The Apron Library

Antoine Miné

CNRS, École normale supérieure

CEA Seminar
December the 10th, 2007

Outline

- Introduction
  - Main goals
  - Theoretical background
  - The APRON project
- The Apron Library
  - General description
  - Library API (data-types, abstract functions)
  - Abstract domain examples (intervals, octagons, polyhedra)
  - Linearization
- The Interproc Analyzer
  - Description
  - Demonstration

Introduction

Static Analysis

Goal : Static Analysis

Discover properties of a program statically and automatically.

Applications :

- compilation and optimisation, *e.g.* :
  - array bound check elimination
  - alias analysis
- verification and debugging, *e.g.* :
  - infer invariants
  - prove the absence of run-time errors
    (division by zero, overflow, invalid array access)
  - prove functional properties

Invariant Discovery Examples

Insertion Sort

```
for i=1 to 99 do
  p := T[i]; j := i+1;
  while j <= 100 and T[j] < p do
    T[j-1] := T[j]; j := j+1;
  end;
  T[j-1] := p;
end;
```

Invariant Discovery Examples

Interval analysis :

Insertion Sort

```
for i=1 to 99 do
  i ∈ [1,99]
  p := T[i]; j := i+1;
  i ∈ [1,99], j ∈ [2,100]
  while j <= 100 and T[j] < p do
    i ∈ [1,99], j ∈ [2,100]
    T[j-1] := T[j]; j := j+1;
    i ∈ [1,99], j ∈ [3,101]
  end;
  i ∈ [1,99], j ∈ [2,101]
  T[j-1] := p;
end;
```

$\Longrightarrow$ there is no out of bound array access

Invariant Discovery Examples

Linear relation analysis :

Insertion Sort

```
for i=1 to 99 do
  i ∈ [1,99]
  p := T[i]; j := i+1;
  i ∈ [1,99], j = i + 1
  while j <= 100 and T[j] < p do
    i ∈ [1,99], i + 1 ≤ j ≤ 100
    T[j-1] := T[j]; j := j+1;
    i ∈ [1,99], i + 2 ≤ j ≤ 101
  end;
  i ∈ [1,99], i + 1 ≤ j ≤ 101
  T[j-1] := p;
end;
```

$\Longrightarrow$ there is no out of bound array access

Theoretical Background

Abstract Interpretation : unifying theory of program semantics

Provide theoretical tools to design and compare static analyses that :

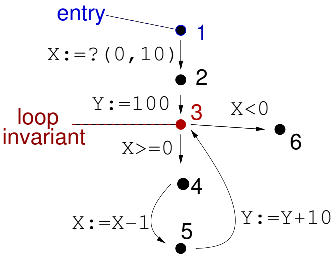
- always terminate
- are sound by construction (no behavior is omitted)
- are approximate (solve undecidability and efficiency issues)

Concrete Semantics

Concrete Semantics :

most precise mathematical expression of the program behavior

Example : from program to equation system



$$
\begin{cases}
\mathcal{X}_2 = \{\!|\, \mathtt{X} := ?(0,10) \,|\!\}(\mathcal{X}_1) \\
\mathcal{X}_3 = \{\!|\, \mathtt{Y} := 100 \,|\!\}(\mathcal{X}_2) \cup \\
\qquad\quad \{\!|\, \mathtt{Y} := \mathtt{Y} + 10 \,|\!\}(\mathcal{X}_5) \\
\mathcal{X}_4 = \{\!|\, \mathtt{X} \geq 0 \,|\!\}(\mathcal{X}_3) \\
\mathcal{X}_5 = \{\!|\, \mathtt{X} := \mathtt{X} - 1 \,|\!\}(\mathcal{X}_4) \\
\mathcal{X}_6 = \{\!|\, \mathtt{X} < 0 \,|\!\}(\mathcal{X}_3)
\end{cases}
$$

Where :

- $\mathcal{X}_i$ is a set of states, here $\mathcal{X}_i \in \mathcal{P}(\{\mathtt{X}, \mathtt{Y}\} \to \mathbb{Z}) = \mathcal{D}$
- $\{\!|\, \cdot \,|\!\}$ model the effect of tests and assignments
- the recursive system has a unique least solution (lfp)

Abstract Domains

Undecidability Issues :

- the concrete domain $\mathcal{D}$ is not computer-representable
- $\{\!|\cdot|\!\}$ and $\cup$ are not computable
- lfp is not computable

$\Longrightarrow$ we work in a **abstract domain $\mathcal{D}^\sharp$** instead

Definition of an abstract domain :

