A method to infer inductive numeric invariants inspired from Constraint Programming

Dagstuhl Seminar 14351

Experimental Talk

Antoine Miné

(inspiration from Charlotte Truchet & Sriram Sankaranarayanan)

CNRS & École normale supérieure
Paris, France

24–29 August 2014

Introduction

Constraint Programming $\underset{\text{Sriram}}{\overset{\text{Charlotte}}{\rightleftarrows}}$ Abstract Interpretation

Two fields: with different goals, tools, communities

- Constraint Programming
- Abstract Interpretation

yet there are similarities we can exploit!

- **Previous work:** with Marie Pelleau, Charlotte Truchet, Frédéric Benhamou
  use abstract domains in a constraint programming solver

- **Today:** original idea by Sriram Sankaranarayanan
  maybe we can adapt constraint programming algorithms
  to infer post-fixpoints of semantic functions
  instead of solutions of constraints

Overview

- Reminders on Constraint Programming
- Inductive numeric invariant inference
  - Motivation: invariants and inductive invariants
  - **Algorithm**
- Very preliminary experiments

Constraint Programming Primer

Constraint Programming

Goal: solve hard, combinatorial problems

- express the problem using constraints
  declarative language
  use conjunctions of first-order formulas with arithmetic ($\mathbb{R}$, $\mathbb{Z}$, enumerations)
- solve the constraints
  using generic methods

Here, we consider only continuous constraints (not discrete ones)

Constraint satisfaction problem

Definition: Constraint Satisfaction Problem (CSP)

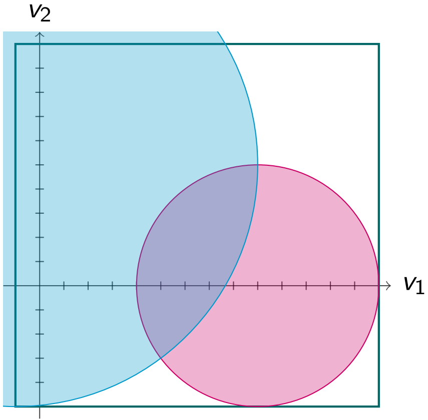
- $\mathcal{V} \stackrel{\text{def}}{=} \{ v_1, \ldots, v_n \}$: set of variables
- $\mathcal{D} \stackrel{\text{def}}{=} D_1 \times \cdots \times D_n$: a set of initial domains
  $\forall i$: $D_i \subseteq \mathbb{R}$ and D_i is bounded
- $\mathcal{C} \stackrel{\text{def}}{=} \{ C_1, \ldots, C_m \}$ set of constraints on $\mathcal{V}$

CSP solution:

- $\mathcal{S} \stackrel{\text{def}}{=} \{ \vec{x} \in \mathcal{D} \mid \forall i{:}\, \vec{x} \models C_i \}$

(also possible: look for a single solution instead of all solutions)

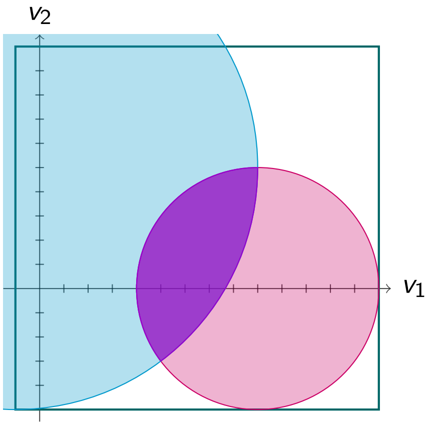
Constraint satisfaction problem example



- $\mathcal{V} \overset{\text{def}}{=} (v_1, v_2)$
- $D_1 \overset{\text{def}}{=} [-1, 14]$
  $D_2 \overset{\text{def}}{=} [-5, 10]$
- $C_1 : (v_1 - 9)^2 + v_2^2 \leq 25$
  $C_2 : (v_1 + 1)^2 + (v_2 - 5)^2 \leq 100$

(slide from Marie Pelleau & Charlotte Truchet)

Constraint satisfaction problem example



- $\mathcal{V} \stackrel{\text{def}}{=} (v_1, v_2)$
- $D_1 \stackrel{\text{def}}{=} [-1, 14]$
  $D_2 \stackrel{\text{def}}{=} [-5, 10]$
- $C_1 : (v_1 - 9)^2 + v_2^2 \leq 25$
  $C_2 : (v_1 + 1)^2 + (v_2 - 5)^2 \leq 100$

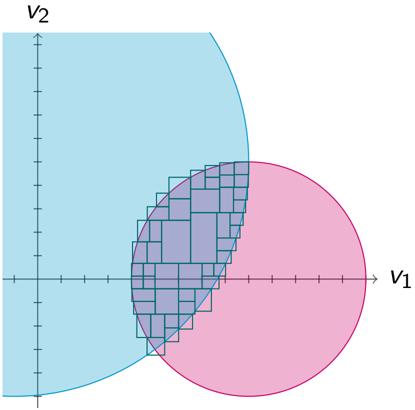
(slide from Marie Pelleau & Charlotte Truchet)

Constraint satisfaction problem solution

We would like to enumerate $\mathcal{S} \subseteq \mathbb{R}^f rm{-}e$, but this is impossible!
$\Longrightarrow$ instead, we cover $\mathcal{S}$ tightly with a finite set of boxes

$\mathcal{S}^\sharp$ set of boxes such that:

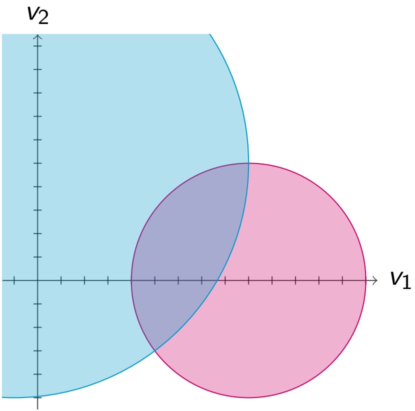
- $\mathcal{S} \subseteq \cup \mathcal{S}^\sharp$
- $\forall B \in \mathcal{S}^\sharp$:
  - either $B \subseteq \mathcal{S}$
  - or $\text{size}(B) \leq \epsilon$ and $B \cap \mathcal{S} \neq \emptyset$



(on discrete problems, solvers eventually enumerate $\mathcal{S}$)

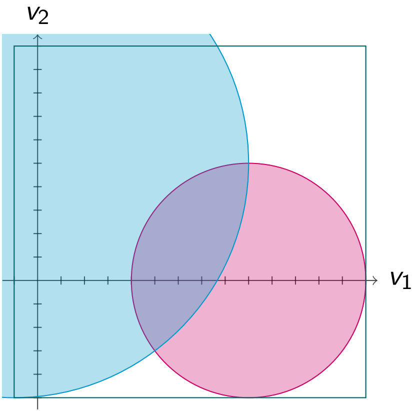
Continuous constraint programming algorithm

- list of boxes todo $:= \{ D \}$
- while todo is not empty



Continuous constraint programming algorithm

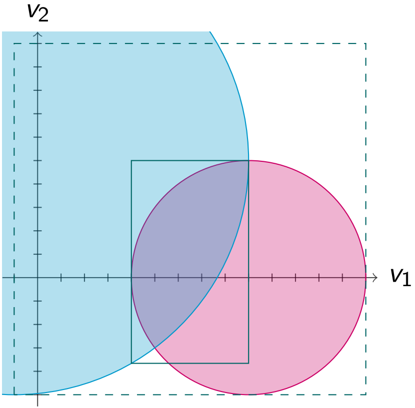
- list of boxes todo := $\{ D \}$
- while todo is not empty
  - pop a box from todo



