Program verification

Data Structures in Abstract Interpretation

Laure Gonnord and David Monniaux

University of Lyon / LIP

November 2016

Introduction

Plan

Floating-point numbers

C types **float** and **double** (sometimes **long double**).
Most often follow the IEEE-754 standard

Plus:

- IEEE-754 floating-point operations $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\sqrt{}$ are **deterministic**
  The standard defines the result **exactly**
- The **rounding error** $|(x \oplus y) - (x + y)|$ may be **bounded**

Definition of floating-point numbers

Nonzero: $\pm mantissa.2^{exponent}$

- fixed number of bits in mantissa
- first bit of mantissa is always 1, not stored
- IEEE-754 single precision (often **float**): 23 bit mantissa, 8 bits exponent (biased, includes sign), 1 sign bit = 32 bits
- IEEE-754 double precision (often **double**) : 52 bit mantissa, 11 bit exponent (biased), 1 sign bit = 64 bits
- In 32-bit x86, **long double**: 64 bit mantissa, 15 bit exponent, 1 sign bit = 80 bits.

In reality, more complicated due to denormal numbers, ± 0, $\pm\infty$, and *not-a-number* (NaN, warns about errors).

Floats, difficulties

Not so deterministic...

- Float operations are **not associative**: in general $(x \oplus y) \ominus y \neq x$.
  For instance, $x = 1$, $y = 10^{10}$, $x \oplus y = 10^{10}$, $(x \oplus y) \ominus y = 0$.
- Certain compilers, by default, allow rewriting by associativity for optimizations.
  In gcc, such optimizations need explicit option `-ffast-math`.
- On some architectures *fma* (*fused multiply-add*) combines $+$ and $\times$.
  $a \times b + c$ may be, depending on context, be compiled into $fma(a, b, c)$ ou $a \otimes b \oplus c$
- 32-bit x86 under Windows and Linux, often used with 80-bit registers and 32- or 64-bit variables, the result depends on register scheduling.

Fun example

```
void do_nothing(double *x) { }
int main(void) {
  double x = 0x1p-1022, y = 0x1p100, z;
  do_nothing(&y);
  z = x / y;
  if (z != 0) {
    do_nothing(&z);
    assert(z != 0);
  }
}
```

0x1p−1022 is C99 notation for 2^{-1022}, 0x1p100 is 2^{100}.
assert(cond) issues a runtime error if cond is false.

We could believe **assert**(z != 0) never issues a runtime error...

Subtle differences

With gcc 4.0 on 32-bit Linux:

- Without optimization, x / y is a nonzero **long double** (80 bits), converted into zero **double**, stored into z, reloaded from memory for z != 0. **if** is not taken.
- If in a single source code with optimization, gcc seems that do_nothing() does nothing, realizes that z est nul, thus **if** is not taken and main() does nothing. main() is compiled as an empty function.

Nonintuitive

do_nothing() and main() in separate source code and optimization is used
gcc does not know that do_nothing() does nothing to the variable passed to it
Test z != 0 is over nonzero **long double** in register.
After do_nothing(), z is reloaded from memory as zero.
Assertion failed and runtime error.

Weakest precondition approaches may consider both z != 0 expressions to be equivalent, thus the assertion never fails!
Semantic modeling problem.

Other example on Intel

```
static inline double f(double x) {
  return x/1E308;
}
double square(double x) {
  double y = x*x;
  return y;
}
int main(void) {
  printf("%g\n", f(square(1E308)));
}
```

1E308 is approximately 10^{308}.

Depending on optimization, prints $+\infty$ or an approximation of 10^{308}.

Abstract interpretation

- For **intervals**, compute $[l, h] \oplus [l', h']$ with $l \oplus l'$ rounded below and $h \oplus h'$ rounded above.
- On relational domains, often designed for $\mathbb{R}$ or $\mathbb{Q}$, use $x \oplus y = x + y + \varepsilon$ and ε bounded according to $|x + y|$.