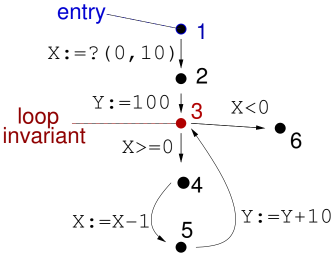
- $\mathcal{D}^\sharp$: a set of computer-representable elements
- a partial order $\sqsubseteq^\sharp$ on $\mathcal{D}^\sharp$
- $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$, monotonic, gives a meaning to abstract elements
- $\{\!|\cdot|\!\}^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ and $\cup^\sharp : (\mathcal{D}^\sharp)^2 \to \mathcal{D}^\sharp$ are abstract sound counterparts to $\{\!|\cdot|\!\}$ and $\cup$:

$$\begin{array}{ll} \forall \mathcal{X} \in \mathcal{D}^\sharp & (\gamma \circ \{\!|\cdot|\!\}^\sharp)(\mathcal{X}) \supseteq (\{\!|\cdot|\!\} \circ \gamma)(\mathcal{X}) \\ \forall \mathcal{X}, \mathcal{Y} \in \mathcal{D}^\sharp & \gamma(\mathcal{X} \cup^\sharp \mathcal{Y}) \supseteq \gamma(\mathcal{X}) \cup \gamma(\mathcal{Y}) \end{array}$$

- $\triangledown : (\mathcal{D}^\sharp)^2 \to \mathcal{D}^\sharp$ abstracts $\cup$ and enforces termination :
  $\forall \mathcal{Y}_i \in \mathcal{D}^\sharp,\ \mathcal{X}_0 \stackrel{\text{def}}{=} \mathcal{Y}_0,\ \mathcal{X}_{i+1} \stackrel{\text{def}}{=} \mathcal{X}_i \triangledown \mathcal{Y}_{i+1}$ converges finitely

Abstract Semantics

The concrete equation system is replaced with an abstract one :



$$\left\{\begin{array}{lll} \mathcal{X}_2^\sharp & \sqsupseteq^\sharp & \llbracket\, \mathtt{X} := ?(0,10) \,\rrbracket^\sharp(\mathcal{X}_1^\sharp) \\ \mathcal{X}_3^\sharp & \sqsupseteq^\sharp & \llbracket\, \mathtt{Y} := 100 \,\rrbracket^\sharp(\mathcal{X}_2^\sharp) \cup^\sharp \\ & & \llbracket\, \mathtt{Y} := \mathtt{Y} + 10 \,\rrbracket^\sharp(\mathcal{X}_5^\sharp) \\ \mathcal{X}_4^\sharp & \sqsupseteq^\sharp & \llbracket\, \mathtt{X} \geq 0 \,\rrbracket^\sharp(\mathcal{X}_3^\sharp) \\ \mathcal{X}_5^\sharp & \sqsupseteq^\sharp & \llbracket\, \mathtt{X} := \mathtt{X} - 1 \,\rrbracket^\sharp(\mathcal{X}_4^\sharp) \\ \mathcal{X}_6^\sharp & \sqsupseteq^\sharp & \llbracket\, \mathtt{X} < 0 \,\rrbracket^\sharp(\mathcal{X}_3^\sharp) \end{array}\right.$$

A solution can be found in finite time by iterations :

- start from $\mathcal{X}_1^{0\sharp} \stackrel{\text{def}}{=} \top^\sharp$, $\mathcal{X}_{k\neq 1}^{0\sharp} \stackrel{\text{def}}{=} \bot^\sharp$
- update all $\mathcal{X}_k^\sharp$ at each iteration : *e.g.* $\mathcal{X}_4^{i+1\sharp} \stackrel{\text{def}}{=} \llbracket\, \mathtt{X} \geq 0 \,\rrbracket^\sharp(\mathcal{X}_3^{i\sharp})$
- use widening at loop heads : *e.g.*
  $\mathcal{X}_3^{i+1\sharp} \stackrel{\text{def}}{=} \mathcal{X}_3^{i\sharp} \,\triangledown\, (\llbracket\, \mathtt{Y} := 100 \,\rrbracket^\sharp(\mathcal{X}_2^{\mathtt{i}\sharp}) \cup^\sharp \llbracket\, \mathtt{Y} := \mathtt{Y} + 10 \,\rrbracket^\sharp(\mathcal{X}_5^{\mathtt{i}\sharp}))$

It is a sound abstraction of the concrete semantics $\mathcal{X}_i$.

