Domain-Specific Abstractions & How to Pick a Good Domain

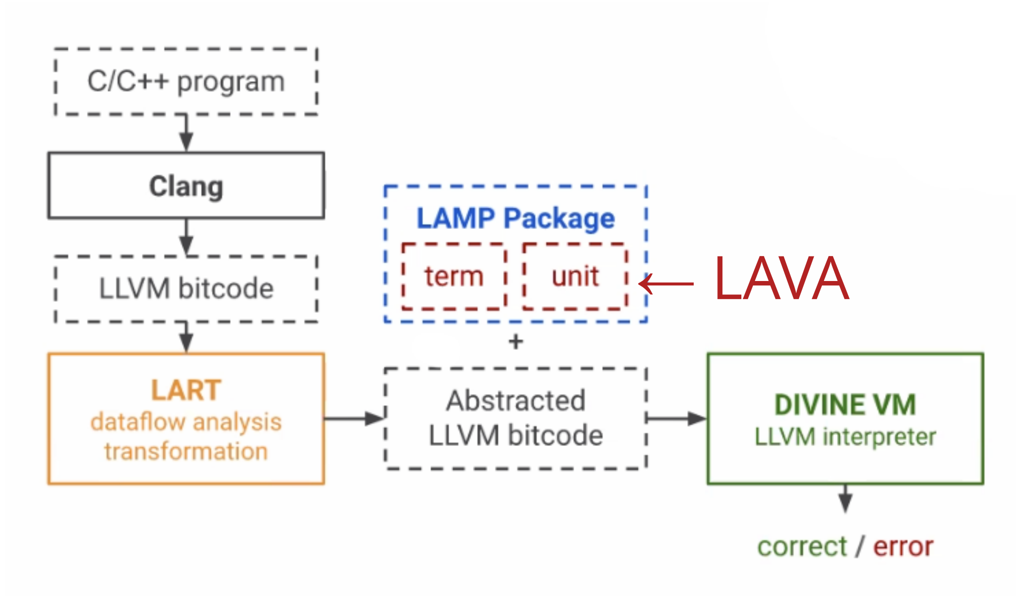
Hugo Adamove

Introduction & Fast Recap

- Program analysis with non-deterministic values
- How to effectively represent these values
- Symbolic vs abstract representation

Abstract Domains

- Unit
- Term
- Zero
- Interval
- ...



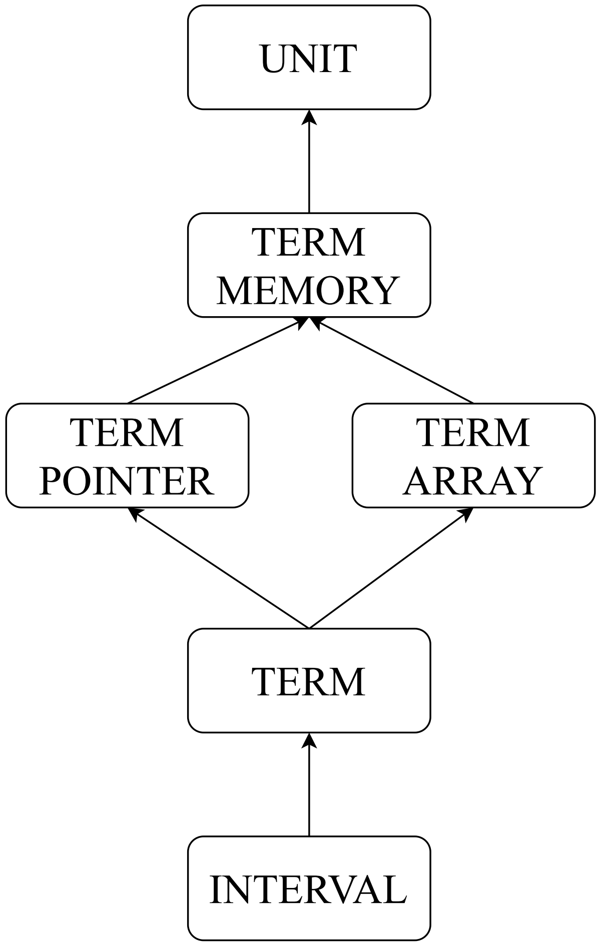
Unit

- Only one value that represents the whole set of values
- Ignore values of variables
- The most "coarse-grained" abstraction