Verimag

Lessons learned

- Dangerous to reason on floats as though reals.
- Dangerous to assume compilers respect the C standard.
- Easier to define float semantics on assembly code than on C.
- Debugging float computations may be difficult. To debug one often disables optimizations, which may change results.
- Changing compilers, processors, operating systems may change results even if everybody "IEEE-754 compliant".
- *Weakest precondition* methods are fragile w.r.t fuzzy semantics.
- Abstract interpretation is more resilient.

More details in [Monniaux, TOPLAS 2008]

Verimag

Plan

The problem

Discrete-time digital filters implemented in software (general-purpose CPUs, DSPs) or in hardware.
A lot of the filtering linear: what we'll deal with
Implemented in fixed- or **floating-point**.
Need to provide sound assurance of absence of **runtime errors** in the program, **including overflows**.
Thus **need to bound all filter outputs** (and all intermediate values).

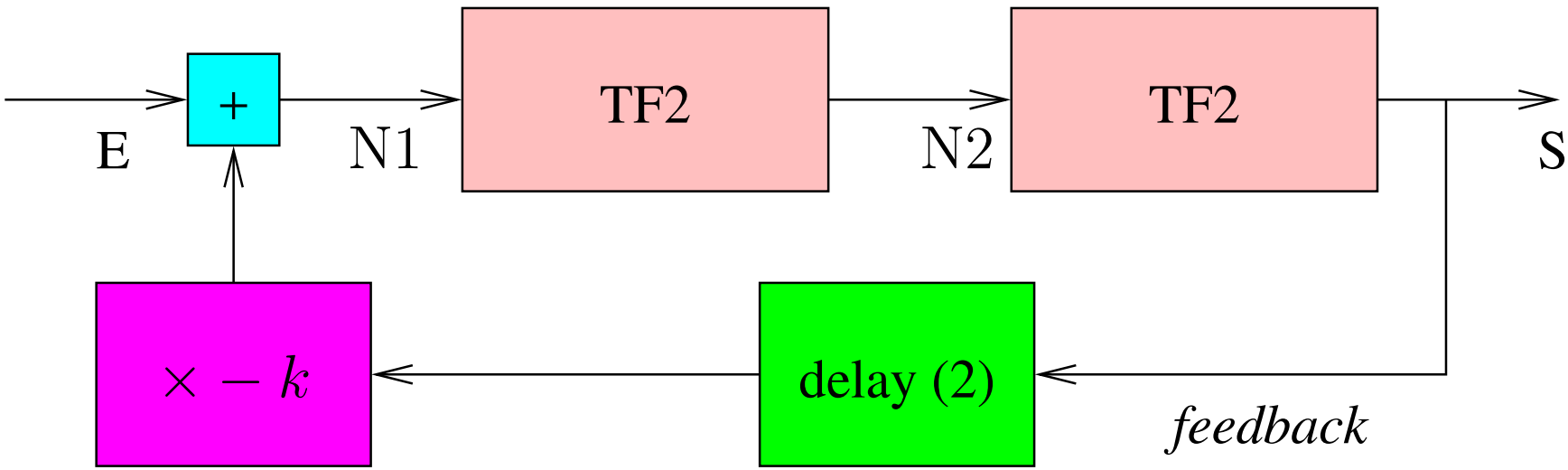
Discrete-time causal linear filters

Inputs and outputs **streams** of data on "wires"
Complex filters made of elementary blocks connected by wires:

- **delays** (buffer for 1 or more clock tick(s))
- multiplication by a scalar
- addition of 2 streams

Network topology may contain **feedback loops** going through a delay

Example of complex filter



Each TF2 element itself a complex filter with internal filter feedback loop.