Numerical Abstract Domains

Important case :

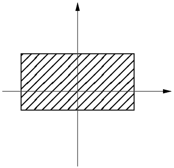
When $\mathcal{D}^\sharp$ abstract $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\texttt{Var} \rightarrow \mathbb{I})$ and

- $\texttt{Var}$ is a finite set of variables
- $\mathbb{I}$ is a numerical set, *e.g.*, $\mathbb{Z}$ or $\mathbb{R}$

Applications :

- discover numerical properties on program variables
- prove the absence of a large class of run-time errors
  (division by 0, overflow, out of bound array access, etc.)
- parametrize non-numerical analyses
  (pointer analysis, shape analysis)

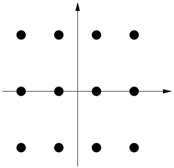
Some Existing Numerical Abstract Domains



Intervals

$X_i \in [a_i, b_i]$

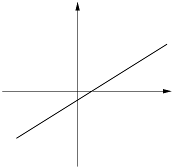
[Cousot-Cousot-76]



Simple Congruences

$X_i \equiv a_i \, [b_i]$

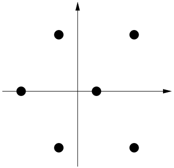
[Granger-89]



Linear Equalities

$\sum_i \alpha_i X_i = \beta$

[Karr-76]

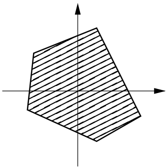


Linear Congruences

$\sum_i \alpha_i X_i \equiv \beta \, [\gamma]$

[Granger-91]

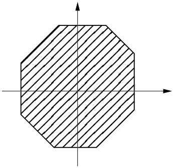
Some Existing Numerical Abstract Domains (cont.)



Polyhedra

$\sum_i \alpha_i X_i \geq \beta$

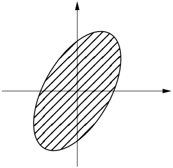
[Cousot-Halbwachs-78]



Octagons

$\pm X_i \pm X_j \leq \beta$

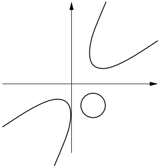
[Miné-01]



Ellipsoids

$\alpha X^2 + \beta Y^2 + \gamma XY \leq \delta$

[Feret-04]



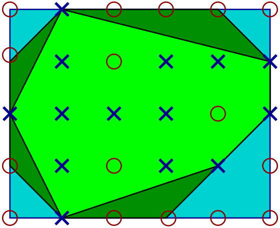
Varieties

$P(\vec{X}) = 0,\ P \in \mathbb{R}[\mathtt{Var}]$

[Sankaranarayanan-Sipma-Manna-04]

Precision vs. Cost Tradeoff

Example : three abstractions of the same set of points



Worst-case time cost per operation wrt. number of variables :

- polyhedra : exponential
- octagons : cubic
- intervals : linear

The Apron Project

Apron = Analyse de programmes numériques

Action Concertée Incitative "Sécurité et Informatique" (ACI SI)

October 2004 – October 2007

Partners

- École des Mines (CRI), coordinator : François Irigoin
- Verimag (Synchrone team)
- IRISA (VERTECS project)
- École normale supérieure
- École Polytechnique

Project Goals

- Theoretical side

  Advance the research on numerical abstract domains.

- Practical side

  Design and implement a library providing :

  - ready-to-use numerical abstract domains under a common API easing the design of new analysers
  - a platform for integration and comparison of new domains
  - teaching, demonstration, dissemination tools

  Steams from the fact that current implementations

  - have incompatible API
  - sometimes have very low-level API
  - sometimes lack important features (transfer functions)
  - often duplicate code

The Apron Library

Current Status of the Library

Available at : http://apron.cri.ensmp.fr/library/

- released under the LGPL licence
- 52 000 lines of C (v 0.9.8, not counting language bindings)
- main programmers : Bertrand Jeannet & Antoine Miné

Currently Available Domains

- polyhedra (NewPolka & PPL)
- linear equalities
- octagons
- intervals
- congruence equalities (PPL)
- reduced product of polyhedra and congruence equalities

Current Language Bindings : C, C++, OCaml

The implementation effort continues.

Implementation Choices

- C programming language for the kernel
- domain-neutral API and concrete data-types
- two-level API :
  - level 0 : abstracts $\mathbb{Z}^p \times \mathbb{R}^q$
  - level 1 : abstracts $(\mathtt{Var}_{\mathbb{Z}} \to \mathbb{Z}) \times (\mathtt{Var}_{\mathbb{R}} \to \mathbb{R})$
- functional and imperative transfer functions
- thread-safe
- exception mechanism (API errors, out-of-memory, etc.)
- user-definable options (trade-off precision/cost)
- (limited) object orientation (abstract data-types)

Implementation Choices

User–implementor contract :

- a domain must provide all sound transfer functions
- but the functions may be non-exact and non-optimal.