Unit

- Only one value that represents the whole set of values
- Ignore values of variables
- The most "coarse-grained" abstraction
- Can be used to reduce the program state space

LAMP



Zero Abstract Domain

- Remembers whether the value is zero (Z), non-zero (N) or unknown ($\top$)
- Useful for detecting possible division by zero and null pointer dereference

$$\begin{tikzcd}
 & \top \arrow[dl, dash] \arrow[dr, dash] & \\
Z \arrow[dr, dash] & & NZ \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

Example Operations

$$Z + Z = Z$$

$$Z + N = N$$

$$N + N = \top$$

$$\top \times Z = Z$$

$$Z \bmod N = Z$$

Example Operations

$$Z + Z = Z$$

$$Z + N = N$$

$$N + N = \top$$

$$\top \times Z = Z$$

$$Z \bmod N = Z$$

op. add (a + b)			
a \ b	Z	N	T
Z	Z	N	T
N	N	T	T
T	T	T	T

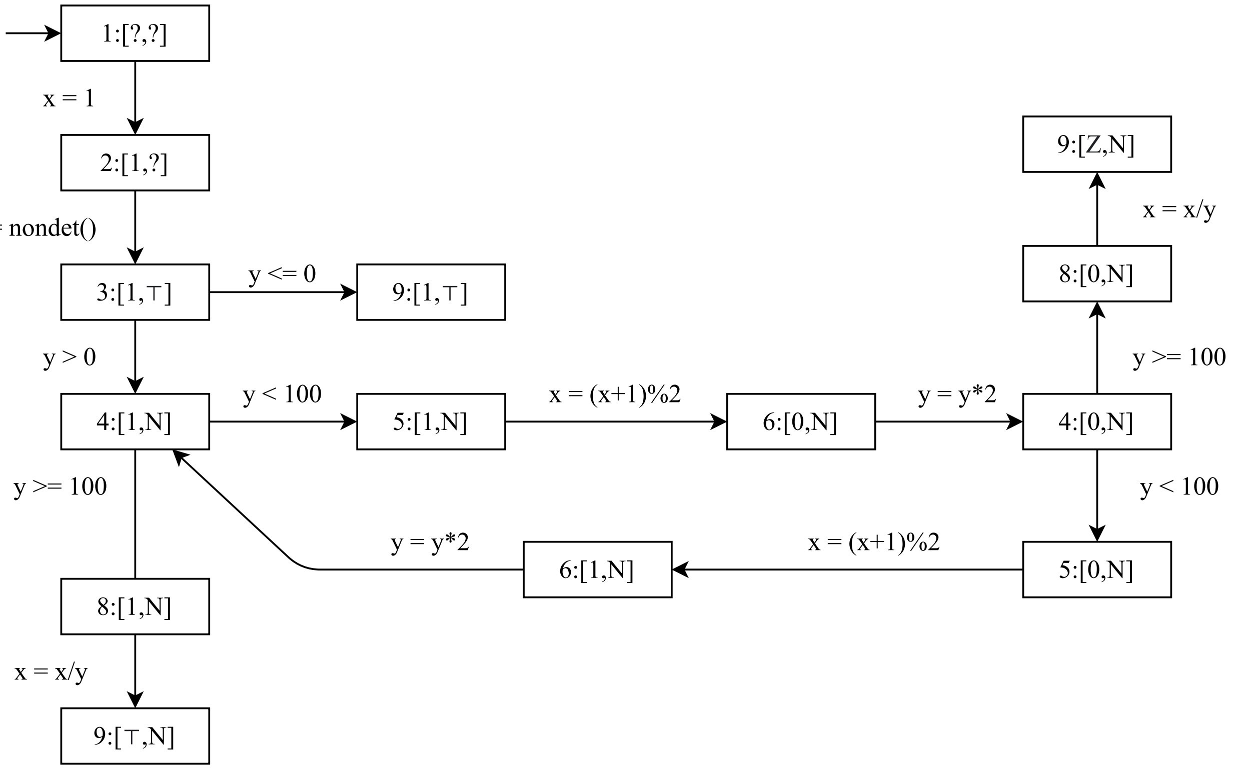
op. mul (a * b)			
a \ b	Z	N	T
Z	Z	Z	Z
N	Z	N	T
T	Z	T	T

