Astrée: Building a Static Analyzer for Real-Life Programs

Antoine Miné

CNRS & École normale supérieure
Paris, France

Workshop on Learning From eXperience
June 6th, 2010

What is Astrée ?

Analyse Statique Temps RÉEl (Real-Time Static Analysis)

- checks statically for the absence of run-time errors (RTE) on a subset of C for embedded applications
- semantic-based, sound, based on Abstract Interpretation
- specialized for synchronous reactive real-time codes : 0 alarms

Development team : P. Cousot, R. Cousot, L. Mauborgne, J. Feret, A. Miné, X. Rival, *B. Blanchet*, *D. Monniaux*

`http://www.astree.ens.fr`

`http://www.absint.com/astree`

History

Milestones

Nov. 2001	Astrée academic project starts
Nov. 2003	primary control software of the Airbus A340 analyzed
	proof of the absence of RTE
Apr. 2005	maiden flight of the A380
	proof of the absence of RTE (for current version)
Sep. 2008	study on applicability to space software with ESA
Jan. 2009	start industrialization with AbsInt GmbH

- initially developed at École Normale Supérieure & CNRS, France
- now developed & commercially available from AbsInt, Germany

Engineering Facts

Analyzer :

- 80 K lines of OCaml
- 10 K lines of C (one module + low-level code)
- 150 command-line options
- (+ GUI & integration in AbsInt tool-chain)

Analyzed codes :

- 80 K to 715 K lines of C
- in large part generated from a proprietary synchronous language
- 40 mn to 62 h analysis time (Intel Xeon 2.66Ghz, 64-bit, 1 core)
- smallest fit in 2 GB RAM, largest fit in 32 GB

Overview

- semantic-based static analysis
    - Abstract Interpretation
- defining a semantics for C
    - the C norm
    - Astrée's concrete semantics
    - Astrée's abstract semantics
- selected topics
    - algorithms & data-structures
    - abstract domains
- conclusion

Semantic-Based Static Analysis

Review of Verification Methods

Testing

- well-established method
- but no formal warranty, high cost

Review of Verification Methods

Testing

- well-established method
- but no formal warranty, high cost

Formal methods :

Theorem proving

- proof essentially manual, but checked automatically
- powerful, but very steep learning curve

Model checking

- checks a model of the program (usually user-specified, finite)
- automatic and complete (wrt. model), but costly

Review of Verification Methods (cont.)

(Semantic-based) static analysis

- works directly on the source code (not a model)
- automatic, always terminating
- sound (full control and data coverage)
- incomplete (properties missed, false alarms)
- parameterized by one/several abstraction(s)
- mostly used to check simple properties,
  with low precision requirement (e.g., for optimisation)

Review of Verification Methods (cont.)

(Semantic-based) static analysis

- works directly on the source code (not a model)
- automatic, always terminating
- sound (full control and data coverage)
- incomplete (properties missed, false alarms)
- parameterized by one/several abstraction(s)
- mostly used to check simple properties, with low precision requirement (e.g., for optimisation)

Specialized static analyzer

- checks for run-time errors (overflow, etc.)
- is very precise on a chosen class of programs (no false alarm)
- gives sound results on all programs

Semantics

Concrete semantics :

- defines all relevant aspects of computation
  (variable and expression values, memory state, errors, etc.)
- formally defined, but generally undecidable
- solution of a recursive equation system (⇔ fix-point)

Example : numeric invariants

$$\mathcal{D} = \mathcal{P}(\{\mathtt{i}, \mathtt{a}[0], \ldots, \mathtt{a}[9]\} \to [-2^{31}, 2^{31}[)$$

```
a0: i = 0;
a1: while (i<10) {
a2:  a[i] = 0;
a3:  i++;
a4: }
a5:
```

$$\begin{array}{rcl}
X_0 & = & \mathcal{D} \\
X_1 & = & \{\, \rho[\mathtt{i} \mapsto 0] \mid \rho \in X_0 \,\} \cup X_4 \\
X_2 & = & \{\, \rho \mid \rho \in X_1,\ \rho(\mathtt{i}) < 10\} \\
X_3 & = & \{\, \rho[\mathtt{a}[\rho(\mathtt{i})] \mapsto 0] \mid \rho \in X_2 \,\} \\
X_4 & = & \{\, \rho[\mathtt{i} \mapsto \rho(\mathtt{i}) + 1] \mid \rho \in X_3 \,\} \\
X_5 & = & \{\, \rho \mid \rho \in X_1,\ \rho(\mathtt{i}) \geq 10\}
\end{array}$$