To add a new domain :

- only level 0 API to implement
- fallback functions provided
- ready-to-use convenience libraries
  - numbers (machine int, float, GMP, MPFR)
  - intervals
  - linearization
  - reduced product

$\Longrightarrow$ only a small core of functions actually needs to be implemented

API Types : Numbers

API Types :

concrete data-types used by the user to call the library
($\neq$ types used internally by domain implementations)

API types come with (scarce) support functions
(mainly constructors, destructors, printing)

- Scalar constants `ap_scalar_t`
  - arbitrary precision rationals (GMP)
  - IEEE doubles
  - $+\infty$, $-\infty$
  - (to come) arbitrary precision floats (MPFR)
- Coefficients `ap_coeff_t`
  - either a scalar
  - or an interval (with scalar bounds)

a coefficient represents a set of constant scalars

Level 0 Affine Expressions and Constraints

Level 0 : "variables" are dimension indices, starting from 0
p dimensions in $\mathbb{Z}$, followed by q dimensions in $\mathbb{R}$

- Affine expressions `ap_linexpr0_t`
  - $\ell \stackrel{\text{def}}{=} c + \sum_i c_i \mathtt{X_i}$
  - c and c_i are `ap_coeff_t` coefficients
  - either dense representation (array)
    or sparse representation (ordered list of pairs (i, c_i))
  - functions to modify, resize, permute, etc.
- Affine constraints `ap_lincons0_t`
  - equality constraints : $\ell = 0$
  - inequality constraints : $\ell \geq 0$ or $\ell > 0$
  - disequality constraints : $\ell \neq 0$
  - congruence constraints : $\ell \equiv 0\ [i]$

Non-scalar coefficients represent non-deterministic choices
$\Longrightarrow$ we actually represent sets of expressions and constraints

Level 0 Expressions and Constraints

- Expression trees `ap_texpr0_t`
  - variable indices and coefficients at the leaves
  - operators include : $+$, $-$, $\times$, $/$, mod, $\sqrt{\ }$
  - optional rounding to $\mathbb{Z}$ or IEEE floats of various size
  - optional rounding direction to $+\infty$, $-\infty$, 0, nearest, ?
  - operations : variable substitution, dimension reordering, etc.

- Constraints `ap_tcons0_t`
  - equality constraints : $t = 0$
  - inequality constraints : $t \geq 0$ or $t > 0$
  - disequality constraints : $t \neq 0$
  - congruence constraints : $t \equiv 0\ [i]$

As before, we actually represent expression and constraint sets.

Level 0 Generators and Arrays

- Generators `ap_generator0_t`
  - vertices : $\{\, \vec{v} \,\}$
  - lines : $\{\, \lambda\vec{v} \mid \lambda \in \mathbb{R} \,\}$
  - rays : $\{\, \lambda\vec{v} \mid \lambda \in \mathbb{R},\ \lambda \geq 0 \,\}$
  - modular lines : $\{\, \lambda\vec{v} \mid \lambda \in \mathbb{Z} \,\}$
  - modular rays : $\{\, \lambda\vec{v} \mid \lambda \in \mathbb{N} \,\}$

  where all coefficients in $\vec{v}$ must be scalar.

- Arrays `ap_`*xxx*`_array_t`
  - hold a size and a pointer to a C array
  - simplify memory management (allocation, resize, free)
  - arrays for intervals, (affine) constraints, and generators

Level 1 Variables and Environments

Level 1 : uses variable names instead of indices.

- Variable names `ap_var_t`
  - generic type : `void*`
  - totally ordered, by user-definable `compare` function
  - user-definable memory management (`copy`, `free`)
  - default implementation : C strings
- Environments `ap_environment_t`
  - ordered variable list, with integer or real type
    $\Longrightarrow$ defines a mapping names→indices
  - addition, removal, renaming of variables
    (the library maintains the mapping for us)
  - all level 1 types store an environment
    - environments are reference counted
    - the compatibility of environments is checked

Abstract Elements

- **Abstract elements `ap_abstract0_t`**

  Abstract data-type representing a set of points in $\mathbb{Z}^p \times \mathbb{R}^q$.

  Operations include :

  - construction : empty set, full set
  - set-theoretic : $\cup$, $\cap$
  - predicates : $=$, $\subseteq$, constraint saturation
  - property extraction : expression and variable bounds, conversion to constraints, generators, or box
  - transfer functions : constraint addition, (parallel) assignment or substitution, time elapse
  - dimension manipulation : addition, removal, forget, permutation, expansion, and folding
  - widening

  All functions take a manager as argument.

Managers

- Managers `ap_manager_t`

  Class-like structure for abstract elements.