cmp. ugt (a > b)			
a \ b	Z	N	T
Z	Z	Z	Z
N	N	T	T
T	T	T	T

bitop. or (a | b)			
a \ b	Z	N	T
Z	Z	N	T
N	N	N	N
T	T	N	T

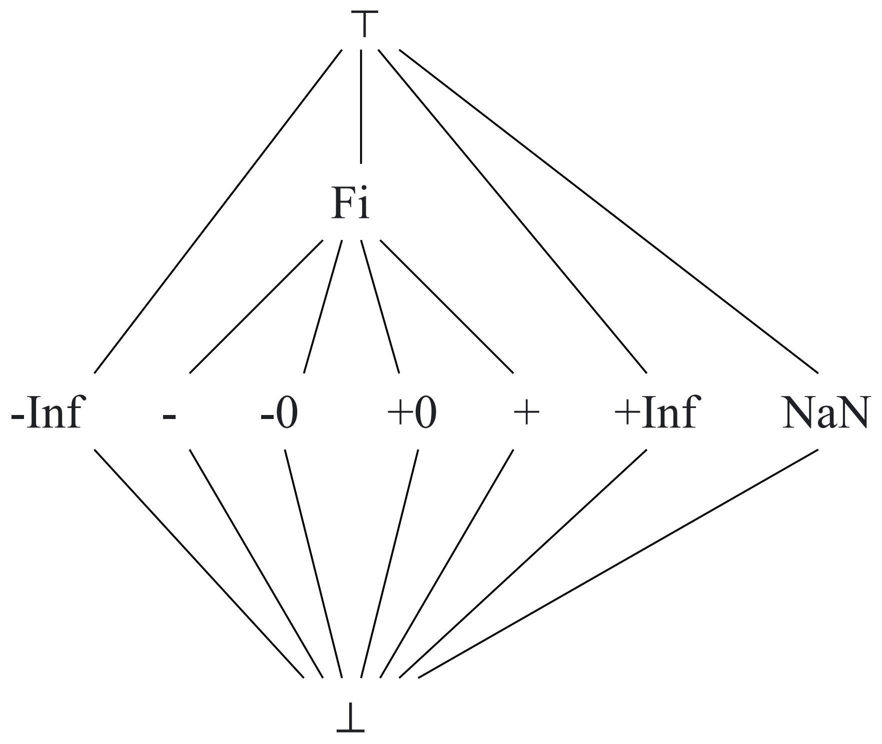
Example State Space Reduction

```
1 int x = 1;
2 int y = nondet();
3 if (y > 0) {
4     while (y < 100) {
5         x = (x + 1) % 2;
6         y = y * 2;
7     }
8     x = x / y;
9 }
```