Causal, time-invariant linearity

Causal: values at clock tick $n \geq 0$ depend on those at clock ticks $\geq n$ only
(Non causal filters typically need entire buffering of the data — we do not cover them here.)
Linearity: outputs a **linear** function of the inputs (**over the reals**)
$\Rightarrow$ each output at time n a linear function of the inputs at times $\geq n$
Time invariance: this function is always the same
convolution, i.e. $o^{(n)} = \sum_k t^{(k)} i^{(n-k)}$

Transfer function

Several inputs and initial values in the "delay" operators:

$$o^{(n)} = \sum_x \sum_k r_x^{(k)} i_x^{(n-k)} + \sum_y k_y d_y^{(n)}$$

Define $O = \sum_{n=0}^{\infty} o^{(n)} z^n$ a formal power series. The equation becomes

$$O = \sum_x T_x . I_x + \sum_y k_y . D_y$$

k_y initial value of delay labeled y;
I_x series for input stream labeled x; T_x **unit response** for input x (**Z-transform**)

Bounding the transfer

We know some bound $[-B_x, B_x]$ on input I_x: $\|I_x\|_\infty \leq B_x$.

What is the $\|I \mapsto T.I\|$ **operator norm** w.r.t $\|\cdot\|_\infty$?
I.e. the least M such that $\|T.I\|_\infty \leq M.\|I\|_\infty$.

Answer: $\|I \mapsto T.I\| = \|T\|_1$ with $\|T\| = \sum_n |t_n|$.

Then

$$\|O\|_\infty \leq \sum_x \|T_x\|_1.\|I_x\|_\infty + \left\|\sum_y k_y.D.y\right\|_\infty$$

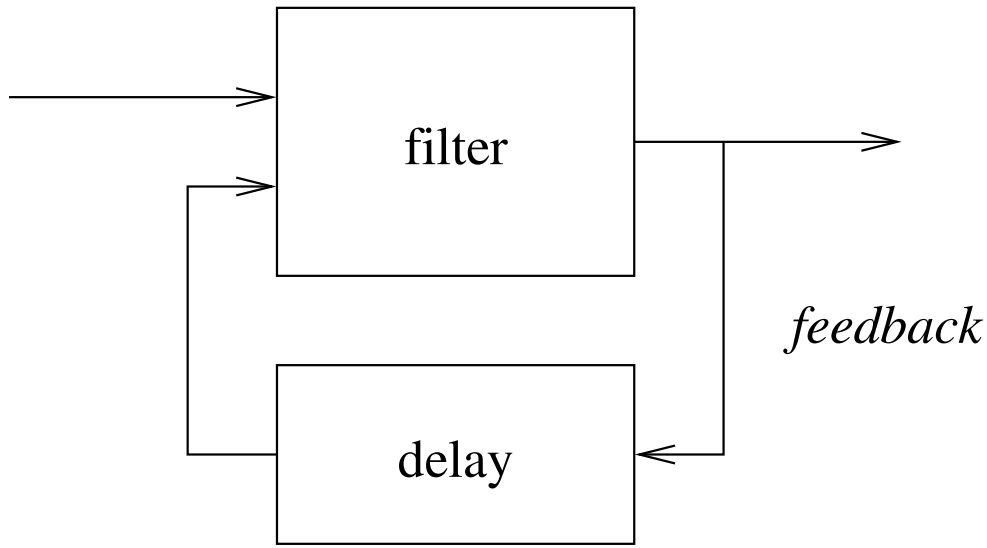
Examples

Multiplication by a scalar: $T = \alpha$,

Addition: $T_1 = 1,\ T_2 = 1$

Delay: by n clock ticks, $T = z^n$

Feedback loop



Filter F with two inputs I and L, output L fed back into L through unit delay (we leave out the initialization):
$O = T_I.I + T_L.L = T_I.I + T_L.zO$ and thus $O = T.I$ with