  - each abstract domain library provides a manager factory
  - holds pointers to actual functions (virtual dispatch)
  - exposes user-definable parameters (*e.g.*, precision control)
  - exposes extra return values (*e.g.*, exactness flag)
  - provides static storage (thread-safety)
  - provides dynamic typing

Precision

Operations can be non-exact and non-optimal.

- For predicates :
  - `true` means definitely true
  - `false` means maybe true, maybe false
- For property extractions :
  the returned constraints, generators, intervals may be loose.
- When returning an abstract element :
  the returned element may not be an exact / best abstraction.

Some possible causes of imprecision :

- limited expressiveness (abstract domain, constraints, etc.)
- widening (inherently imprecise)
- not implemented (no algorithm, or too inefficient)
- conversion between user and internal data-type
- the user asked for a fast, imprecise answer

Precision Control and Feedback

Precision Control

Per-function domain-specific `algorithm` slider in the manager :

- 0 : default precision
- `MIN_INT`...`-1` : more efficiency at the cost of precision
- 1...`MAX_INT` : more precision at the cost of efficiency

Precision Feedback

Set in the manager after each function call :

- `flag_exact` (exact predicate, exact property, exact abstraction)
- `flag_best` (tightest property, best abstraction)

(if `flag_exact_wanted`, `flag_best_wanted` set by the user)

Fail-safe

- per-function user-definable timeout
- per-function user-definable maximum object size

Construction

Full and empty abstract elements

```
ap_abstract0_t* ap_abstract0_top
       (ap_manager_t* man, size_t p, size_t q);

ap_abstract0_t* ap_abstract0_bottom
       (ap_manager_t* man, size_t p, size_t q);
```

Returns a newly allocated abstract element :

- man indicates the instance of the library used
- p is the number of integer dimensions
- q is the number of real dimensions
- top returns an abstraction of $\mathbb{Z}^p \times \mathbb{R}^q$
- bottom returns an abstraction of $\emptyset$

We keep track of which dimensions are integers.
The result of all transfer functions is intersected with $\mathbb{Z}^p \times \mathbb{R}^q$.

Set-Theoretic Binary Operations

Example : binary join

```
ap_abstract0_t* ap_abstract0_join
        (ap_manager_t* man, bool destructive,
         ap_abstract0_t* a1, ap_abstract0_t* a2);
```

Computes `r` such that $\gamma(\mathtt{r}) \supseteq \gamma(\mathtt{a1}) \cup \gamma(\mathtt{a2})$

- `destructive` indicates an imperative version
  - if `false`, returns a newly allocated abstract element
  - if `true`, recycles the memory for `a1`

  `a2` is always preserved
- `flag_exact` indicates whether $\gamma(\mathtt{r}) = \gamma(\mathtt{a1}) \cup \gamma(\mathtt{a2})$
- `flag_best` indicates whether
  $\gamma(\mathtt{r}) = \min_{\subseteq} \{\ \gamma(\mathtt{x}) \mid \mathtt{x} \in \mathcal{D}^\sharp,\ \gamma(\mathtt{x}) \supseteq \gamma(\mathtt{a1}) \cup \gamma(\mathtt{a2})\ \}$

`ap_abstract0_meet` is similar, but for $\cap$.

Set-Theoretic N-Aray Operations

Example : n-aray join

```
ap_abstract0_t* ap_abstract0_join_array
    (ap_manager_t* man, ap_abstract0_t** tab, size_t size);
```

Returns a newly allocated abstract element `r` such that :

$$\gamma(\texttt{r}) \quad \supseteq \bigcup_{\texttt{0}\leq\texttt{i}<\texttt{size}} \gamma(\texttt{tab[i]})$$

`ap_abstract0_meet_array` is similar, but for $\cap$.

Note : why do we need `_array` versions ?

- may be more efficient than several `ap_abstract0_join`
- different meaning for `flag_exact` and `flag_best`

Adding Constraints

Example : adding arbitrary constraints

```
ap_abstract0_t* ap_abstract0_meet_tcons_array
        (ap_manager_t* man, bool destructive,
         ap_abstract0_t* a, ap_tcons0_array_t* c);
```

Definitions

- semantics of a deterministic constraint : $[\![c]\!] : \mathcal{D} \rightarrow \{\mathtt{t}, \mathtt{f}\}$
- each `c[i]` represents a set $\beta(\mathtt{c[i]})$ of deterministic constraints

`meet_tcons_array` computes an abstract element `r` such that :

$$
\begin{aligned}
\gamma(\mathtt{r}) &\supseteq \{\, \vec{x} \in \gamma(\mathtt{a}) \mid \forall \mathtt{i},\ \exists c \in \beta(\mathtt{c[i]}),\ [\![c]\!](\vec{x}) = \mathtt{true} \,\} \\
&= \bigcup_{\forall \mathtt{i},\, c_i \in \beta(\mathtt{c[i]})} \{\, \vec{x} \in \gamma(\mathtt{a}) \mid \forall \mathtt{i},\ [\![c_{\mathtt{i}}]\!](\vec{x}) = \mathtt{true} \,\}
\end{aligned}
$$

It models the semantics of tests.