Floats Abstract Domain

- Extension of zero domain
- Float edge cases



Example Operations

$$\oplus + \oplus = \oplus$$

$$\ominus + Inf = Inf$$

$$Inf - Inf = NaN$$

$$\ominus \div\ 0 = -Inf$$

Example Operations

$$\oplus + \oplus = \oplus$$

$$\ominus + Inf = Inf$$

$$Inf - Inf = NaN$$

$$\ominus \div \; 0 = -Inf$$

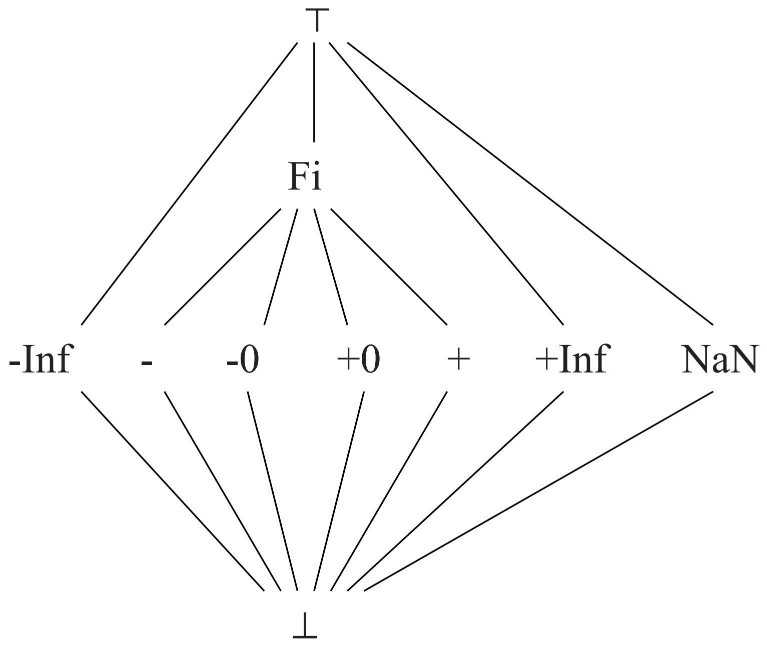
floats division (a ÷ b)									
a \ b	NEG	-Z	+Z	POS	FI	-INF	INF	NAN	T
NEG	POS	INF	-INF	NEG	T	+Z	-Z	NAN	T
-Z	+Z	NAN	NAN	-Z	T	+Z	-Z	NAN	T
+Z	-Z	NAN	NAN	+Z	T	-Z	+Z	NAN	T
POS	NEG	-INF	INF	POS	T	-Z	+Z	NAN	T
FI	FI	T	T	FI	T	FI	FI	NAN	T
-INF	INF	INF	-INF	-INF	T	NAN	NAN	NAN	T
INF	-INF	-INF	INF	INF	T	NAN	NAN	NAN	T
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN
T	T	T	T	T	T	T	T	NAN	T

Smashing Array

- Functor domain
- Arbitrary length array represented by one value

Example

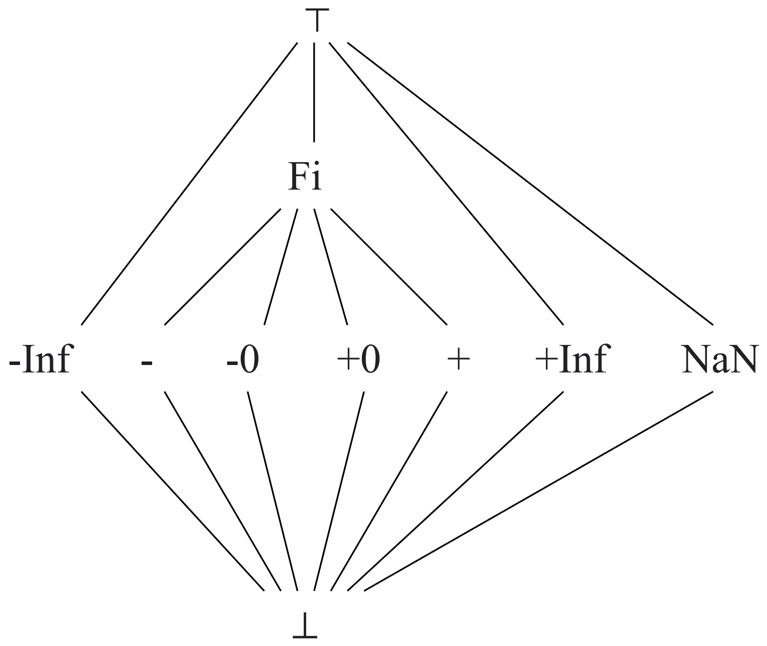
- SmashingArray(Floats)
- Store instruction acts as join (supremum) in the Floats lattice



Example

- SmashingArray(Floats)
- Store instruction acts as join (supremum) in the Floats lattice