$\Longrightarrow$ no overflow on `i`, `a` is initialized to 0

Abstract Interpretation

Abstract Interpretation

General theory of semantic approximation [Cousot–Cousot77,91]
Can be used to design effective, sound static analyses

Static approximation :

- choose an effective encoding $\mathcal{D}^\sharp$ of a set of properties of interest ($\gamma : \mathcal{D}^\sharp \rightarrow \mathcal{D}$ monotonic)
- for each operator $F : \mathcal{D} \rightarrow \mathcal{D}$ used in the semantics design an abstract version $F^\sharp : \mathcal{D}^\sharp \rightarrow \mathcal{D}^\sharp$ (algorithm) (soundness condition : $\gamma \circ F^\sharp \supseteq F \circ \gamma$)

Dynamic approximation :

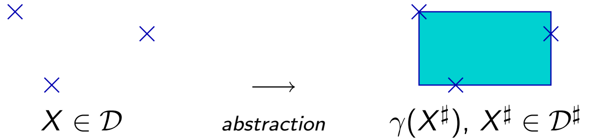
- needed to approximate fixpoints when $\mathcal{D}^\sharp$ has infinite chains
- provided by convergence acceleration operators $\triangledown$, $\vartriangle$
  (e.g., $X^\sharp_{i+1} = X^\sharp_i \triangledown F^\sharp(X^\sharp_i)$) converges in finite time to a post-fixpoint of $F^\sharp$)

Example Abstract Domain : Intervals

Definition :

- $\mathcal{D}^\sharp \overset{\text{def}}{=} \texttt{Var} \to (\mathbb{Q} \cup \{-\infty\}) \times (\mathbb{Q} \cup \{+\infty\})$
- $\gamma(X^\sharp) = \{\ \rho \mid \forall v \in \texttt{Var},\ \rho(v) \in [\ell, h],\ (\ell, h) = X^\sharp(v)\ \}$
- $F^\sharp$ implemented using interval arithmetics
  ($[\ell, h] +^\sharp [\ell', h'] = [\ell + \ell', h + h']$, etc.)
- convergence extrapolation : set unstable bounds to $\pm\infty$
  ($h \mathbin{\triangledown} h' =$ if $h < h'$ then $+\infty$ else h, etc.)

Abstraction : infer variable bounds, forget relations



Difficulties

We may not find the best invariant expressible in $\mathcal{D}^\sharp$:

```
a0: i = 0;
a1: while (i<10) {
a2:   a[i] = 0;
a3:   i++;
a4: }
```

at a1 : $\mathtt{i} \in [0, 10]$
$\forall j,\ \mathtt{a}[j] \in [-2^{31}, 2^{31}[$

$\Longrightarrow$ no overflow on `i` proven
0-initialization of `a` *not* proven

Causes :

- approximations accumulate
  (the combination of optimal abstract operators is not optimal)
- extrapolation $\triangledown$ introduces extra approximation
- the need to find inductive loop invariants of a complex form
  (e.g., $\forall \mathtt{j} \leq \mathtt{i}, \mathtt{a}[\mathtt{j}] = 0$)

An (infinite) abstract domain works on infinitely many programs and fails on infinitely many programs !

Construction by Refinement

Theoretical completeness :

- for each program and property, an abstract domain exists
- it's construction is not mechanizable

Practical approach

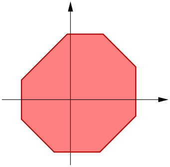
- build a simple and fast analyzer (intervals)
- refine the analyzer until 0 false alarm :
  - determine which necessary properties are missed
  - add / refine an abstract domain to infer it

Benefits

- sound by construction
- efficient (adapted cost / precision trade-off)
- encourages modular, reusable abstractions

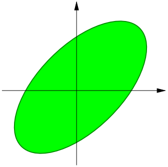
More Abstract Domain Examples

A few of the abstract domains used in Astrée.