$$T = (1 - zT_L)^{-1}.T_I.I$$

Rational functions

All the T power series that we construct are the developments around 0 of **rational functions** $P(z)/Q(z)$ (P, Q polynomials, $Q(0) = 1$) — ring $\mathbb{R}[z]_{(z)}$.
If a filter has m inputs $I_1, \ldots, I_n$, r initialization values $k_1, \ldots, k_r$, and n outputs $O_1, \ldots, O_m$, then
T is a $n \times m$ matrix over $\mathbb{R}[z]_{(z)}$; D is a $n \times r$ matrix over $\mathbb{R}[z]_{(z)}$;
K is a m-vector of $\mathbb{R}$;
I is a m-vector of $\mathbb{R}[z]_{(z)}[[z]]$ (series); O is a n-vector of $\mathbb{R}[z]_{(z)}[[z]]$ (series) and

$$O = T.I + D.K$$

Feedback loops

Take a filter F (T^F, $D^F \dots$), feedback its n outputs into the last of its m inputs.
$O = T_1^F.I + T_2^F.zO + D.K$ and thus

$$O = (\mathrm{Id}_n - z.T_2^F)^{-1}.(T_1^F.I + D.K)$$

(this matrix is necessarily **invertible**)
Computations doable over $\mathbb{Q}[z]_{(z)}$!

Summary

Any of the filters can be summarized by **matrices** of **rational functions** over the rationals.
These matrices can be computed simply from the coefficients of the various elementary blocks **or from the matrices of whole sub-filters** (compositional design).
Example: filter $O^{(n)} = \sum_{k=0}^{d} \alpha_k I^{(n-k)} + \sum_{k=1}^{e} \beta_k O^{(n-k)}$ has transfer function

$$\frac{\alpha_0 + \alpha_1 z + \cdots + \alpha_d z^d}{1 - \beta_1 z - \cdots - \beta_e z^e}$$

$d = e = 2$: TF2 filter in our example

Bounding the output

Let N_x apply $\|\cdot\|_x$ to all coordinates in a matrix or vector.
Then $N_\infty(O) \leq N_1(T).N_\infty(I) + N_\infty(D.K)$
The main problem: given a rational function $P(z)/Q(z)$, $Q(0) \neq 0$, **give an upper bound on** $\|P/Q\|_1$ (of its development around 0).
Idea:

- compute explicitly the first N terms of this development, compute a bound for $\|P/Q\|_1^{<N}$
- bound the tail: $\|P/Q\|_1^{\geq N}$

Explicit development

Development of P/Q: division by ascending powers of $P(z)$ by $Q(z)$ (eqv. to running a filter $O^{(n)} = \sum_{k=0}^{d} p_k I^{(n-k)} - \sum_{k=1}^{e} q_k O^{(n-k)}$).

Problem: **numerical instability** using **interval arithmetics on floating-point numbers**.

After a while, error on the same order as the coefficients, **sign of the coefficients unknown**, then quick amplification.

Solution: develop until sign of the coefficients unknown. (Can go further with extended-precision arithmetic, see GNU MP's MPFR).

Tail bounding

Poles: the zeroes of $Q(z)$ are called **poles** of $P(z)/Q(z)$
The poles determine the behavior of the system.
Theorem: system stable ($\|P/Q\|_1 < \infty$) iff all poles have absolute value ¿ 1
Intuition: (distinct poles) $[P(z)/Q(z)]^{(n)} = \sum_i \alpha_i \xi^{-n}$, ξ_i poles.
Theorem: let R be the remainder of the division by ascending powers of P/Q up to order N, then

$$\|P/Q\|_1^{\geq N} \leq \frac{\|R\|_1}{(|\xi_1| - 1) \dots (|\xi_n| - 1)}$$

Tail bounding

Implementation: Good algorithms and libraries (GSL...) for finding approximate roots $\tilde{\xi}_i$ of polynomial Q
Methods for sound bounds on the error on a root
$(|\tilde{\xi}_i - \xi| \leq e(Q, \tilde{\xi}_i))$

Decomposition

Let $\tilde{O}$ be the output of the filter implemented in fixed- or floating- point, O the ideal real output, then we obtain for single input, output, no initialization:

$$\|\tilde{O} - O\|_\infty \leq \varepsilon_{\text{rel}}.\|I\|_\infty + \varepsilon_{\text{abs}}$$