Continuous constraint programming algorithm

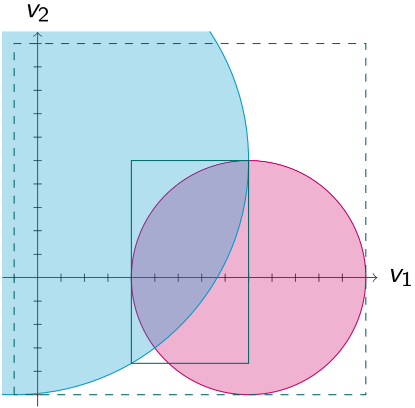
- list of boxes todo := { D }
- while todo is not empty
  - pop a box from todo
  - shrink it using the constraints

    consistency
    $\simeq$ interval test transfer function



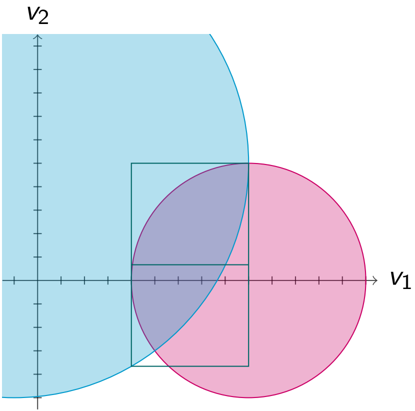
Continuous constraint programming algorithm

- list of boxes todo := { D }
- while todo is not empty
  - pop a box from todo
  - shrink it using the constraints
    consistency
    $\simeq$ interval test transfer function
  - if empty, continue
  - if small or contains only solutions
    shift it the solution list



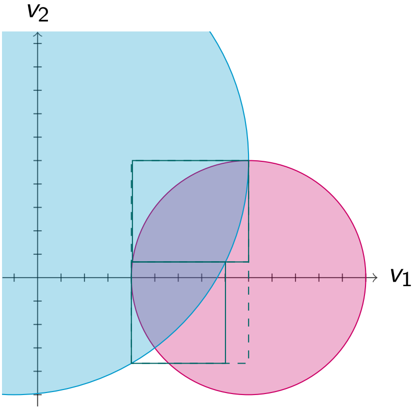
Continuous constraint programming algorithm

- list of boxes todo := $\{ D \}$
- while todo is not empty
  - pop a box from todo
  - shrink it using the constraints
    consistency
    $\simeq$ interval test transfer function
  - if empty, continue
  - if small or contains only solutions
    shift it the solution list
  - else
    split the box
    push the pieces into todo



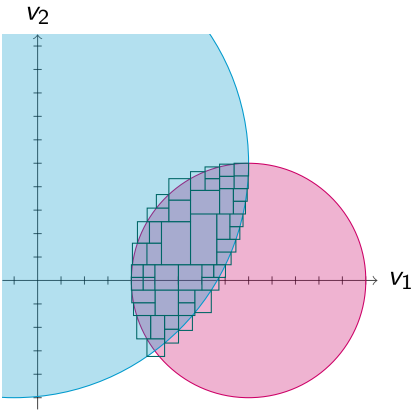
Continuous constraint programming algorithm

- list of boxes todo := { D }
- while todo is not empty
  - pop a box from todo
  - shrink it using the constraints
    consistency
    $\simeq$ interval test transfer function
  - if empty, continue
  - if small or contains only solutions
    shift it the solution list
  - else
    split the box
    push the pieces into todo



Continuous constraint programming algorithm

- list of boxes todo := { D }
- while todo is not empty
  - pop a box from todo
  - shrink it using the constraints
    consistency
    $\simeq$ interval test transfer function
  - if empty, continue
  - if small or contains only solutions
    shift it the solution list
  - else
    split the box
    push the pieces into todo



Inductive Numeric Invariant Inference

Motivating example

program

```
x := input [-1,1]
y := input [-1,1]
while true do
   x' := 0.7 * (x + y)
   y' := 0.7 * (x - y)
   x := x'; y := y'
done
```

Goal: prove that $x, y \in [-2, 2]$ is a loop invariant

Program semantics:

- initial values of (x, y): $I \stackrel{\text{def}}{=} [-1, 1] \times [-1, 1]$
- loop effect on (x, y):

$$F : \mathcal{P}(\mathbb{R}^2) \to \mathcal{P}(\mathbb{R}^2)$$
$$F(X) \stackrel{\text{def}}{=} \{\, (0.7(x + y), 0.7(x - y)) \mid (x, y) \in X \,\}$$

Invariants and inductive invariants

Given:

- $F : \mathcal{P}(\mathbb{R}^n) \xrightarrow{\cup} \mathcal{P}(\mathbb{R}^n)$ a $\cup-$morphism
- $I \subseteq \mathbb{R}^n$: an initial set

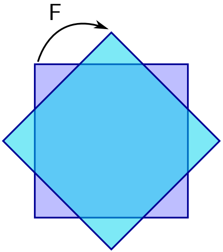
Then:

- $S \subseteq \mathbb{R}^n$ is an inductive invariant if $I \subseteq S \wedge F(S) \subseteq S$
  (invariant + proof)
- $\text{lfp}_I\, F$ is the best inductive invariant
  (Tarski's Theorem $\Longrightarrow$ inductive invariants exist)
- any $S \supseteq \text{lfp}_I\, F$ is an invariant
  (invariants are not always inductive invariants: $F(S) \not\subseteq S$)

Motivating example

program

```
x := input [-1,1]
y := input [-1,1]
while true do
   x' := 0.7 * (x + y)
   y' := 0.7 * (x - y)
   x := x'; y := y'
done
```



$$I \stackrel{\text{def}}{=} [-1,1] \times [-1,1]$$
$$F(X) \stackrel{\text{def}}{=} \{\, (0.7(x+y), 0.7(x-y)) \mid (x,y) \in X \,\}$$

- $G = [-2,2] \times [-2,2]$ is an invariant
  all executions satisfy $(x,y) \in G$ at loop head
- $G = [-2,2] \times [-2,2]$ is not an inductive invariant
  $F(G) \not\subseteq G$

in fact, no box is inductive!

Problem statement

Goal: infer an inductive invariant from an invariant

Given:

- $F : \mathcal{P}(\mathbb{R}^n) \xrightarrow{\cup} \mathcal{P}(\mathbb{R}^n)$ a $\cup-$morphism
- $I \subseteq \mathbb{R}^n$: an initial set
- $G \subseteq \mathbb{R}^n$: a goal invariant

find S such that:

- $I \subseteq S \wedge F(S) \subseteq S$ (S is an inductive invariant)
- $S \subseteq G$ (S implies the invariant G)

Abstract domain

Issue: we cannot compute in $\mathcal{P}(\mathbb{R}^n)$

$\Longrightarrow$ we reason in a computable abstract domain instead

Abstract domain: $\mathcal{D}^\sharp \subseteq \mathcal{P}(\mathbb{R}^n)$

- selected subsets of $\mathbb{R}^n$
- e.g.: boxes $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \{\, \Pi_{i=1}^n [a_i, b_i] \mid \forall i: a_i, b_i \in \mathbb{R} \cup \{-\infty, +\infty\} \,\}$

Abstraction closure: $\rho : \mathcal{P}(\mathbb{R}^n) \xrightarrow{\subseteq} \mathcal{D}^\sharp$

- soundness condition: $\forall S: S \subseteq \rho(S)$
- monotonicity: $\forall S, R: S \subseteq R \implies \rho(S) \subseteq \rho(R)$
- idempotence: $\rho \circ \rho = \rho$

we simplify traditional AI by assimilating abstract elements to their representation
we have a Galois connection $\mathcal{P}(\mathbb{R}^n) \underset{\rho}{\overset{\text{id}}{\leftrightarrows}} \mathcal{D}^\sharp$

Abstract operators

Abstract operator: $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$

- soundness: $\forall S \in \mathcal{D}^\sharp$: $F(S) \subseteq F^\sharp(S)$
- optional optimality: $F^\sharp = \rho \circ F$

Abstract initial set: $I^\sharp \in \mathcal{D}^\sharp$

- soundness: $I \subseteq I^\sharp$
- optional optimality: $I^\sharp = \rho(I)$

Abstract goal: $G^\sharp \in \mathcal{D}^\sharp$
$(G = G^\sharp \in \mathcal{D}^\sharp)$