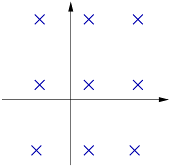
octagons

$\pm X \pm Y \leq c$



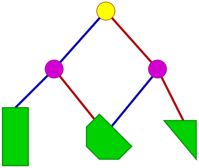
ellipsoids

digital filters

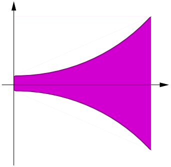


congruences

$X \equiv a\,[b]$

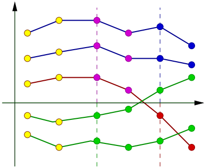


boolean decision trees



exponentials

$X \leq (1+\alpha)^{\beta t}$



trace partitions

The Difference Between Theory and Practice

Nice in theory. . . the reality is more complex :

- difficult to define a concrete semantics of C
- abstract domains are designed to abstract perfect numbers
  (not machine integers nor floats)
- abstract domains are sensitive to programming practices
  (non-expressible idioms cause catastrophic loss of precision)
- algorithms good for academic analyzers
  may not scale-up to large programs

A Semantics for C

Considered C Subset

Matches the restrictions of embedded critical software.

Handled

- C types : integers, enums, floats, structs, arrays, unions, bitfields
- pointers : aliases, arithmetics, type-punning
- structured control : `if`, `for`, `while`, `switch`
- forward `goto`, `break`, `return`
- function calls

Unhandled

- dynamic memory allocation
- recursivity
- libraries (stubs needed)
- backward `goto`
- `longjmp`
- variable argument number
- concurrency, threads (work in progress)

C Norm

Norms

- C99 norm (*portable programs*)
- IEEE 754-1985 norm (*floating-point arithmetics*)

But the C standard is largely underspecified :

- implementation-defined behaviors
  (documented, sensible semantics)
  (e.g. : overflow in cast to signed integer)
- unspecified behaviors
  (undocumented but sensible semantics)
  (e.g. : argument evaluation order)
- undefined behaviors
  (anything can happen)
  (e.g. : overflow in integer arithmetics)

Astrée Semantics

Fact

People do not write strictly conforming programs !

Some behaviors should be considered run-time errors,
others return a deterministic result. . .
some do both !

The semantics is refined by :

- platform-dependent choices :
  - range of types
  - bit-representation (`sizeof`, endianess, `struct` padding, etc.)
- compiler- and linker-dependent choices :
  - automatic variable initialization (optional)
  - symbol redefinition (forbidden)

Run-Time Errors

Run-time errors in Astrée

- overflow in float, integer, `enum` arithmetic and cast
- division, modulo by 0 on integer and float
- invalid right argument of bit-shift
- out-of-bound array access
- invalid pointer arithmetic or dereference
- violation of a user-specified assertion (`__ASTREE_assert`)

Some RTE alarms can be toggled on and off by the end-user.

Semantics After a Run-time errors

Several semantics are possible after an error :

- halt the program
  - division, modulo by zero
  - floating-point overflow
  - assertion failure
- unpredictable value returned (need to consider all values in type)
  - invalid bit-shift
- well-defined result
  - modulo on integer arithmetics overflow
  - type-punning
- unpredictable behavior (treated as halting the program)
  - invalid dereference

Some choices can be configured by the end-user.

It is important to continue the analysis after alarms !

Conflicting Side-Effects

An expression can have several side-effects,
only partially ordered by sequence points !

Undefined behavior

Between two sequence points, an object is modified more than once, or is modified and the prior value is accessed other than to determine the value to be stored.

Some undefined / unspecified cases are difficult to detect :

- `++(*a) - ++(*b)` (need pointer analysis)
- `f() + g()` (need to look inside `f` and `g`)

Current solution in Astrée : (not fully satisfactory)

- sound but coarse syntactic static expression analysis
  $\Longrightarrow$ issues warnings if conflicts found
- continue, assuming a left-to-right evaluation order

Semantics of Integers

Concrete semantics

- bounded integers in $[0, 2^n[$ or $[-2^{n-1}, 2^{n-1}[$ (2's complement)
- in case of error, issue an alarm and continue with :
  - either the whole range (e.g., overflow)
  - or a modular result (e.g., overflow)
  - or non-erroneous results only (e.g., division by 0)
- for `enums`, optional checking wrt. value *set*

