Parametric Abstract Domains for Shape Analysis

Xavier Rival
(INRIA & École Normale Supérieure)

Joint work with

Bor-Yuh Evan Chang (University of Maryland U University of Colorado)
and George Necula (University of California at Berkeley)
Purpose of Shape Analysis

- Infer precise information about memory layout:
  - pointers
  - dynamic, unbounded data-structures
    - e.g., lists, queues, stacks, trees
    - complex composite structures: e.g., in device drivers
- Wide range of applications:
  - proving memory safety
    - absence of memory leaks, null/dangling pointer dereference
  - proving the preservation of shapes
    - e.g., no cycle is introduced in what should be a tree
  - establishing stronger properties
    - e.g., correctness of a sorting algorithm
  - allow other analyses to support manipulation of complex structures
A Basic Example

• Closing a list of File Descriptors:

```
assume(l points to a list)
c = l;
while(c ≠ NULL){
    close(c → FD);
    c = c → next;
}
```

• Properties of interest:
  ♦ 1. memory safety: e.g., c = c → next; should not crash
    ▶ no null pointer dereference (easy on this program)
    ▶ no segmentation fault
  ♦ 2. preservation of the shape
    i.e., l should still point to a list
  ♦ 3. functional correctness
    i.e., at the end, all FDs are closed
Our Proposal

user-supplied inputs

structure descriptions
list, tree, ...

C code

parametric analyzer

parametric abstract domain
unfolding
widening

analysis engine
fixpoint computation

• Static analysis by abstract interpretation:
  ♦ sound
  ♦ automatic

• Based on an over-approximation of sets of stores
Main Difficulties

• Express the adequate properties:
  ♦ points-to relations, aliasing, etc
  ♦ data-structures need be summarized
  ♦ fragments of structures

• Infer these properties, using abstract interpretation techniques:
  ♦ analysis of elementary statements
  ♦ choice of a widening operator, to enforce termination

• Make the abstract domain expressive
Outline

√ An Abstract Domain for Shapes
  · Unfolding Edges: Local Concretization
  · Abstraction of Segments
  · Infering Shape Invariants
  · Relations among Shape Properties
  · Need for a Reduction Operator
  · Conclusion
Abstraction of Shapes

- Abstract value: a graph representation
  - symbolic nodes ($\alpha, \beta, \gamma \ldots$) represent integer values e.g., addresses, data...
  - edges stand for constraints about memory regions
  - a graph can be written as a separation logic formula i.e., constraints about a separating conjunction of regions ($\ast$)

- Simplest kind of edges: points-to edges
  - denoted $\alpha@f \rightarrow \beta$, means “at address $\alpha$ plus offset $f$, we read $\beta$”
  - graph representation:

    ![Graph Representation](image)

    denotes stores like:

    ![Store Representation](image)

- other kinds of edges to be defined later...
Concretization for Graphs (1)

- One graph stands for a set of concrete stores:
  e.g., \( \alpha \) \( \to \) \( \text{addr} \) \( \beta \) \( \to \) \( \text{addr} \) \( \gamma \) \( \to \) \( \text{addr} \) stands for stores like:

- Relations to state:
  ♦ node to addresses
  ♦ variables to addresses
Concretization for Graphs (2)

- **environment:** \( \mathcal{E} : \text{Var} \rightarrow \text{nodes} \)
- **address mapping:** \( \text{to_addr}[.] : \text{nodes} \rightarrow \mathbb{N} \)
- **stores:** \( \sigma : \mathbb{N} \rightarrow \mathbb{N} \)
- **rules, such as:**
  - **points-to edge:** \( \gamma(\alpha @ f \leftrightarrow \beta) = \{ [\text{to_addr}[\alpha] + f \mapsto \text{to_addr}[\beta]] \} \)
  
    ![Diagram of points-to edge]

  - **separating conjunction:**
    \[
    \gamma(S_0^\sharp \ast S_1^\sharp) = \{ \sigma_0 \otimes \sigma_1 \mid \sigma_0 \in \gamma(S_0^\sharp) \land \sigma_1 \in \gamma(S_1^\sharp) \}
    \]
    where \( \otimes \) "glues" two separate stores together

![Diagram of separating conjunction]
We have to deal with **unbounded structures**

**Lists:**

```c
typedef PList{
    void *data;
    typedef PList *next;
}PList;
```

How to **abstract regions** pointed to by a PList∗?

**Some concrete values:**

◊ the null pointer

◊ pointer to cells of addresses to_addr[αi] in the store below:
Inductive Definitions and Summarization

- **Summarization:**
  - predicate
    “a memory region satisfied inductive definition \( c \) at the address represented by \( \alpha \)”
    (concretization on the next slide...)
  - graph representation: \( \alpha \xrightarrow{c} \)

- **List inductive definition (or checker):**

  \[
  \alpha \cdot \text{list()} := \begin{align*}
  \alpha &= 0 \land \text{emp} \\
  \lor \quad \alpha &\neq 0 \land \alpha@\text{next} \mapsto \beta \land \alpha@\text{data} \mapsto \beta' \land \beta \cdot \text{list()}
  \end{align*}
  \]

- **Why also call these definitions “checkers”?**
  - programmers often write functions like `check_doubly_linked_list`
  - such code can be (almost) directly used (caveat: separation)
  - rough intuition: it “checks” that the memory region satisfies the property
Semantics of Inductive Definitions

- Checker edges are indexed with the induction depth
- The concretization of $\alpha_c$ is the join of the concretizations of all $\alpha_c^i$ ($i \in \mathbb{N}$)

- Example for lists:
  - at rank 0, $\alpha_{list}^0$ abstracts the empty store where $\alpha$ is 0
  - at rank 1, $\alpha_{list}^1$ abstracts stores of the form:
    - $\text{to_addr}[\alpha]$
    - @data
    - @next
    - NULL
  - at rank 2, $\alpha_{list}^2$ abstracts stores of the form:
    - $\text{to_addr}[\alpha]$
    - @data
    - @next
    - @data
    - @next
    - NULL
  - ...