Motivating example

program

```
x := input [-1,1]
y := input [-1,1]
while true do
   x' := 0.7 * (x + y)
   y' := 0.7 * (x - y)
   x := x'; y := y'
done
```

Abstract semantics on boxes:

$F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$

$F^\sharp([l_x, h_x], [l_y, h_y]) \stackrel{\text{def}}{=}$
$([0.7(l_x + l_y), 0.7(h_x + h_y)], [0.7(l_x - h_y), 0.7(h_x - l_y)])$

$I^\sharp \stackrel{\text{def}}{=} [-1, 1] \times [-1, 1]$

Traditional Abstract Interpretation approach:

- find an abstract post-fixpoint of $F^\sharp$: $F^\sharp(X^\sharp) \subseteq X^\sharp$, $I^\sharp \subseteq X^\sharp$
- by iterating $F^\sharp$ from $\emptyset$ with acceleration $\triangledown$, $\vartriangle$, $\tilde{\vartriangle}$, $\boxminus$

Issue: in our example $F^\sharp$, in the box abstract domain, the only abstract post-fixpoint greater than $I^\sharp$ is $\top$

Traditional solution:

use a stronger abstract domain $\mathcal{D}^\sharp$ (relational domain)
$\Longrightarrow$ high design cost

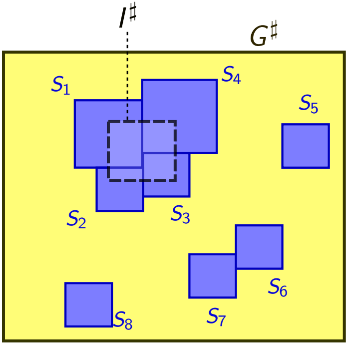
Note: traditional disjunctive completion and trace partitioning will not help as $F^\sharp$ has no control and no join

Abstract problem statement

We ask that the goal invariant G is represented in $\mathcal{D}^\sharp$
yet $\mathcal{D}^\sharp$ may not be able to represent any inductive invariant
$\Longrightarrow$ use a set of abstract elements

Goal: infer $\mathcal{S} = \{S_1, \ldots, S_n\} \subseteq \mathcal{D}^\sharp$ such that:

- $i \neq j \implies \text{vol}(S_i \cap S_j) = 0$
  no overlap
- $I^\sharp \subseteq \cup_i S_i$
  $\mathcal{S}$ covers the initial set I
- $\forall i$: $F^\sharp(S_i) \subseteq \cup_i S_i$
  implies $F(\cup_i S_i) \subseteq \cup_i S_i$, i.e., $\mathcal{S}$ is inductive
- $\forall i$: $S_i \subseteq G^\sharp$
  $\mathcal{S}$ implies the goal invariant



$\Longrightarrow$ $\cup_i S_i$ is an inductive invariant implying $G^\sharp$

Solving algorithm: overview

Algorithm: inspired by constraint programming

Given: $F^\sharp$, $I^\sharp$, $G^\sharp$ such that $I^\sharp \subseteq G^\sharp$

The algorithm maintains a "soup" $\mathcal{S} \in \mathcal{P}_{\text{finite}}(\mathcal{D}^\sharp)$

Algorithm

- start with $\mathcal{S} \stackrel{\text{def}}{=} \{G^\sharp\}$
- while $\exists k$: $F^\sharp(S_k) \not\subseteq \cup_i S_i$
  - choose $S_k \in \mathcal{S}$
  - either keep S_k, discard S_k, or split S_k

- at each step $\cup_i S_i$ can only decrease
- at each step $I^\sharp \subseteq \cup_i S_i$
- we stop when $\forall k$: $F^\sharp(S_k) \subseteq \cup_i S_i$

Solving algorithm: element classification

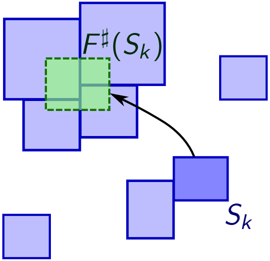
Each $S_k \in \mathcal{S}$ can be classified according to three criteria:

- whether $F^\sharp(S_k) \subseteq \cup_i S_i$ or not
  (S_k is benign: it does not impede $\mathcal{S}$'s inductiveness)
- whether $\exists i$: $F^\sharp(S_i) \cap S_k \neq \emptyset$ or not
  (S_k is useful, for some S_i to be benign)
- whether $I^\sharp \cap S_k \neq \emptyset$ or not
  (S_k is necessary, for $\mathcal{S}$ to be invariant)

Discarding a useful S_k
can change the fact that some $S_{i \neq k}$ is benign
(different from regular CP)

Dilemma:
shrinking $F(X)$ makes $F(X) \subseteq X$ more likely
but this requires shrinking X, which makes $F(X) \subseteq X$ less likely!

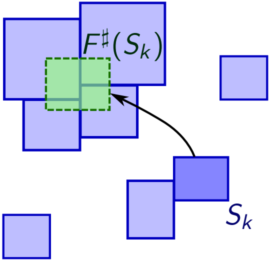
Solving algorithm: cases



case 1: $F^\sharp(S_k) \subseteq \cup_i S_i$

S_k is benign

Solving algorithm: cases

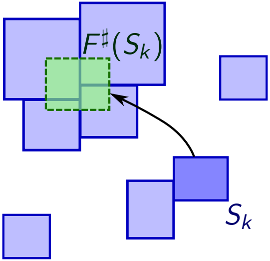


case 1: $F^\sharp(S_k) \subseteq \cup_i S_i$

subcase 1a: $\forall i\colon S_k \cap F^\sharp(S_i) = \emptyset \quad \wedge \quad I^\sharp \cap S_k = \emptyset$

S_k is benign but useless

$\Longrightarrow$ **discard** S_k



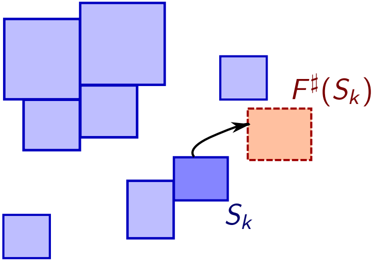
case 1: $F^\sharp(S_k) \subseteq \cup_i S_i$

subcase 1b: $\exists i$: $S_k \cap F^\sharp(S_i) \neq \emptyset \quad \vee \quad I^\sharp \cap S_k \neq \emptyset$

S_k is benign and useful (or necessary)

$\Longrightarrow$ **keep** S_k

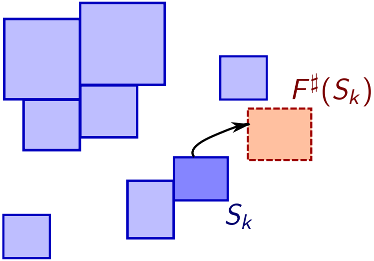
Solving algorithm: cases



case 2: $F^\sharp(S_k) \cap (\cup_i S_i) = \emptyset$

S_k can never become benign

Solving algorithm: cases

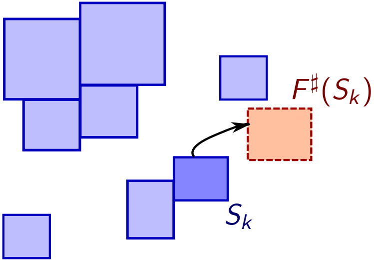


case 2: $F^\sharp(S_k) \cap (\cup_i\, S_i) = \emptyset$
subcase 2a: $S_k \cap I^\sharp = \emptyset$

S_k is not necessary

$\Longrightarrow$ **discard** S_k

Solving algorithm: cases

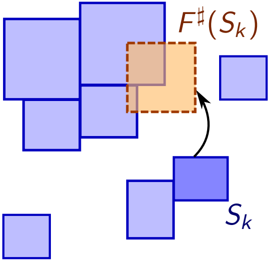


<u>case 2:</u> $F^\sharp(S_k) \cap (\cup_i S_i) = \emptyset$
<u>subcase 2b:</u> $S_k \cap I^\sharp \neq \emptyset$

S_k is necessary but can never become benign!