Constraint Saturation

Example : testing an arbitrary constraint

```
bool ap_abstract0_sat_tcons
  (ap_manager_t* man, ap_abstract0_t* a, ap_tcons0_t* c);
```

Returns `true` if it can prove that :

$$\forall \vec{x} \in \gamma(\mathtt{a}),\ \forall c \in \beta(\mathtt{c}),\ [\![c]\!](\vec{x}) = \mathtt{true}$$

If it returns `false` then :

- if `flag_exact=true`, then
  $$\exists \vec{x} \in \gamma(\mathtt{a}), \exists c \in \beta(\mathtt{c}),\ [\![c]\!](\vec{x}) = \mathtt{false}$$
- otherwise, don't know

Note : saturation of a constraint we just added may return `false`

- due to over-approximation
- or due to non-determinism

Assignments

Example : assigning an arbitrary expression

```
ap_abstract0_t* ap_abstract0_assign_texpr
        (ap_manager_t* man, bool destructive,
         ap_abstract0_t* a, ap_dim_t dim,
         ap_texpr0_t* e, ap_abstract0_t* dst);
```

Semantics of an expression : $[\![\texttt{e}]\!] : \mathcal{D} \to \mathcal{P}(\mathbb{R})$

`assign_texpr` computes an abstract element `r` such that :

$$\gamma(\texttt{r}) \supseteq \{\ x[\texttt{v}_{\texttt{dim}} \mapsto v] \mid \vec{x} \in \gamma(\texttt{a}),\ v \in [\![\texttt{e}]\!](\vec{x})\ \} \cap \gamma(\texttt{dst})$$

`dst` (optional) is used to refine the result according to some *a priori* knowledge of the result.
(often more precise in the abstract than calling `meet` afterwards)

Substitutions

Example : substituting an arbitrary expression

```
ap_abstract0_t* ap_abstract0_substitute_texpr
        (ap_manager_t* man, bool destructive,
         ap_abstract0_t* a, ap_dim_t dim,
         ap_texpr0_t* e, ap_abstract0_t* dst);
```

`substitute_texpr` computes an abstract element `r` such that :

$$\gamma(\mathtt{r}) \supseteq \{\, \vec{x} \mid \exists v \in [\![\mathtt{e}]\!](\vec{x}),\ \vec{x}[\mathtt{v}_{\mathtt{dim}} \mapsto v] \in \gamma(\mathtt{a}) \,\} \cap \gamma(\mathtt{dst})$$

(intuitively, if $\gamma(\mathrm{a}) \models \mathrm{c}$ then $\gamma(\mathtt{r}) \models \mathrm{c}[\mathtt{v}_{\mathtt{dim}}/\mathrm{e}]$)

It models the backwards semantics of assignments.

Parallel Assignments and Substitutions

Example : parallel assignment of arbitrary expressions

```
ap_abstract0_t* ap_abstract0_assign_texpr_array
        (ap_manager_t* man, bool destructive,
         ap_abstract0_t* a, ap_dim_t* dim,
         ap_texpr0_t** e, size_t size,
         ap_abstract0_t* dst);
```

`assign_texpr_array` computes an abstract element `r` such that :
$\gamma(\mathtt{r}) \supseteq \{\, \vec{x}[\mathtt{v}_{\mathtt{dim[i]}} \mapsto v_i] \mid \vec{x} \in \gamma(\mathtt{a}),\, \forall i,\, v_i \in [\![\mathtt{e[i]}]\!](\vec{x}) \,\} \cap \gamma(\mathtt{dst})$

All assignments take place at the same time.

Could be emulated using `assign_texpr`
at the cost of using temporary variables.