Abstractions

- interval domain : can detect errors in expressions and has a decent abstract operator for modulo (more later. . .)
- relational domains : rely on the interval domain
  - no error : operate normally, assuming $\mathbb{Z}$ semantics (permitting symbolic reasoning, e.g. $(2 \times X)/2 \rightarrow X$)
  - error : abort and use the interval result

Semantics of IEEE 754-1985 Floats

Concrete semantics

- operation in $\mathbb{Q}$ followed by rounding to nearest, $+\infty$, -∞, or 0
- generating a $\pm\infty$ or *NaN* is a program-halting error
  (overflow, invalid operation)
- computing on $\pm\infty$ or *NaN* is a program-halting error
  (unchecked data from the environment)

Non-deterministic rounding issues :

- rounding mode can change during program execution
- precision of types not strictly respected
  (e.g., double rounding, fused add-multiply)

$\Rightarrow$ Astrée assumes $a \diamond b$ can be any number in $[a \diamond_{-\infty} b, a \diamond_{+\infty} b]$

Semantics of IEEE 754-1985 Floats (cont.)

Abstraction :

- interval domain :
  - enriched with flags to denote possible $\pm\infty$ and *NaN*
  - can detect errors in expressions
  - sound float interval arithmetics is very easy :

    round lower (upper) bounds towards $-\infty$ ($+\infty$)
    using the same precision as in the concrete operator

- relational domains :
  - disabled if the interval domain detects an error
  - transform expressions to make rounding error explicit
    ($\text{e.g., } \mathrm{x} + \mathrm{y} \rightarrow \mathrm{x} + \mathrm{y} + [-10^{-2}, 10^{-2}]$
    or $\mathrm{x} + \mathrm{y} \rightarrow (1 + [-10^{-6}, 10^{-6}])(\mathrm{x} + \mathrm{y}) + [-10^{-20}, 10^{-20}]$)

    the expression now has a semantics in $\mathbb{Q}$ (maths possible)
  - abstract computations use only floats internally (efficient)

Semantics of Aggregates

C aggregates : `structs`, `unions`, arrays

Concrete semantics (low level)

- variable = untyped contiguous sequence of bytes
- all memory accesses statically reduced to :
    - taking the address of a variable &
    - performing pointer arithmetics in bytes
    - dereferencing * scalar objects
      (integer, float, or pointer type)

⟹ access possible to the binary representation of types specified by an Application Binary Interface

Lesson learned

The first iteration of Astrée's memory model assumed only strict C norm well-structured accesses…
which did not match actual programming practice

Semantics of Aggregates (cont.)

A few examples. . .

Union

```
union {
  struct { uint8 al,ah,bl,bh } b;
  struct { uint16 ax,bx } w;
} r;
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
r.w.ax++;
```

Type-punning

```
uint8 buf[4] = { 1,2,3,4 };
uint32 i = *((uint32*)buf);
```

Ill-typed copy

```
float a,b;
*((int*)&a) = *((int*)&b);
```

All examples have :

- no error
- a well-defined semantics

Semantics of Aggregates (cont.)

Abstraction

Do not abstract values at the byte level !

We decompose dynamically the memory into cells of scalar type :

- cell = variable + offset + scalar type
- materialize new cells when needed by a dereference
  (possible reduction with existing cells)
- allow overlapping cells, with an intersection semantics

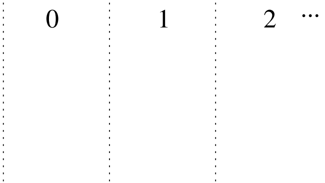
Orthogonality of abstractions

- numerical domains only see a collection of independent cells
  (abstract $\mathcal{P}(\text{cells} \to \mathbb{Q})$)
- a memory domain maintains the mapping cells ↔ memory
  (handles byte-level aliasing of cells)

Semantics of Aggregates (cont.)

Union

```
r.w.ax = 258 ;
if (r.b.al==2) r.b.al++ ;
r.w.ax++ ;
```

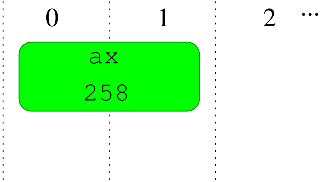


initial state : no cell ($\top$)

Semantics of Aggregates (cont.)