$\Longrightarrow$ **abort**, we fail to find an inductive invariant

Solving algorithm: cases

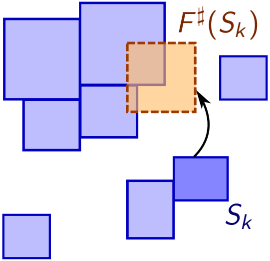


<u>case 3:</u> $F^\sharp(S_k) \not\subseteq \cup_i S_i \wedge F^\sharp(S_k) \cap (\cup_i S_i) \neq \emptyset$

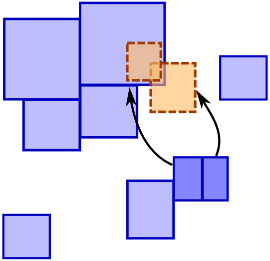
S_k is partially benign

$\Longrightarrow$ two possible choices

Solving algorithm: cases



$\Longrightarrow$

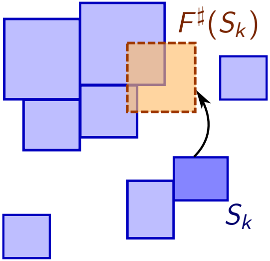


case 3: $F^\sharp(S_k) \not\subseteq \cup_i S_i \wedge F^\sharp(S_k) \cap (\cup_i S_i) \neq \emptyset$

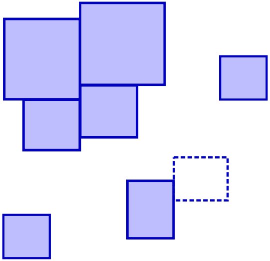
$\Longrightarrow$ either **split** S_k into S_k^1 and S_k^2, such that $S_k^1 \cup S_k^2 = S_k$

(often $F^\sharp(S_k^1) \cup F^\sharp(S_k^2) \subset F^\sharp(S_k) \Longrightarrow$ progress)

Solving algorithm: cases



$\Longrightarrow$

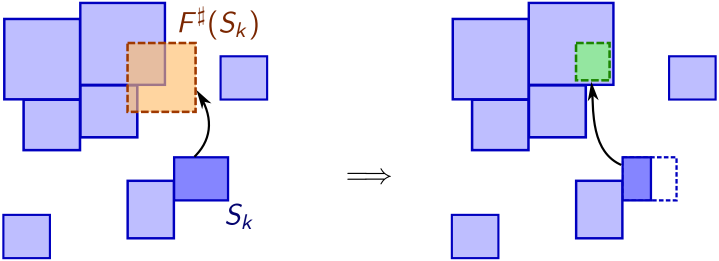


case 3: $F^\sharp(S_k) \not\subseteq \cup_i S_i \wedge F^\sharp(S_k) \cap (\cup_i S_i) \neq \emptyset \wedge S_k \cap I^\sharp = \emptyset$

$\Longrightarrow$ or **discard** S_k

(except if S_k is necessary)

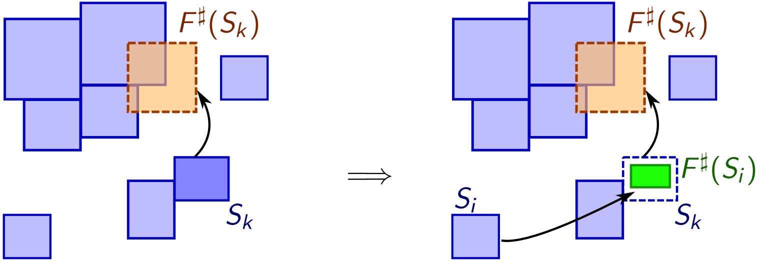
Solving algorithm: cases



<u>case 3:</u> $F^\sharp(S_k) \not\subseteq \cup_i\, S_i \wedge F^\sharp(S_k) \cap (\cup_i\, S_i) \neq \emptyset$

Splitting followed by discarding achieves shrinking S_k

Solving algorithm: cases

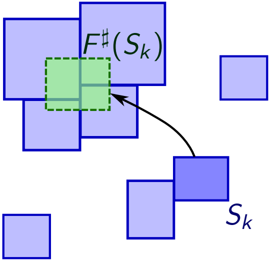


case 3: $F^\sharp(S_k) \not\subseteq \cup_i S_i \wedge F^\sharp(S_k) \cap (\cup_i S_i) \neq \emptyset$

Discarding a useful S_k can cause some $S_{\ell \neq k}$ to be become **not benign**

$F(S_\ell) \subseteq \cup_i S_i$ but $F(S_\ell) \not\subseteq \cup_{i \neq k} S_i$

Solving algorithm: termination

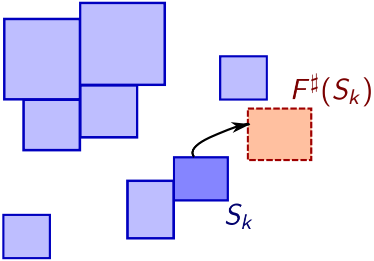


Terminate when:

- either $\forall k$: $F^\sharp(S_k) \subseteq \cup_i \; S_i$

  $\Longrightarrow$ **success**: $\cup_i S_i$ is an inductive invariant

Solving algorithm: termination

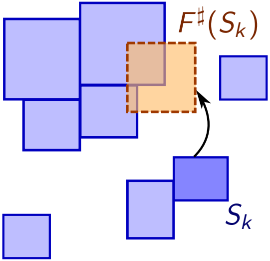


Terminate when:

- or $\exists k$: $F^\sharp(S_k) \cap (\cup_i S_i) = \emptyset \quad \wedge \quad S_k \cap I^\sharp \neq \emptyset$

  $\Longrightarrow$ **failure**: cannot find an inductive invariant

Solving algorithm: termination



Terminate when:

- reaching a split limit

  $\Longrightarrow$ **failure**: cannot find an inductive invariant

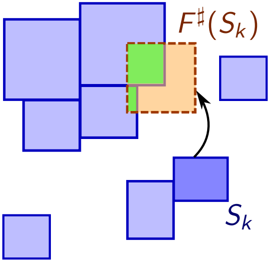
Solving algorithm: choices

Choosing S_k

Pick the element with the least coverage

$$\text{coverage}(S_k) = \frac{\sum_i \text{vol}(F^\sharp(S_k) \cap S_i)}{\text{vol}(F^\sharp(S_k))} \in [0,1]$$

(quantifies how benign an element is)



$$\text{coverage} = \frac{\text{green}}{\text{green+red}}$$

Our ultimate goal is to have $\forall k$: $\text{coverage}(S_k) = 1$

(indeed $\forall k$: $\text{coverage}(S_k) = 1 \iff F^\sharp(S_k) \subseteq \cup_i S_i$)

Solving algorithm: choices

Choosing whether to split or to discard S_k

for the case $F^\sharp(S_k) \not\subseteq \cup_i S_i \quad \wedge \quad F^\sharp(S_k) \cap (\cup_i S_i) \neq \emptyset$

- if $\text{coverage}(S_k) < \epsilon$ and $S_k \cap I^\sharp = \emptyset$, discard S_k
  unlikely that $\text{coverage}(S_k)$ will ever be 1
- if $\text{vol}(S_k) < \epsilon$ and $S_k \cap I^\sharp = \emptyset$, discard S_k
  splitting threshold
- if $\text{vol}(S_k) < \epsilon$ and $S_k \cap I^\sharp \neq \emptyset$, keep S_k intact
  we must ensure $I^\sharp \subseteq \cup_i S_i$ at all time
  $\Longrightarrow$ avoid discarding necessary elements
- otherwise, split S_k
  increases coverage

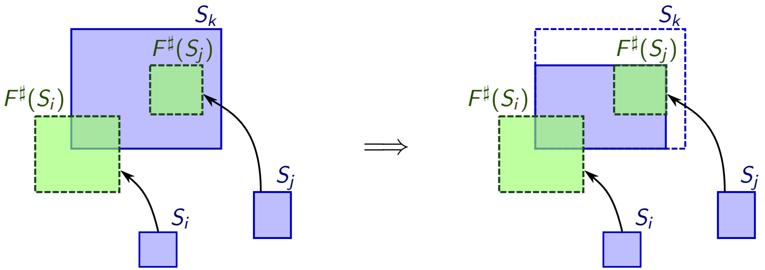
other heuristics are possible: usefulness, backtracking, etc.