Expand and Fold

Expand and fold

```
ap_abstract0_t* ap_abstract0_expand
    (ap_manager_t* man, bool destructive,
     ap_abstract0_t* a, ap_dim_t dim, size_t n);
ap_abstract0_t* ap_abstract0_fold
    (ap_manager_t* man, bool destructive,
     ap_abstract0_t* a, ap_dim_t dim*, size_t n);
```

expand adds n copies of $\mathtt{v}_{\mathtt{dim}}$ to a :

$$\gamma(\mathtt{r}) \supseteq \{\ (\vec{x}, v_1, \ldots, v_{\mathtt{n}}) \mid \vec{x} \in \gamma(\mathtt{a}),\ \forall i,\ \vec{x}[\mathtt{v}_{\mathtt{dim}} \mapsto v_i] \in \gamma(\mathtt{a})\ \}$$

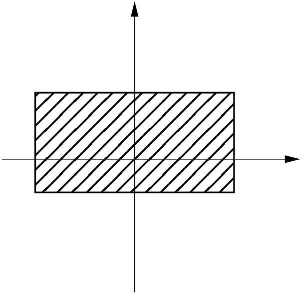
fold merges n variables into $\mathtt{v}_{\mathtt{dim[0]}}$:

$$\gamma(\mathtt{r}) \supseteq \bigcup_{0 \leq \mathtt{i} < \mathtt{n}} \{\ \mathrm{proj}_{\mathtt{i}}(\vec{x}) \mid \vec{x} \in \gamma(\mathtt{a})\ \}$$

where $\mathrm{proj}_{\mathtt{i}}$ maps dimension dim[i] to dim[0] and projects out dimensions dim[k], $\mathtt{k} \neq \mathtt{i}$.

Models arrays and weak updates [Gopan-DiMaio-Dor-Reps-Sagiv04].

The Interval Domain



Constraints of the form $\mathtt{v}_i \in [a_i, b_i]$.

The Interval Domain

Abstract representation :

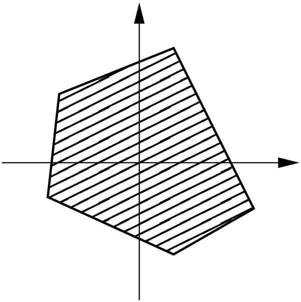
Associate two bounds for each variable, can be :

- GMP rationals, enriched with $\pm\infty$, or
- IEEE double

Abstract transfer functions :

Uses interval arithmetics.
IEEE double bounds are rounded correctly.

The Polyhedron Domain



Constraints of the form $\sum_i \alpha_i \mathrm{v}_i \geq \beta$.

The Polyhedron Domain : Representation

Abstract representation :

We use the double description method :

- conjunction of affine constraints $\bigwedge_j \left(\sum_i \alpha_{ij} \mathrm{v}_i \geq \beta_j\right)$
- sum of generators

$$\{ \sum_i \lambda_i \vec{v}_i + \sum_j \mu_j \vec{r}_j \mid \lambda_i, \mu_j \geq 0, \sum_i \lambda_i = 1 \}$$

where $\alpha_{ij}, \beta_j, \vec{v}_i, \vec{r}_j$ are GMP rationals.

Optimization : equalities and lines are encoded specially.

The Polyhedron Domain : Transfer Functions

Abstract transfer functions :

The main algorithm is the Chernikova–LeVerge algorithm :

- switches from one representation to the other
- minimizes both representations
- tests for emptiness

Most transfer functions are easy using the right representation :

- intersection (constraints), convex hull (generators)
- affine assignments, substitutions, constraint addition
- classical widening [Halbwachs-79]
- etc.

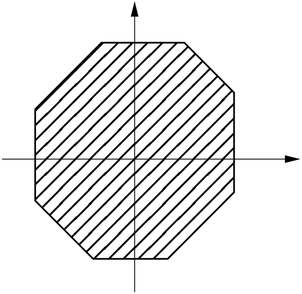
Optimization : equalities and lines use Gauss elimination.

The Polyhedron Domain : Extra Features

Advanced features include :

- strict constraints
  (encoded through an extra slack variable)
- approximation
  rotate or remove constraints to reduce the size of coefficients
  (activated through `algorithm`)
- integer tightening
  tighten existing constraints involving integer variables
  (polynomial, non-complete algorithm)
  (activated through `algorithm`)
- non-deterministic and non-linear transfer functions
  expressions are linearized into $[a_0, b_0] + \sum_i c_i \mathtt{v}_i$
  which can be treated directly

The Octagon Domain



Constraints of the form $\pm \mathtt{v}_i \pm \mathtt{v}_j \leq c$.

The Octagon Domain : Representation

Abstract representation :

A set of constraints is represented as a square matrix :

- $\mathbf{m}_{2i,2j}$ is an upper bound for $\mathtt{v}_j - \mathtt{v}_i$
- $\mathbf{m}_{2i+1,2j}$ is an upper bound for $\mathtt{v}_j + \mathtt{v}_i$
- $\mathbf{m}_{2i,2j+1}$ is an upper bound for $-\mathtt{v}_j - \mathtt{v}_i$
- $\mathbf{m}_{2i+1,2j+1}$ is an upper bound for $-\mathtt{v}_j + \mathtt{v}_i$

Upper bounds may be encoded using either :

- GMP integers, enriched with $+\infty$
- GMP rationals, enriched with $+\infty$
- IEEE double or long double

Optimization : only the lower-left triangle is actually stored.