Union

```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
r.w.ax++;
```

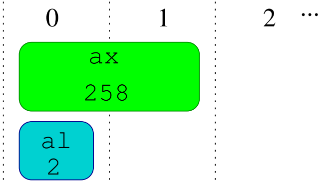


create `r.w.ax`, a `uint16` cell at offset 0

Semantics of Aggregates (cont.)

Union

```
r.w.ax = 258;
if (r.b.al==2) r.b.al++;
r.w.ax++;
```

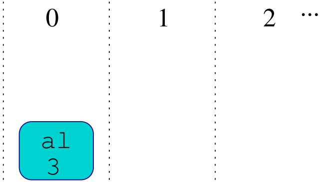


create `r.b.al`, a `uint8` cell at offset 0
initialized with : `r.w.ax` mod 256

Semantics of Aggregates (cont.)

Union

```
r.w.ax = 258 ;
if (r.b.al==2) r.b.al++ ;
r.w.ax++ ;
```



modify cell `r.b.al`
destroy invalidated cell `r.w.ax`

Semantics of Pointers

Concrete semantics

- pointer = variable base + byte offset
  or `NULL` or `invalid` pointer (uninitialized or dangling)
- pointer arithmetics = offset arithmetics
  (in `size_t`, `ptrdiff_t`)

  impossible to “jump” from one variable to another
- only one type of pointers (cast = identity)

Abstraction

- explicit set of bases for each variable (non relational)
- offsets as dimensions in numerical abstract domains
  (relations possibles between offsets and integers)

Memory and Pointer Errors

Memory errors :

- dereference out of a variable block
- unaligned dereference
- dereference of `NULL` or `invalid`

Pointer arithmetic errors :

- offset overflow `size_t`
- arithmetics on `NULL` or `invalid`
- `-`, comparison of pointers to different variables

Inherent limitations

- cannot check proper use of `unions`
- no offset bound-check inside variables

except array bound-checking :

- `t[i]` when the *static type* of `t` is of sized array (broken by type-punning)

Unreachable Code

Astrée computes at each program point an over-approximation of the set of possible environments
$\Longrightarrow$ $\bot$ indicates a necessarily unreachable location

Embedded code guideline

There shall be no unreachable code

But Astrée cannot prove the absence of unreachable code !

Confusion from some end-users :
they still ask us to output "necessarily unreachable" code

Abstractions,
Algorithms & Data-Structures

Syntax-Based Recursive Iterator

Equational static analysis :

- write an equation system
  (one variable & equation per program point)
- solve it by global iterated solving
  (many algorithms and iteration ordering)

Interpreter-based static analysis :

- start from the `main` function
- traverse the program recursively by induction on the syntax
  - analyze both branches of tests, and merge
  - iterate loops (with $\triangledown$) until stabilisation
  - step into functions at calls
    (inlining, i.e., full context-sensitivity)

Syntax-Based Recursive Iterator (cont.)

A syntax-based iterator is a special case of general equational one.

Advantages / drawbacks

- simple, predictable
- full context-sensitivity : biased towards precision at the cost of efficiency
- full context-sensitivity greatly simplifies the memory model (the set of live variables is exactly known)
- very memory efficient (minimizes the number of abstract elements kept in memory)
- can perform much recomputation (e.g., nested loops) (mitigated by cache techniques)

Local Abstractions

Some abstract domains are too costly :

- trace partitioning : exponential in the program size
- boolean decision trees : exponential in the number of booleans
- octagons : cubic in the number of variables

Solution : use them **locally**

- on (possibly many) small variable packs
- over (possibly many) small code blocks

$\Longrightarrow$ linear cost in the number of packs / blocks

Local Abstraction Heuristics

Syntactic intra-procedural heuristics to choose local abstractions :

- pack variables used together linearly within a syntactic block
  (`{ x=t; if (...) { x=y+z; ...} }`)
- partition multiple accesses to small arrays
  (`x = a[i+1] - a[i];`)
- partition assignments trailing after small loops
  (`for (i=0; i<10 && x<=a[i]; i++);`
  `y = (x-a[i])*b[i] + c[i]);`)
- etc.

Does not influence the soundness, only the precision
⟹ we can use unsound static analyses here
(e.g, dependency analysis that ignores aliases and calls)

Simpler to improve heuristics than to design new domains !