ε_{rel} **relative error**, ε_{abs} **absolute error**.
In general: $N_\infty(\tilde{O} - O) \leq \varepsilon_{\text{rel},T}.N_\infty(I) + \varepsilon_{\text{rel},D}.N_\infty(K) + \varepsilon_{\text{abs}}$
with $\varepsilon_{\text{rel},T}$, $\varepsilon_{\text{rel},D}$ matrices in $\mathbb{R}_+$, ε_{abs} vector in $\mathbb{R}_+$

Error in elementary blocks

$x \oplus y$ in fixed- or floating-point $= r(x + y)$, $x \otimes y$ in fixed- or floating-point $= r(x.y)$; r **rounding function** such that $|r(x) - x| \leq \varepsilon_{\text{rel}}.|x| + \varepsilon_{\text{abs}}$

Fixed-point: $\varepsilon_{\text{rel}} = 0$, $\varepsilon_{\text{abs}} = u/2$ (round-to-nearest), $\varepsilon_abs = u$ (other modes) with u the least representable positive number

Floating-point: ε_{rel} error at the n-th binary position after the point

ε_{abs} is the **denormalization** error: ex, if u is the least representable positive number, then $0.25 \otimes u = 0$ (ε_{abs} very small, but included for soundness)

$|r(x) - x| \leq \max(\varepsilon_{\text{rel}}.|x|, \varepsilon_{\text{abs}})$ overapproximated by affine form

Error propagation

Simple blocks: With the above: easy for addition and multiplication by scalar, no error on delays

Feedback loop: around filter F: somewhat complex computation ending up with $\varepsilon_{\text{rel}} = A.\varepsilon_{\text{rel}} + B$ with A with very small coefficients $\Rightarrow$ resolution by fixpoint iteration

Simplification: with a P/Q filter (P, Q with rational coefficients, can be large), replace by $\tilde{P}/\tilde{Q}$ (approximations for P and Q) with some larger ε_{rel}

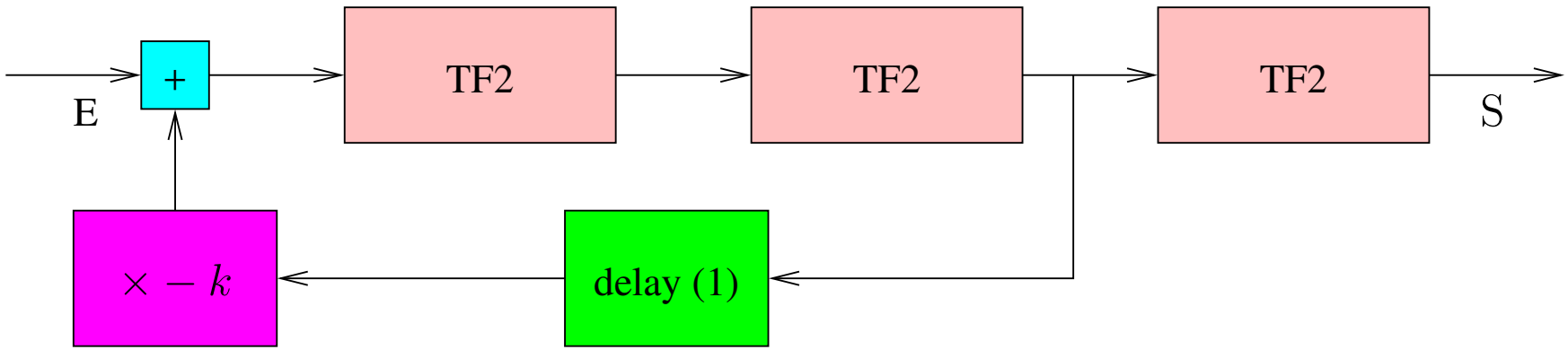
To summarize

Any causal linear filter F with finite memory implemented over fixed-point or floating-point (or a mix thereof) can be summarized into:

- matrices T^F and D^F over $\mathbb{Q}[z]_{(z)}$ expressing the ideal, real input-output relationship by **Z-transform**: T wrt input streams, D wrt values initially in the delay memories
- matrices $\varepsilon^F_{\mathrm{rel},T}$ and $\varepsilon^F_{\mathrm{abs},D}$ over $\mathbb{R}_+$ expressing the **relative errors** wrt input streams and delay memories
- vector $\varepsilon^F_{\mathrm{abs}}$ of **absolute error**

This is **compositional**: computation for complex filters from the analysis results of sub-filters.