Solving algorithm: tightening

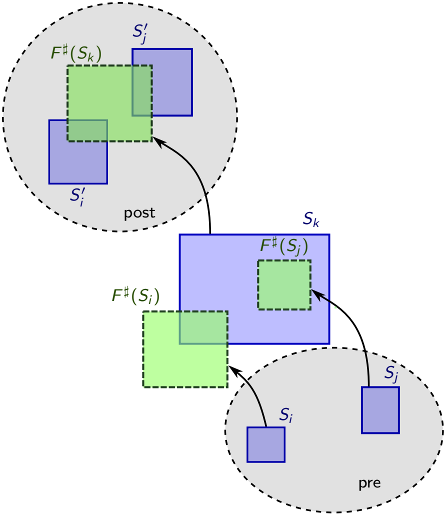
Idea: tighten S_k without removing important points

$$S_k \mapsto \rho \left(\cup_i \left(S_k \cap F^\sharp(S_i) \right) \quad \cup \quad \left(S_k \cap I^\sharp \right) \right)$$

- keep $S_k \cap F^\sharp(S_i)$ unchanged for all i
- keep $S_k \cap I^\sharp$ unchanged
- reduce $\mathrm{vol}(S_k) \implies$ increase coverage(S_k)
- $\implies$ useful after a split (similar to CP consistency)



Solving algorithm: data-structures



Maintain for each $S_k \in \mathcal{S}$:

- $F^\sharp(S_k)$
- coverage(S_k)
- pre$(S_k) \stackrel{\text{def}}{=} \{\, i \mid S_k \cap F^\sharp(S_i) \neq \emptyset \,\}$
- post$(S_k) \stackrel{\text{def}}{=} \{\, i \mid F^\sharp(S_k) \cap S_i \neq \emptyset \,\}$

$\Longrightarrow$ loop iteration cost in $\mathcal{O}(|\text{pre}(S_k)| + |\text{post}(S_k)|) \ll |\mathcal{S}|$

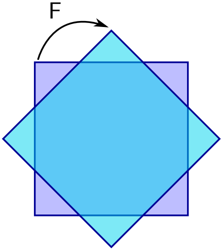
keep $\mathcal{S}$ sorted by increasing coverage(S_k)

Preliminary experiments

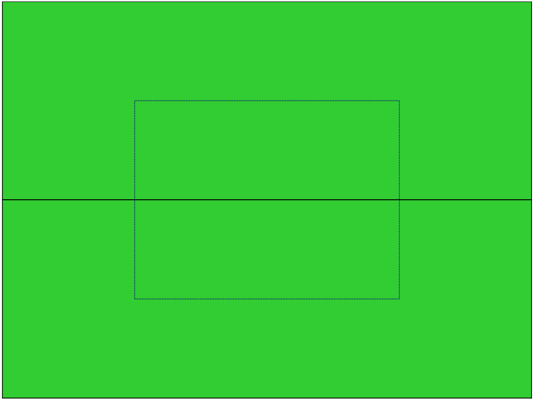
Example 1

program

```
x := input [-1,1]
y := input [-1,1]
while true do
   x' := 0.7 * (x + y)
   y' := 0.7 * (x - y)
   x := x'; y := y'
done
```