Functional Maps with Sharing

Main data-structure : functional maps as binary balanced trees

- maps cells / packs to abstract values
- maps offsets to overlapping cells, etc.

Natural sharing

- e.g., in tests, maps in both branches have a common ancestor

$\Longrightarrow$ memory efficient

Binary operators with shortcut

- e.g., `map2 f m1 m2` where $f(x,x) = x$ (idempotent)

sub-trees at the same address in `m1` and `m2` can be ignored

$\Longrightarrow$ time efficient

($\simeq$ from $\mathcal{O}(|\texttt{Var}|^2)$ to $\mathcal{O}(|\texttt{Var}| \cdot \log |\texttt{Var}|)$)

Expression Representations and Abstractions

Several levels of expressions :

- full C expressions (from source code)

  $\downarrow$ *static translation*
- side-effect free expressions (fed to memory domain)

  $\downarrow$ *dynamic translation*
- dereference-free expressions (fed to pointer domain)

  $\downarrow$ *dynamic translation*
- numeric, pointer-free expressions (fed to numeric domains)

  $\downarrow$ *optional dynamic translation*
- interval affine expressions (used by some relational domains)

Interval affine expressions : $[a_0, b_0] + \sum_i [a_i, b_i] X_i$

- semantics in $\mathbb{Q}$ (good for relational domains)
- affine → easy to manipulate
- interval → abstracts non-linearity as non-determinism

  (e.g., rounding error, non-linear integer operators)

Reduction Network

Problem : how to combine several abstract domains ?

Theoretical solution : fully reduced product

- optimal precision
- non-constructive (algorithms need to be designed)
- algorithms may be very costly

Practical solution :

- *a priori* fully independent domains (most efficient)
- equip domains with communication channels so that they can
  - query information from all domains
  - broadcast information to all domains

  using a common language of predicates
  (orthogonal to abstract domain internal representations)
- easy to enrich and refine

Numerical Domain for Compute-Throw-Overflow

Compute-through-overflow

```
Int16 x,y,z;
x = (Int16) ((Uint16) y + (Uint16) z);
```

Concrete semantics :

- modular casts of `y` and `z`
  map $[-32768, -1]$ to $[32768, 65535]$ (possible overflow)
- zero-extend each result to `int`
- perform an addition in `int`
- modular cast back to $[-32768, 32767]$ (possible overflow)

But it's a "feature" : *"avoids computations in unnecessary word lengths"*

Numerical Domain for Compute-Throw-Overflow

Compute-through-overflow

```
Int16 x,y,z;
x = (Int16) ((Uint16) y + (Uint16) z);
```

Analysis result :

- overflow alarms on negative input : expected
- very imprecise output on some inputs

  $(\texttt{Uint16})\ [-1, 0] = \{\, 0,\ 65535 \,\}$
  abstracted as $[0, 65535]$ in the interval domain

Numerical Domain for Compute-Throw-Overflow

Compute-through-overflow

```
Int16 x,y,z;
x = (Int16) (/*(Uint16)*/ y + /*(Uint16)*/ z);
```

Quite unsatisfactory solution !

Numerical Domain for Compute-Throw-Overflow

Compute-through-overflow

```
Int16 x,y,z;
x = (Int16) ((Uint16) y + (Uint16) z);
```

A better solution : add a new abstract domain

modular intervals : $[a, b] + c\mathbb{Z}$
reduced with standard intervals : $[\ell, h] \cap ([a, b] + c\mathbb{Z})$

example : $\mathtt{y}, \mathtt{z} \in [-1, 0]$

- `(Uint16)` $[-1, 0]$ abstracted as $[0, 65635] \cap ([-1, 0] + 65536\mathbb{Z})$
- $[0, 65635] \cap ([-1, 0] + 65536\mathbb{Z}) + [0, 65635] \cap ([-1, 0] + 65536\mathbb{Z})$
  $= [0, 131070] \cap ([-2, 0] + 65536\mathbb{Z})$ (no overflow)
- `(Int16)` $([0, 131070] \cap ([-2, 0] + 65536\mathbb{Z})) = [-2, 0]$

Predicate Domain For Bit-Level Float Manipulations

A complex example

```
int i;
double d,d1,d2;
unsigned *p1 = &d1, *p2 = &d2;
p1[0] = p2[0] = 0x43300000;
p1[1] = p2[1] = 0x80000000;
p2[1] ^= i;
d = d2 - d1;
```

???