Implementation results



Defined compositionally in the analyzer
Computation in 0.1s (could be optimized), P/Q P of 6th degree, Q of 7th degree
$\varepsilon_{\text{rel}} \leq 4.781.10^{-13}$, $\|O\|_\infty \leq 2.0576\|I\|_\infty$.

Reconstruction of filters

From C or similar program (SSA form)

```
while (1) {
   ...
   filter
}
```

Read all lines in filter, obtain $v = e$ equations from $v := e$ assignment; variables v in e not already initialized in loop become $z.v$

In case of nonlinearity: use approximation to remove the linear part and get large ε_{rel}.

Solve the resulting system.

Summary

Compositional abstract semantics for fixed- and floating-point digital linear filters with fixed memory of **arbitrary complexity**.
Sound bounds on the output obtained as affine function of bounds on the inputs.
Simple implementation already gives good results.
Good for analysis of data-flow languages for automatic systems (graphical Scade etc.). Extension for imperative languages.

Plan

Relational Domains on Floating-Point

Problems:

- Relational numerical abstract domains rely on a **perfect mathematical concrete semantics** (in $\mathbb{R}$ or $\mathbb{Q}$).
- Perfect arithmetics in $\mathbb{R}$ or $\mathbb{Q}$ is costly.
- IEEE 754-1985 floating-point concrete semantics **incurs rounding**.

Verimag

Solution

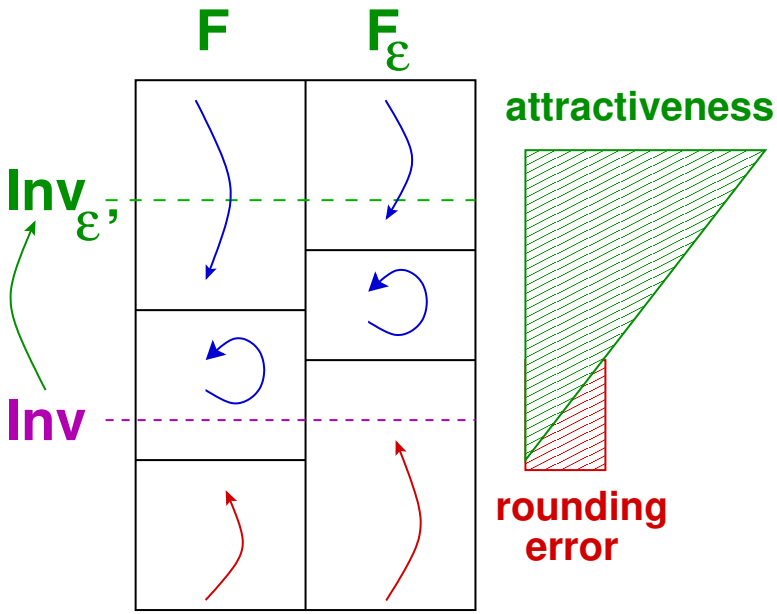
- Build an **abstract mathematical semantics** *in* $\mathbb{R}$ *that over-approximates the concrete floating-point semantics, including rounding.*
- *Implement the abstract domains on* $\mathbb{R}$ **using floating-point numbers** *rounded in a sound way.*

Fixpoint Stabilization for Floating-point

Problem:

- Mathematically, we look for an abstract invariant **inv** such that $F(\mathbf{inv}) \subseteq \mathbf{inv}$.
- Unfortunately, abstract computation uses floating-point and incurs rounding: maybe $F_\varepsilon(\mathbf{inv}) \not\subseteq \mathbf{inv}$!