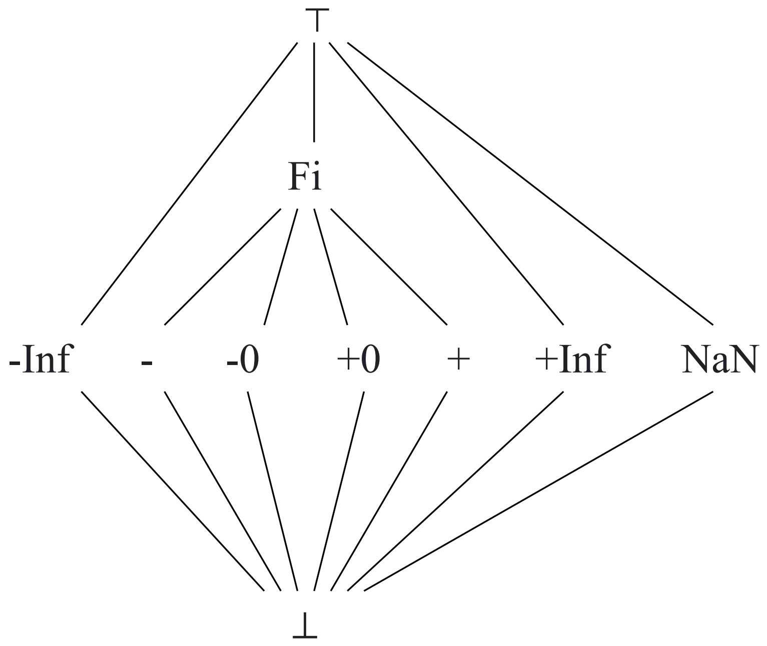
`store([⊥], +) → [+]`



Example

- SmashingArray(Floats)
- Store instruction acts as join (supremum) in the Floats lattice

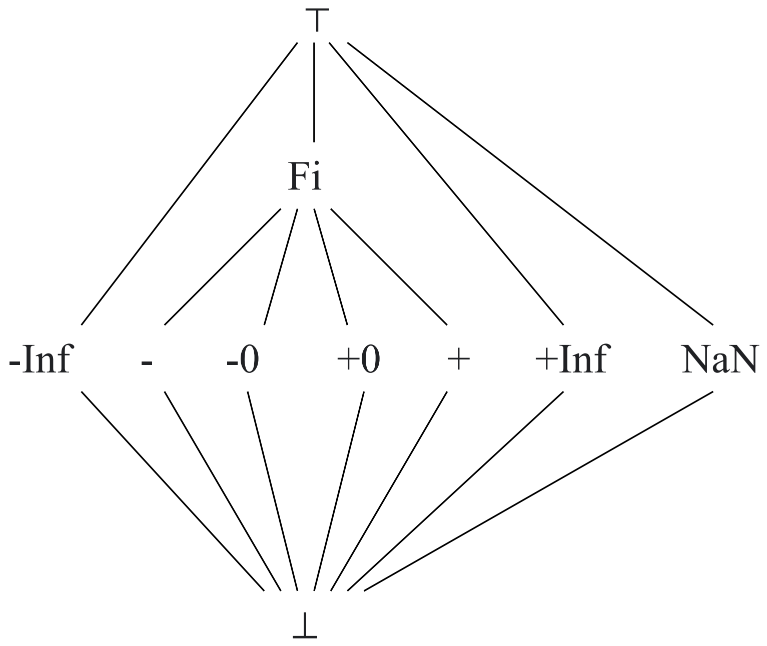
```
store([⊥], +) → [+]
store([+], +0) → [Fi]
```



Example

- SmashingArray(Floats)
- Store instruction acts as join (supremum) in the Floats lattice

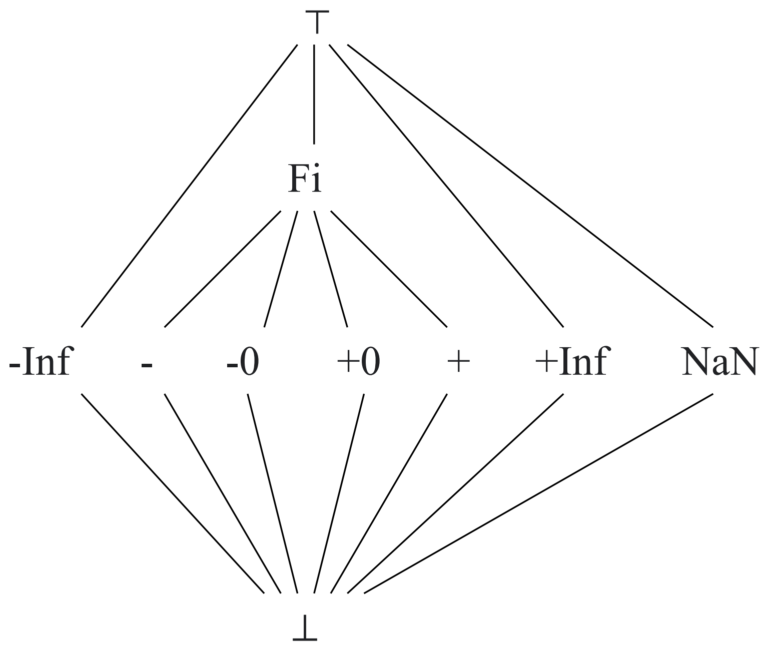
```
store([⊥], +) → [+]
store([+], +0) → [Fi]
store([Fi], -) → [Fi]
```