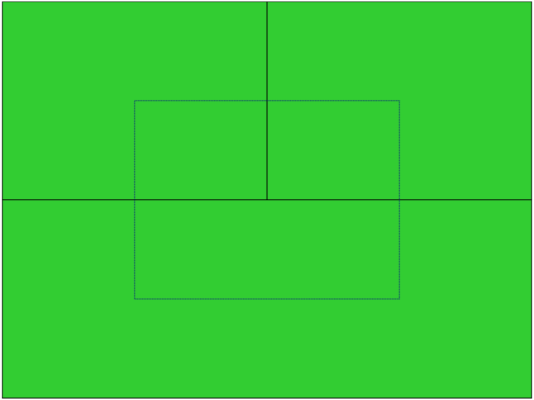


Example 1



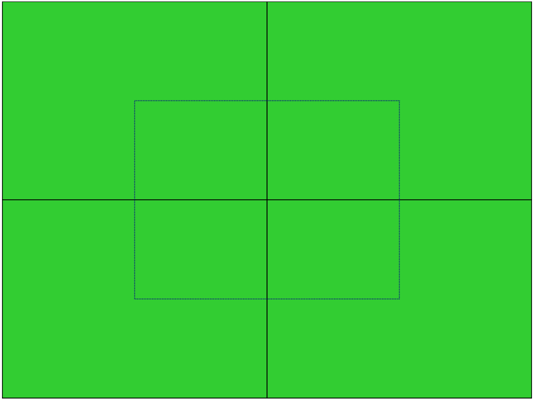
iterate 1

Example 1



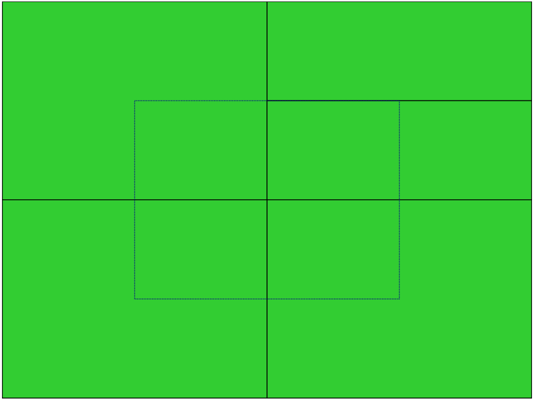
iterate 2

Example 1



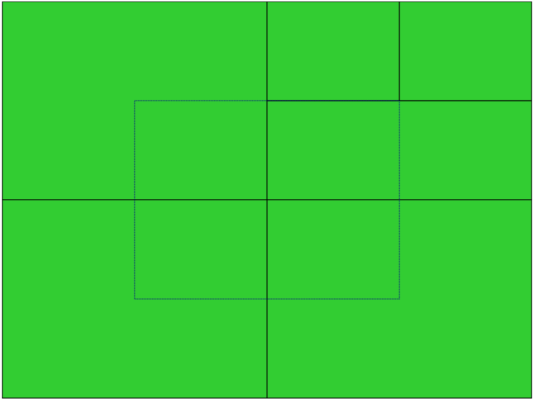
iterate 3

Example 1



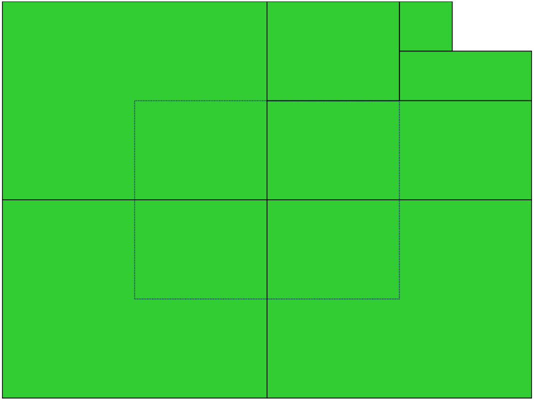
iterate 4

Example 1



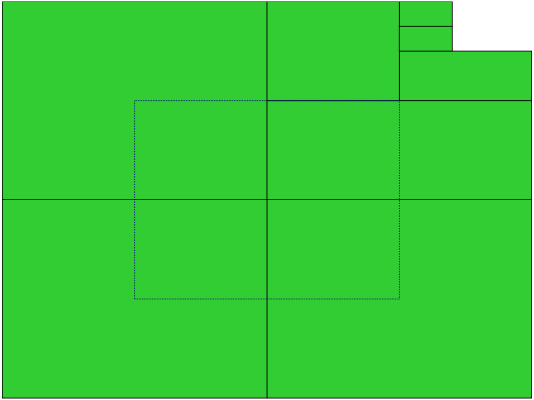
iterate 5

Example 1



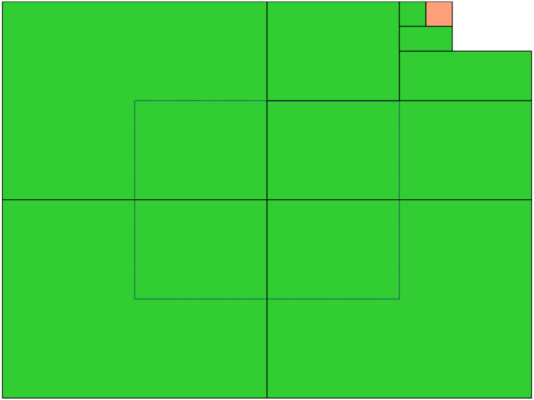
iterate 6

Example 1



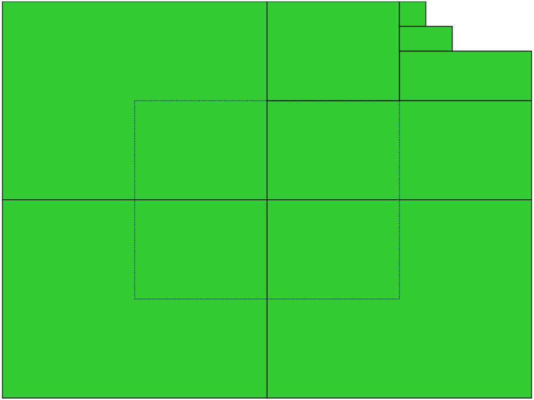
iterate 7

Example 1



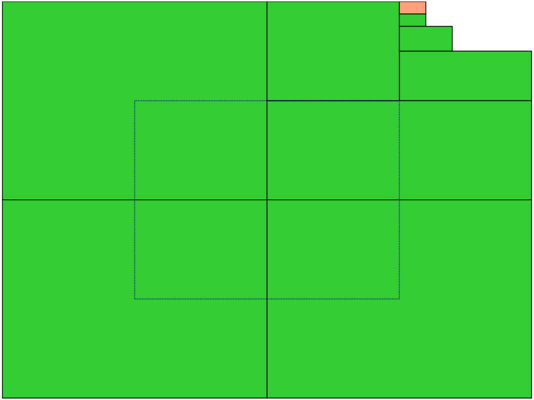
iterate 8

Example 1



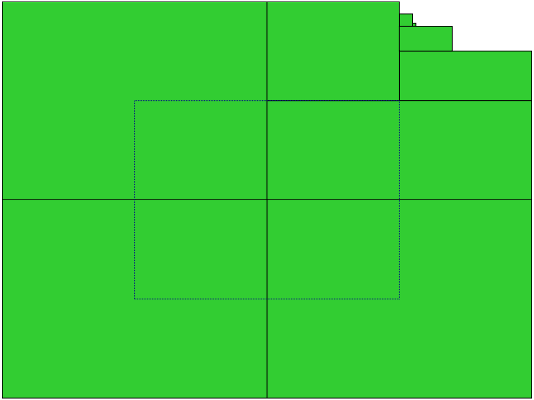
iterate 9

Example 1



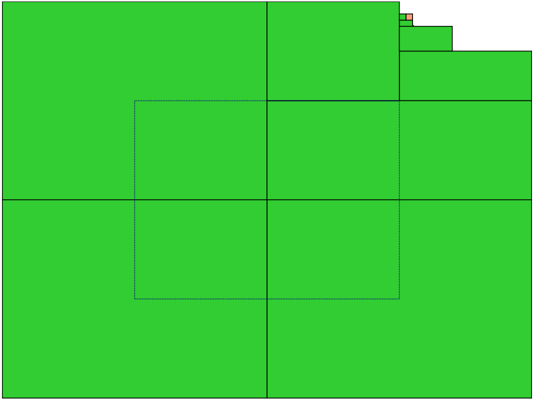
iterate 10

Example 1



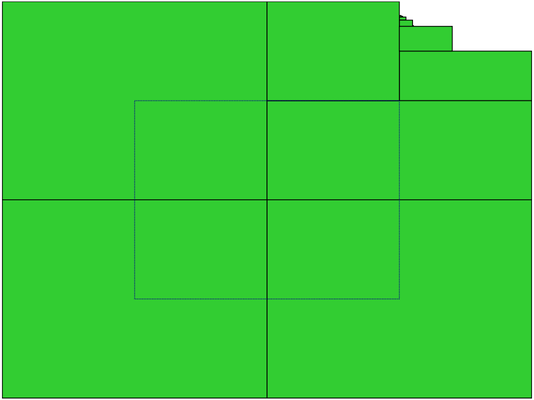
iterate 20

Example 1



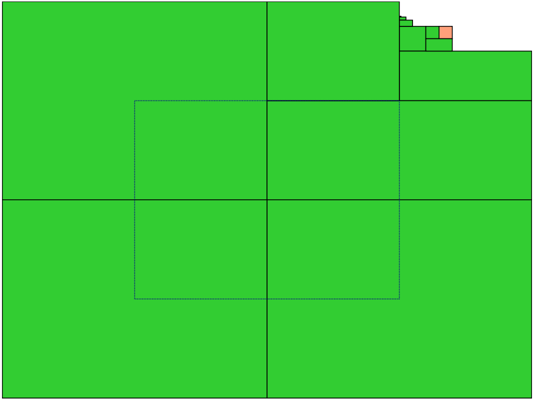
iterate 30

Example 1



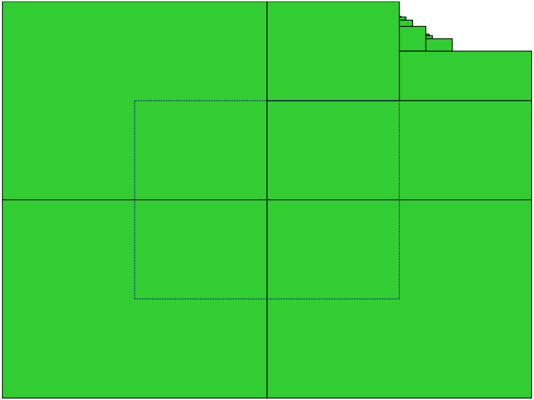
iterate 40

Example 1



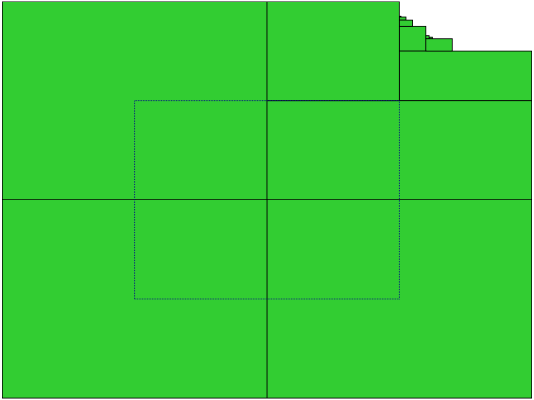
iterate 50

Example 1



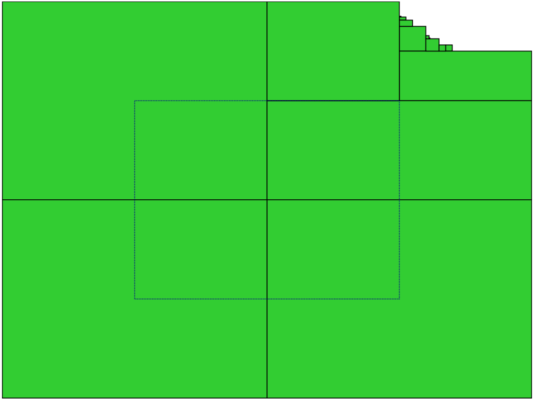
iterate 60

Example 1



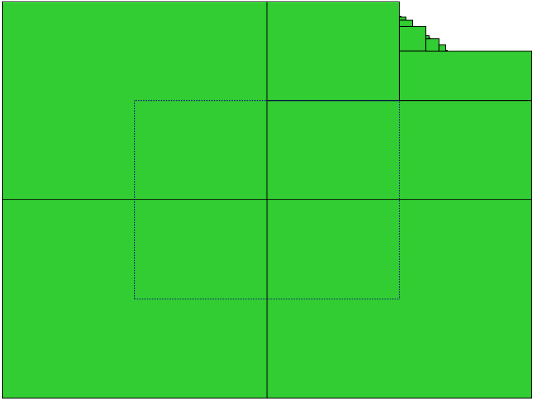
iterate 70

Example 1



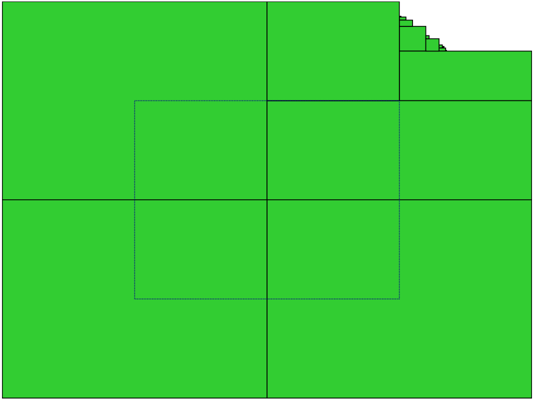
iterate 80

Example 1



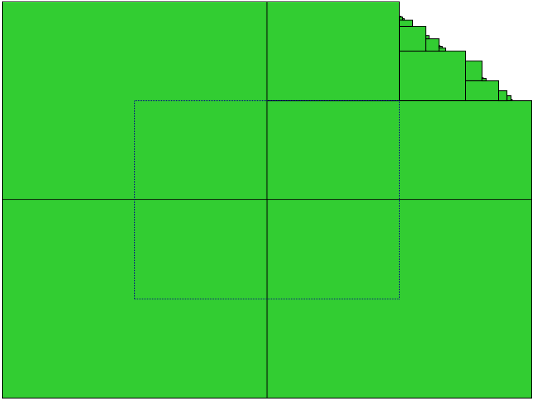
iterate 90

Example 1



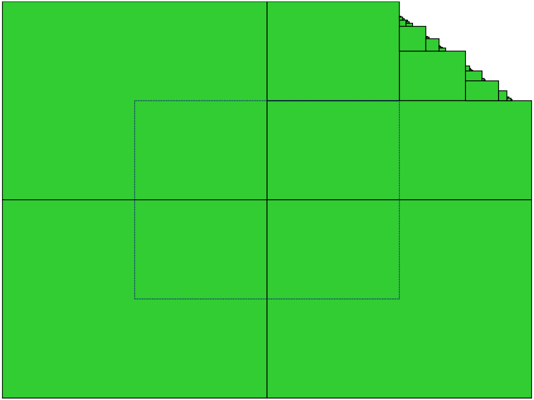
iterate 100

Example 1



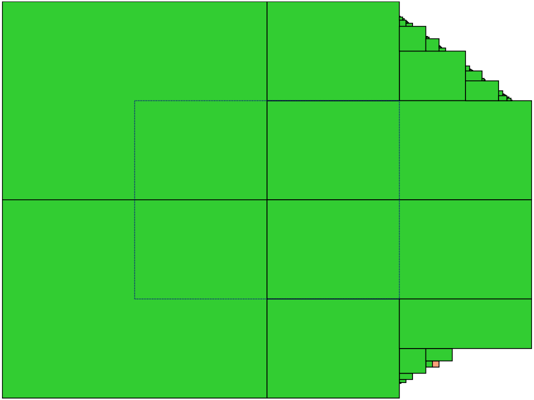
iterate 200

Example 1



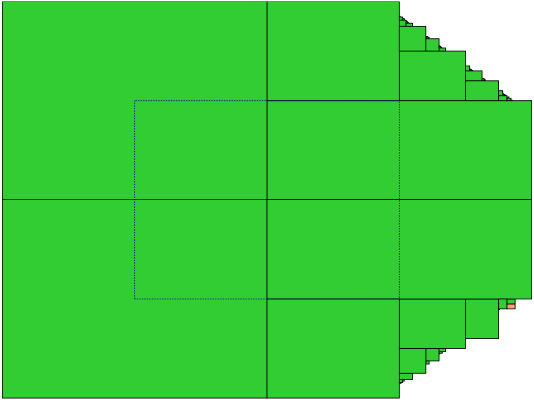
iterate 300

Example 1



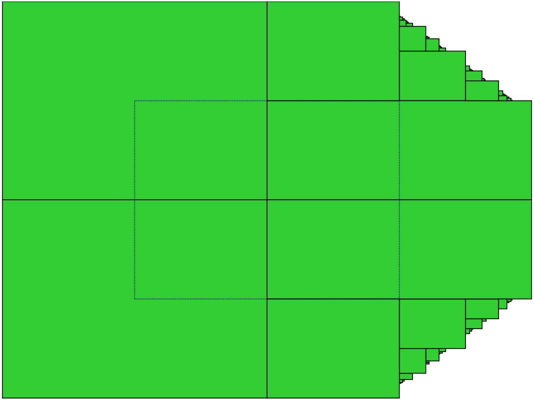
iterate 400

Example 1



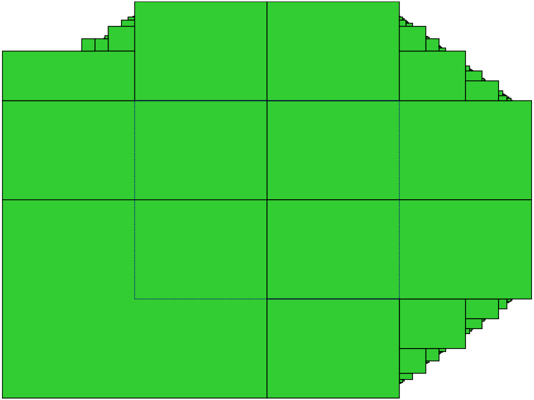
iterate 500

Example 1



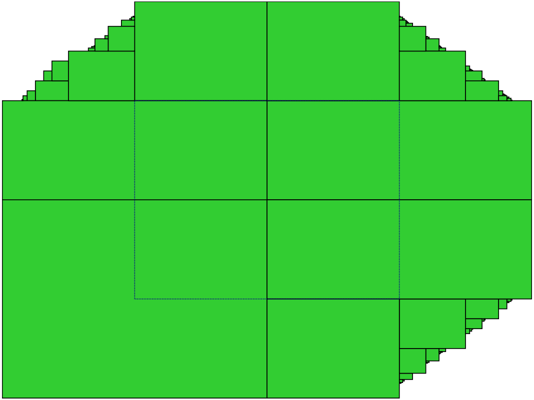
iterate 600

Example 1



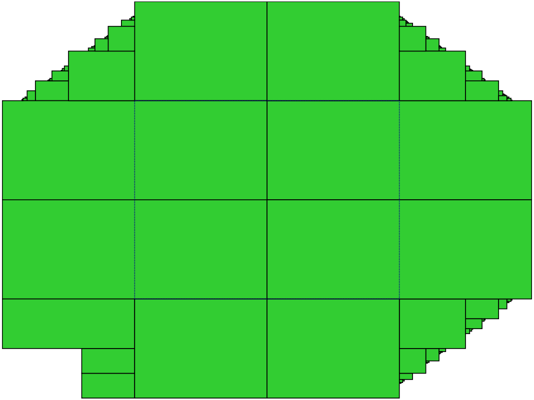
iterate 700

Example 1



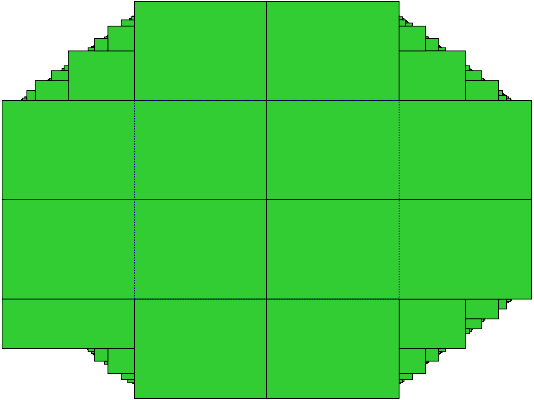