Solution:



- Widen **inv** to $\mathbf{inv}_{\varepsilon'}$ with the hope to jump into a **stable zone of F_ε**.
- Works if F has some **attractiveness** property (otherwise iteration goes on).
- ε' is an analysis parameter.

Plan

Machine words

Most relational abstractions (polyhedra, octagons) assume **unbounded integers** or reals.

Most programming languages (C, Java...) operate upon fixed-width integers:

- with wrap-around semantics ($2^{30} \oplus 2^{30} = -2^{31}$)
- overflows as "undefined behaviours"

Verimag

Operations

- $+$, $-$, $\times$
- $/$, mod need some precision on rounding ($-1/3 = -1 or = 0$?)
- binary and, or, not, xor, bitshift

Programmers may mix different kind of operations

EXERCISE

Design a non relational abstract domain for integers

- supports overflow and “undefined behaviour” semantics
- can be queried for intervals
- carries bitwise information

Plan

Pointer analysis

```
int *p, x, y;
x = 4;
p = &x;
...
*p = 3;
```

Simple solution

- Variables whose address is never taken (&) are retained.
- Others are discarded, writes to them ignored, read considered "nondeterministic choice".
  Or discard only after first (&).
- Writes to pointers are ignored (still need to track valid/invalid pointers for runtime errors).

Not very precise!

More advanced

e.g. llvm `mem2reg`

```
int *p, x, y;
x = 4;
p = &x;
...
*p = 3;
... /* does not touch p or x*/
assert(x == 3);
```

Can deduce that $*p$ is still 3.

Steensgaard's method

Builds a graph of **equivalence classes** of variables
(**union-find** data structure)
Flow-insensitive (global information computed)

If x may point to y and x may point to z, alors $y \sim z$

```
int **p, *x, *y, z, t;
x = &z;
y = &t;
if (toto()) p = &x; else p=&y;
```

$p \longrightarrow \{x, y\} \longrightarrow \{z, t\}$

Refinement?

```
int *p, x, y, z;
if (toto) { p = &x; } else { p = &y; }
*p = 42;
if (toto) { z = x; } else { z = y; }
assert(z == 42);
```

Refinement!

```
#define X 1
#define Y 2
int p_points_to, x, y, z;
if (toto) { p_points_to= X; } else { p_points_to

if (p_points_to==X) x = 42;
else if (p_points_to==Y) y = 42;
else ???

if (toto) { z = x; } else { z = y; }
assert(z == 42);
```

Introduce a "selector variable" among possible points-to.

Summary locations

Above schemes work if finite set of variables, known statically.

How about dynamic memory allocation?

Create a new variable per control location, where malloc is called:

```
p1 = malloc(128);
p2 = malloc(128);
```

$\longrightarrow$

```
typedef m128 char
m128 x, y;
p1 = &x;
p2 = &y;
```

Possible problems?

Problems

- mymalloc()-style wrappers (one single location calls malloc)
  Can be dealt with by partition by call-stack
- loops

Need to introduce summary cells!

Allocation in loop

```
char *q;
for(int i=0; i<n; i++) {
  char *p = malloc(128);
  if (...) q = p;
}
```

Introduce a common "malloc128" variable? But it refers to several blocks allocated!?

Smashed arrays

```
int t[n];
...
for(int i=0; i<n; i++) t[i] = 42;
```

"smash" all cells into one

```
int t;
...
for(int i=0; i<n; i++) t = 42;
```

s this correct?

Incorrect!

```
int t[n];
...
t[0] = 42;
```

cannot be treated as though 42 in all cells!

Distinguish

Strong updates overwrites

```
t = 42;
```

Weak updates "add possible value"

```
if (*) t = 42;
```

Arrays, data structures in general

Hard problem

- **separation logic**
- pre-analysis of memory, split into regions
- **shape analysis**
- "cell morphing"