Example

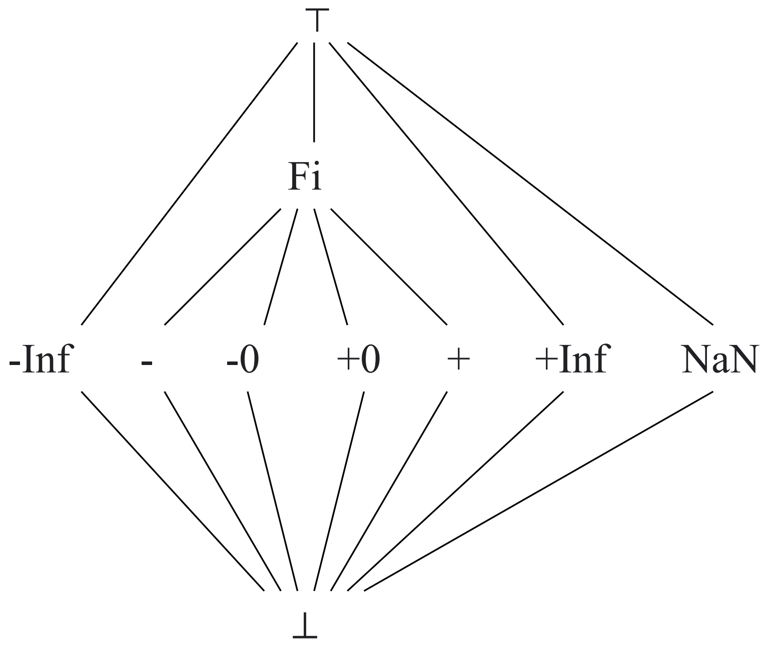
- SmashingArray(Floats)
- Store instruction acts as join (supremum) in the Floats lattice

```
store([⊥], +) → [+]
store([+], +0) → [Fi]
store([Fi], -) → [Fi]
store([Fi], +Inf) → [⊤]
```



Example

- SmashingArray(Floats)
- Store instruction acts as join (supremum) in the Floats lattice