iterate 800

Example 1



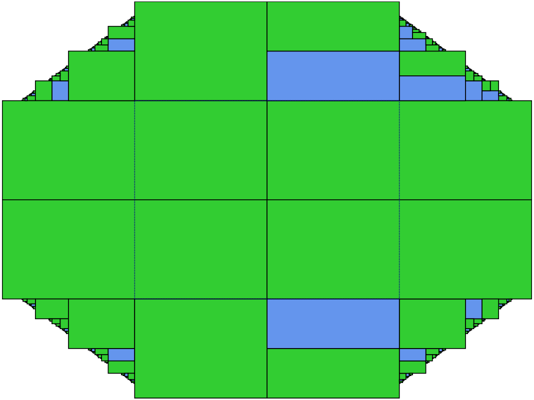
iterate 900

Example 1



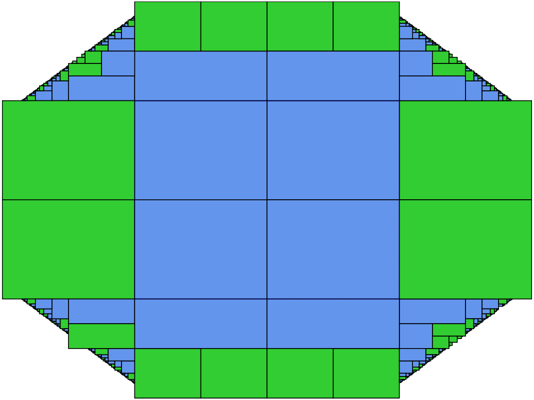
iterate 1000

Example 1



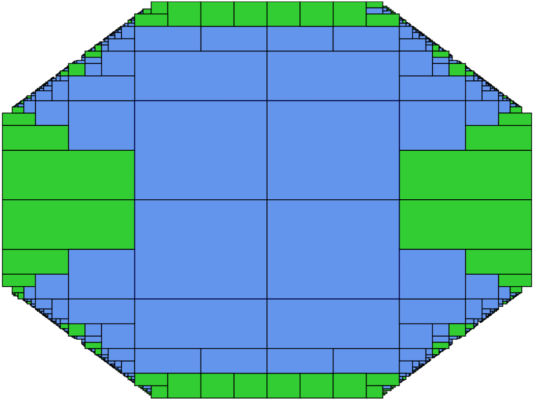
iterate 2000

Example 1



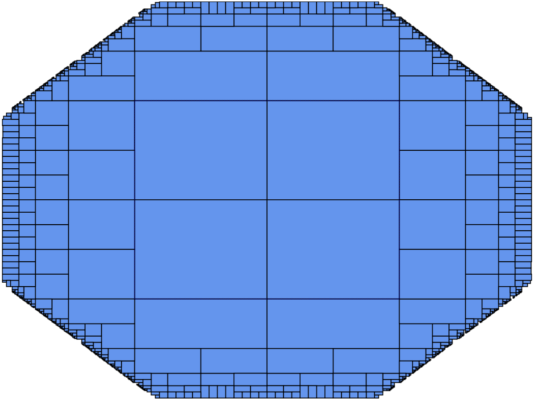
iterate 3000

Example 1



iterate 4000

Example 1



final iterate: 4730

Example 2

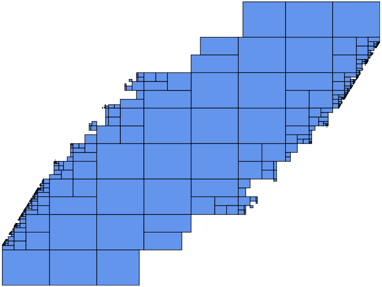
digital filter

```
s0 := input [-0.1,0.1]
s1 := input [-0.1,0.1]
while true do
   r := 1.5 * s0 - 0.7 * s1 + [-0.1,0.1]
   s1 := s0
   s0 := r
done
```

Example 2

digital filter

```
s0 := input [-0.1,0.1]
s1 := input [-0.1,0.1]
while true do
   r := 1.5 * s0 - 0.7 * s1 + [-0.1,0.1]
   s1 := s0
   s0 := r
done
```



after 16330 iterations
2.8 s wall clock

our implementation is still very naïve, it uses arbitrary precision rationals

Conclusion

Comparison with classic Constraint Programming

- both over-approximate the solution with sets of elements
  but CP minimizes the solution
- CP over-approximates $\mathrm{gfp}_D\, F^\sharp$ where $F(X) \stackrel{\text{def}}{=} X \cap C$
  we over-approximate $\mathrm{lfp}_I\, F^\sharp$ where F is a $\cup-$morphism
- both use decreasing iterations
  in CP, every iteration is sound
  we are sound only when the algorithm terminates successfully
- both use split, discard, and tighten operations
- CP discards elements only by consistency (propagation)
  we discard elements by choice

Comparison with classic Abstract Interpretation

- both seek to infer inductive program invariants
- both are parameterized by an arbitrary abstract domain $\mathcal{D}^\sharp$
- both use an abstract version $F^\sharp$ of the program semantics
  $F^\sharp$ is derived compositionally from the program syntax
