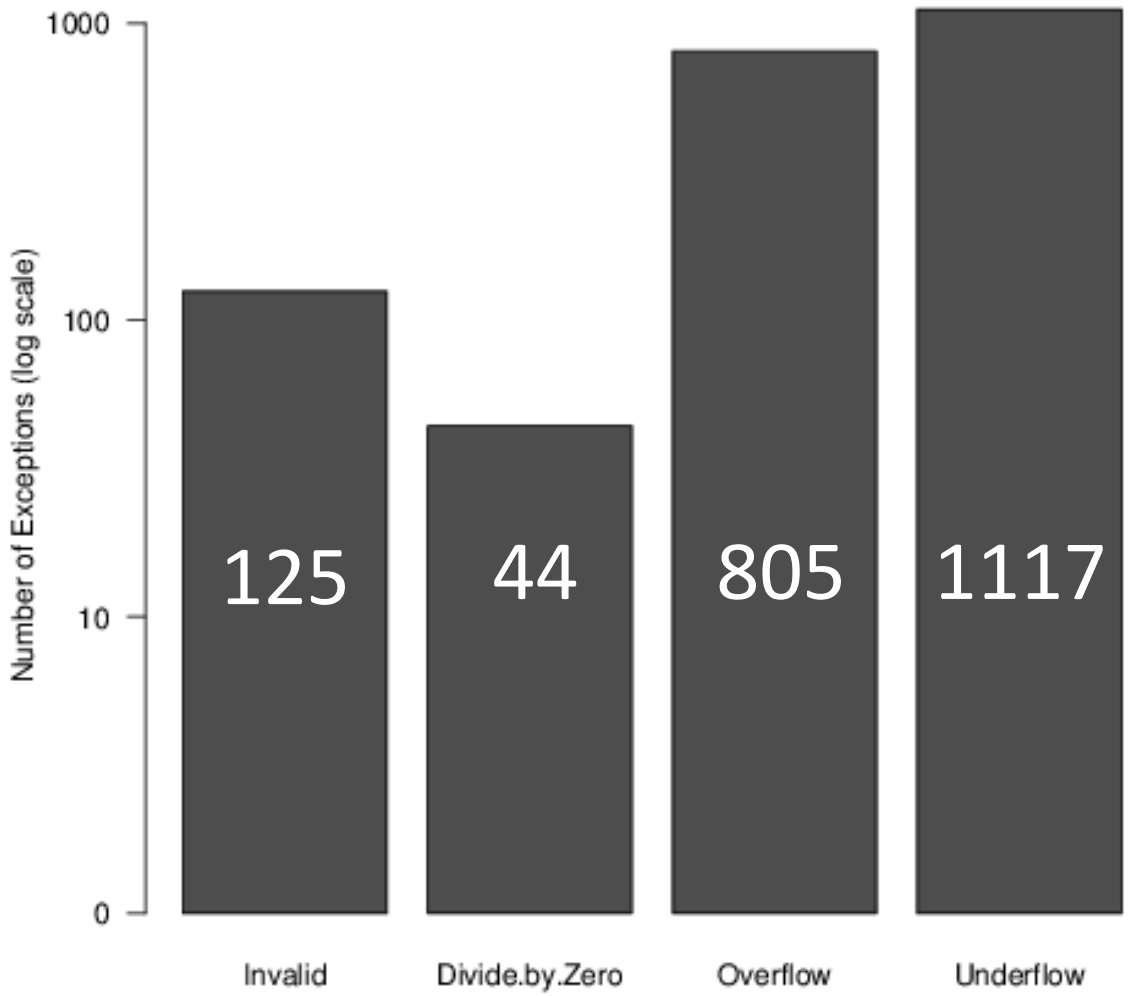
- approximates FP semantics with constraints over the reals, which notably ignore rounding (i.e. Inexact);
- concretizes multivariate constraints into univariate ones;
- rewrites loop conditions to control the number of iterations (Symbolic Execution path explosion prob.)

Evaluation

- GNU Scientific Library (GSL)
  - Widely used and well-tested library

- 467 special functions with scalar input/output

2091 Exception-Triggering Inputs



A Divide By Zero

gsl/specfunc/legendre_con.c

```
867: gsl_sf_conicalP_1_e(const double lambda,
       const double x, gsl_sf_result *result)
```

When `lambda = 20` and `x = 1`, after 113 lines and 8 control points, we reach

```
980:  const double sqrt_xm1 = sqrt(x – 1.0);
981:  const double sqrt_xp1 = sqrt(x + 1.0);
982:  const double sh = sqrt_xm1 * sqrt_xp1;
…
989:  int stat_V =
        conicalP_1_V( th, x/sh, lambda, 1.0, &V0, &V1);
```

Unavoidable Xflows

$$\Omega + \Omega = 2\Omega$$

$$\frac{\lambda}{2} = subnormal$$

Interesting Xflows

XFLOW

$x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_n = ?$

Over floating-point

$x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_n = z \quad \wedge |z| \in [\lambda, \Omega]$

Over the reals

$\oplus \in \{+,-,*,/\}$ is an arithmetic operation

Interesting Xflows

	Overflows	Underflows
Avoidable?	98	180
Total	155	217
Ratio	0.63	0.83

Questions?