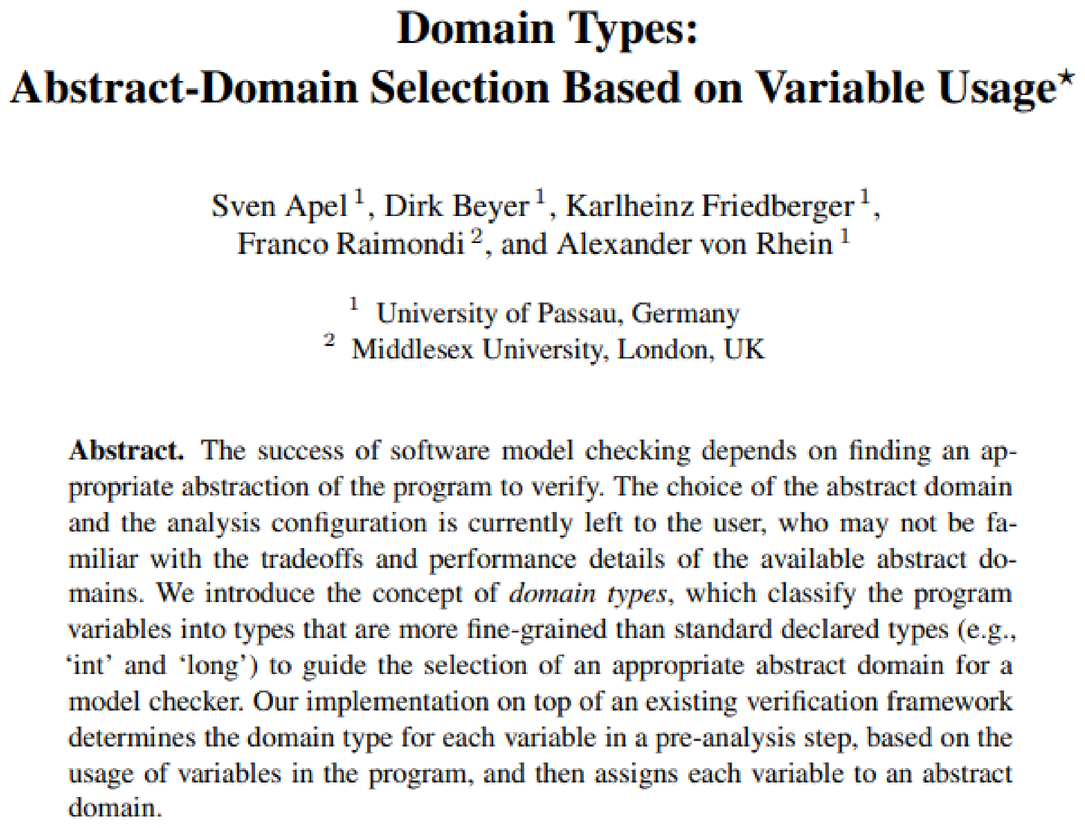


```
store([⊥], +) → [+]
store([+], +0) → [Fi]
store([Fi], -) → [Fi]
store([Fi], +Inf) → [⊤]
store([(1,3)], (2,4)) → [(1,4)]
```

How to Pick a Good Domain

- The choice of the abstract domain and representation usually left to the user
- Use static program pre-analysis to automize this task

How to Pick a Good Domain



Domain Types:
Abstract-Domain Selection Based on Variable Usage*

Sven Apel[1], Dirk Beyer[1], Karlheinz Friedberger[1],
Franco Raimondi[2], and Alexander von Rhein[1]

[1] University of Passau, Germany
[2] Middlesex University, London, UK

Abstract. The success of software model checking depends on finding an appropriate abstraction of the program to verify. The choice of the abstract domain and the analysis configuration is currently left to the user, who may not be familiar with the tradeoffs and performance details of the available abstract domains. We introduce the concept of *domain types*, which classify the program variables into types that are more fine-grained than standard declared types (e.g., 'int' and 'long') to guide the selection of an appropriate abstract domain for a model checker. Our implementation on top of an existing verification framework determines the domain type for each variable in a pre-analysis step, based on the usage of variables in the program, and then assigns each variable to an abstract domain.

https://www.researchgate.net/publication/262601561

How to Pick a Good Domain

- Pre-analysis assigns Domain Type to each variable
- Each Domain Type assigned to Abstract Domain

Domain Types:

1. **IntBool** - negations (!), zero equality test (== 0, != 0)
2. **IntEqBool** - equality test with non-zero, or other variable (== x, != x)
3. **IntAddEqBool** - linear arithmetic (+, –), arbitrary comparisons (==, !=, <, >, <=, >=), bit operators (&, |, ^)
4. **IntAll** - all other operations

Example

- Driver code in C:

```c
int enabled, a, b;
b = 20;
if (enabled) {
    if (a > 5) {
        if (a == 0) {
            b = 0;
        }
        assert(b * b > 200)
    }
}
```