- both can fail to find an inductive invariant
  for us, the choice of $\mathcal{D}^\sharp$ may not matter as much
  (we rely on $\mathcal{P}_{\text{finite}}(\mathcal{D}^\sharp)$, not $\mathcal{D}^\sharp$, to express the inductive invariant)
- AI starts with increasing iterations
  we only perform decreasing iterations
- AI iterates $F^\sharp$
  we iterate an inclusion check $F^\sharp(X) \subseteq Y$
- in AI, disjunctions are caused by control joins $\cup$, ∇
  we create disjunction to refine

We proposed a new algorithm:

- to infer inductive numeric program invariants
- inspired by Constraint Programming on continuous domains

Benefit:

- no *a priori* knowledge of the shape of the inductive invariant
  no need to design a dedicated numeric domain
  we use boxes instead of a relational or even a non-linear abstract domain

Limitations:

- very new and untried approach!
- large number of iterations (compared to a dedicated domain)
- we may fail due to a misguided choice
  high reliance on the choice and split strategies

Future work

- More robust implementation and testing
  (is our method really working?)
- Possible improvements:
  - more clever choices between split and discard
    (maybe backtracking, learning, etc.)
  - use other abstract domains $\mathcal{D}^\sharp$ beside boxes (e.g., octagons)
  - handled unbounded goal invariants $G^\sharp$
  - disjunctive completion: $F^\sharp : \mathcal{D}^\sharp \to \mathcal{P}_{\text{finite}}(\mathcal{D}^\sharp)$
  - classic decreasing iterations to improve the inductive invariant
- Relationship with logic-based methods
  (IC3 with SMT)
- Relationship with abstract interpretation iteration methods
  (is this iteration of some $F'^\sharp$ with dual narrowing $\tilde{\triangle}$?)