The Octagon Domain : Transfer Functions

Abstract transfer functions :

The main algorithm is the Floyd-Warshall algorithm :

- shortest-path closure
- propagates and tightens all constraints
- tests for emptiness

Most transfer functions are then easy :

- intersection : point-wise min
- join : point-wise max on closed matrices
- assignments, substitutions of expressions of the form $\pm \mathrm{v}_i + c$
- adding constraints of the form $\pm \mathrm{v}_i \pm \mathrm{v}_j \leq c$
- etc.

The Octagon Domain : Extra Features

Advanced features include :

- non-deterministic affine transfer functions

  e.g. assignment $\mathtt{v}_k \leftarrow [a_0, b_0] + \sum_i [a_i, b_i]\mathtt{v}_i$

  - extract bounds $[v_i^-, v_i^+]$ for each variable $\mathtt{v}_i$
  - evaluate $[a_0, b_0] + \sum_i [a_i, b_i] \times [v_i^-, v_i^+]$ in interval arithmetics
    $\Longrightarrow$ new bounds for $\mathtt{v}_k$
  - for each $j \neq k$, $\epsilon = \pm 1$, evaluate
    $[a_0, b_0] + \sum_{i \neq j} [a_i, b_i] \times [v_i^-, v_i^+] + [a_j + \epsilon, b_j + \epsilon] \times [v_j^-, v_j^+]$
    $\Longrightarrow$ new bounds for $\mathtt{v}_k + \epsilon \mathtt{v}_j$

  (polynomial algorithm, not best abstraction)

- non-linear transfer functions
  expressions are linearized into $[a_0, b_0] + \sum_i [a_i, b_i]\mathtt{v}_i$
  which can be treated as above.

Linearization : Principle

Core Idea : abstract expressions

Replace e with e' such that : $\forall \vec{x} \in \gamma(\mathtt{a})$, $[\![\mathrm{e}']\!](\vec{x}) \supseteq [\![\mathrm{e}]\!](\vec{x})$, then :

- $\{\!|\, \mathtt{v} \leftarrow \mathrm{e}' \,|\!\}^\sharp(\mathtt{a})$ is a sound abstraction of $\{\!|\, \mathtt{v} \leftarrow \mathrm{e} \,|\!\}(\gamma(\mathtt{a}))$
- $\{\!|\, \mathrm{e}' \geq 0 \,|\!\}^\sharp(\mathtt{a})$ is a sound abstraction of $\{\!|\, \mathrm{e} \geq 0 \,|\!\}(\gamma(\mathtt{a}))$
- etc.

We choose expressions of the form $\mathrm{e}' \stackrel{\text{def}}{=} [a_0, b_0] + \sum_i [a_i, b_i]\mathtt{v}_i$:

- affine expressions are easy to manipulate
- non-deterministic intervals offer abstraction opportunities
- such expressions can be swallowed by many domains :
  - the octagon domain
  - the polyhedron domain, after further abstraction into $[a_0, b_0] + \sum_i c_i \mathtt{v}_i$

Linearization : Algorithm

Interval affine forms is enriched with the following algebra :

- point-wise interval addition and subtraction
- point-wise interval multiplication or division by an interval
- intervalization, *i.e.*, evaluation into a single interval (requires bounds on all variables)

We proceed by structural induction on the expression [Miné-04] :

- real $+$ and $-$ map directly to affine form addition, subtraction
- real $\times$ and $/$ first intervalize one argument
- real $\sqrt{\ }$ perform interval arithmetics on the intervalized argument
- rounding and casting introduce rounding errors by
  - enlarging variable coefficients with a relative error, and/or
  - adding absolute error intervals

The Interproc Analyzer

The Interproc Analyzer

Interproc : showcase analyzer for Apron

- analyzer for a toy language
- infers numerical properties using Apron
- written in OCaml
- authors : Gaël Lalire, Mathias Argoud, and Bertrand Jeannet
- available under LGPL at `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`
- can also be used on-line

Language

Support for :

- `while` loops and tests
- recursive procedures and functions
- integers and reals variables
- all operators from `ap_texpr0_t`, including float rounding

But :

- no arrays
- no dynamic memory allocation
- no I/O, except `random`

Principle of the Analysis

The program is converted into an equation system that is solved by a generic solver that implements :

- parametrization by the choice of an abstract domain
- increasing iterations with (delayed) widening
- decreasing iterations
- iteration ordering [Bourdoncle-93]
- guided analysis [Gopan-Reps-07]
- forward-backward combination

Demonstration

`http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`