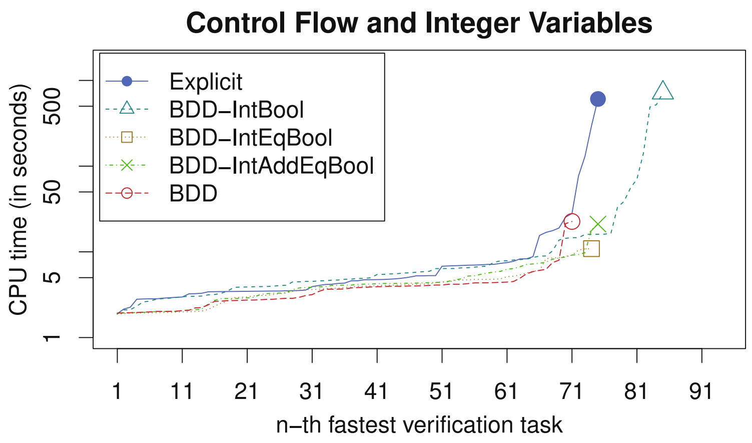
Example

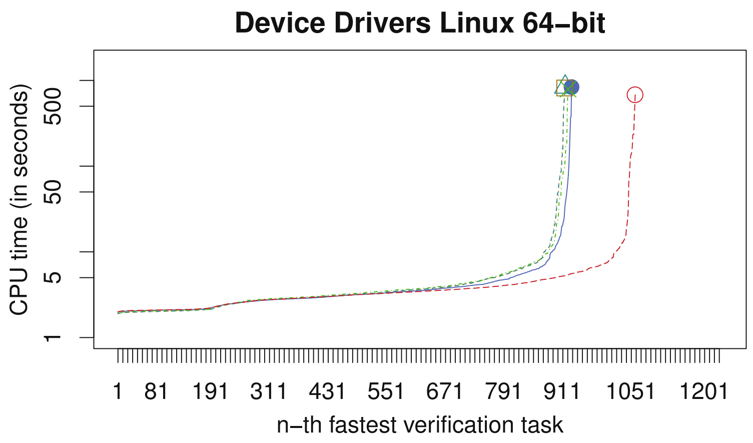
- Driver code in C:

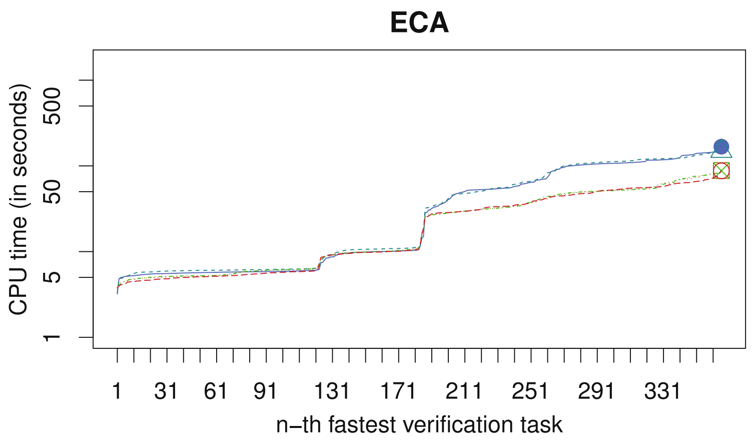
```c
int enabled, a, b;
b = 20;
if (enabled) {
    if (a > 5) {
        if (a == 0) {
            b = 0;
        }
        assert(b * b > 200)
    }
}
```

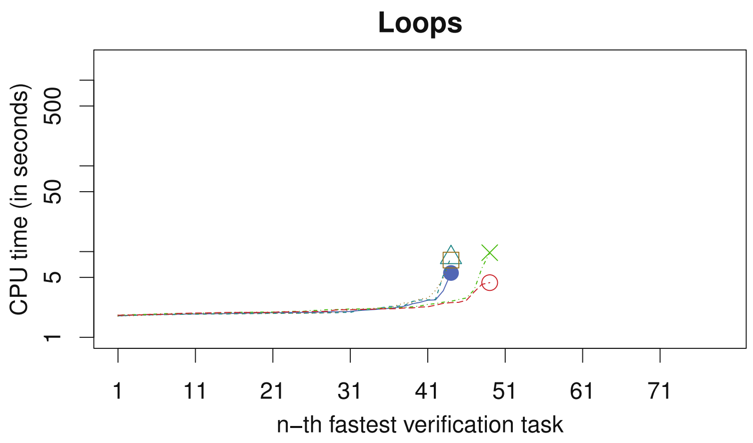
```
enabled :: IntBool
a :: IntAddEqBool
b :: IntAll
```

Benchmark Results









https://www.researchgate.net/publication/262601561

Domain-Specific Abstractions

- "Coarse-grained" abstract interpretation trades precision for state space reduction
- Zero/Floats domain can be useful for detecting division by zero or null pointer dereference with little impact on the state space
- Static program analysis based on variable usage can help us with choosing good domains for variables