Predicate Domain For Bit-Level Float Manipulations

A complex example

```
int i;
double d,d1,d2;
unsigned *p1 = &d1, *p2 = &d2;
p1[0] = p2[0] = 0x43300000;
p1[1] = p2[1] = 0x80000000;
p2[1] ^= i;
d = d2 - d1;
```

Concrete semantics :

- `(0x43300000, 0x80000000)` represents $2^{52} + 2^{31}$
- `(0x43300000, i^0x80000000)` represents $2^{52} + 2^{31} + \mathtt{i}$
- `d` is `i` converted to double

Predicate Domain For Bit-Level Float Manipulations

A complex example

```
int i;
double d,d1,d2;
unsigned *p1 = &d1, *p2 = &d2;
p1[0] = p2[0] = 0x43300000;
p1[1] = p2[1] = 0x80000000;
p2[1] ^= i;
d = d2 - d1;
```

Abstract analysis :

- materialization generates : `d`i `=` `dbl_of_int(p`i`[0],p`i`[1])`
- a special-purpose domain does the rest :
  - pattern-matching and rewrite rules on expressions
  - communicate with other domains (e.g., get intervals)
  - maintain simple predicates
    (e.g., $a = b$ ˆ 0x80000000, $a =$ `dbl_of_int`(0x43300000,b))

Symbolic LValue Domain

A costly example

```
unsigned short i, a[65536] ;
while (...) {
  i = get_index() ; /* returns [0,65535] */
  if (a[i] < 100) a[i]++ ;
}
```

Naive, imprecise analysis :

- `a[i]<100` is translated into
  `(a[0]<100)` $||$ `(a[1]<100)` $|| \cdots ||$ `(a[65535]<100)`
  $\Longrightarrow$ no information
- `a[i]++` causes an arithmetic overflow

Symbolic LValue Domain

A costly example

```
unsigned short i, a[65536];
while (...) {
  i = get_index(); /* returns [0,65535] */
  if (a[i] < 100) a[i]++;
}
```

Precision improvement :

- partition the `if` statement wrt. to the value of `i`

  equivalent to :

  ```
  if (i==0) { if (a[0] < 100) a[0]++; }
    ⋮
  if (i==65535) { if (a[65535] < 100) a[65535]++; }
  ```

- precise but costly (linear in array size)

Symbolic LValue Domain

A costly example

```
unsigned short i, a[65536];
while (...) {
  i = get_index(); /* returns [0,65535] */
  if (a[i] < 100) a[i]++;
}
```

Precise and efficient solution :

- domain to manipulate LValues symbolically :
  - introduce `symb = a[i]` when `a[i]` is first read
  - replace all read of `a[i]` with `symb`
    i.e., `if (symb < 100) a[i] = symb + 1;`
    ($\Rightarrow$ a[i] ≤ 100)
  - forget `symb` when `a` or `i` is modified
- use a single cell to represent $\bigcup_x \mathrm{a}[x]$ (folding)

$\Longrightarrow$ cost independent from the size of `a` !

Conclusion

Summary

Achievement

It is possible to build a static analyzer that is :

- efficient in time, memory, and development cost
- very precise on a given (infinite) class of programs

Recipe

- start from a simple analyzer
- while there are false alarms
    - find their cause, and either
        - tune analysis parameters, or
        - improve local abstraction heuristics, or
        - improve some existing domain, or
        - add some reduction between existing domains, or
        - add a new domain

On-Going Work

- scientific support of the industrialization by AbsInt
  (develop abstract domains required by new applications)
- analysis of parallel embedded real-time programs
- closed-loop analysis
  (software controller + environment model)
- automatic proof of the compilation to binary code
  (by translation validation)
- mechanized proof of the correction of Astrée
  (using theorem provers)

A Few Lessons Learned

- defining the concrete semantics is hard !

  end-users want you to analyze **their** idiom of C
  and are very picky about what is an error and what is **not**

- be prepared to analyze contrived program bits

  (hacks, language abuse, broken design rules, etc.)

- an adapted response : design small *ad hoc* domains
  (*semantical* hacks)

  but keep it sound !

  (easy to achieve thanks to Abstract Interpretation)

proving program correctness with semantic-based static analysis
. . . still much work to do