Parametric Abstract Domains for Shape Analysis – p.12/48
Parameterization of the Abstraction

- Inductive definitions to use:
  - supplied by the user
  - depending on software to analyze

- A numerical abstraction is chosen:
  - $\alpha = 0$, $\alpha \neq 0$;
  - pointer equalities / disequalities;
  - “the File Descriptor pointed to by $\alpha \rightarrow \text{FD}$ is open / closed”
Outline

- An Abstract Domain for Shapes

✓ Unfolding Edges: Local Concretization
  - Abstraction of Segments
  - Inferring Shape Invariants
  - Relations among Shape Properties
  - Need for a Reduction Operator
  - Conclusion
An Example

- Back to the list example...

- First steps of the analysis:
  - **assume** statement: asserts an initial state
    
    - the assignment \( c = 1; \) remaps environment edges (not shown)
    
    - loop: requires computation of a least fix-point (*a few slides later...*)
      - first iteration: analyze the body, start with \( \alpha \neq 0 \)
      - assignment \( c = c \rightarrow \text{next}; \)
        
        - no next edge from \( \alpha \) in \( l, c \)
        
        analysis is stuck; need to deal with full checker edges
Checker Edge Unfolding

- Unfolding an inductive: simply open the definition!
  - edge unfolding: returns one symbolic disjunct per rule
  - rule unfolding:
    - removes former checker edge
    - creates fresh nodes for unboxed fields
    - adds recursive checker edges
    - accounts for side conditions

- Case of lists:

- Soundness of the unfolding:

  \[ \gamma(S^\#) \subseteq \bigcup \{ \gamma(S_0^\#) \mid S^\# \xrightarrow{\text{unfold}} S_0^\# \} \]

  - follows from the definition of checkers
Analysis of an Assignment

• Before the assignment $c = c \to \text{next;}$: 

$$
\begin{array}{c}
\alpha \neq 0 \\
\text{list} \\
l, c
\end{array}
$$

• Result of the unfolding: two rules to consider
  ♦ empty list does not need be considered
  contradiction with num. invariant $\alpha \neq 0$
  ♦ non-empty list case:

• Result of the assignment:

note: analyzing the assignment in itself is trivial (frame rule)
Outline

- An Abstract Domain for Shapes
- Unfolding Edges: Local Concretization

✓ Abstraction of Segments
  - Infering Shape Invariants
  - Relations among Shape Properties
  - Need for a Reduction Operator
  - Conclusion
Need for Folding (1)

- **First iterates** in the loop:
  - at iteration 0 (before entering the loop):
    - $\alpha_0$ list $l, c$
  - at iteration 1:
    - $\alpha_0$ next $l$ data $\beta_1$
    - $\alpha_1$ list $c$
  - at iteration 2:
    - $\alpha_0$ next $l$ data $\beta_1$
    - $\alpha_1$ next $\beta_2$
    - $\alpha_2$ list $c$

- **We are still not folding anything!**
  - how to abstract the “region between l and c”?

---

Parametric Abstract Domains for Shape Analysis – p.19/48
Notion of Segment

- After two iterations:

  \[
  \text{to_addr}[\alpha] @ \text{data} \to @ \text{next} \\
  \text{to_addr}[\beta] @ \text{data} \to @ \text{next} \\
  \text{NULL}
  \]

  properties to abstract:
  - \( \alpha \) points to a list;
  - \( \beta \) points to a list;
  - actually, \( \beta \) points to a sub-list of \( \alpha \)

- Summarization of a segment
  - a structure with a “hole”
    position of the cursor
  - store above abstracted by:

\[
\alpha \text{ list} \to \beta \text{ list}
\]
Formalization

- Segment edge $\alpha \rightarrow \beta$ roughly means: “if we add a memory region starting at address $\beta$, satisfying inductive $c'$, region starting at address $\alpha$ satisfies inductive $c$”

- Concretization: defined by induction as well!

For instance, in the case of lists:

- at rank 0, $\alpha_{\text{list}} \rightarrow \beta_{\text{list}}$:
  - abstracts the empty region;
  - states that $\alpha = \beta$

- at rank $n + 1$, $\alpha_{\text{list}} \rightarrow \beta_{\text{list}}$ abstracts stores of the form:
Back to the Example

• Back to the initial example:

```plaintext
assume(l points to a list)
c = l;
①while(c ≠ NULL){
    ②c = c → next; ③
}
④
```

• Best invariants:

♦ at ①: 

α

\[ \text{list} \]

\[ l, c \]

♦ at ②:

α

\[ \text{list} \]

\[ \text{list} \]

\[ β \]

\[ \text{c list} \]

and \( β \neq 0 \)

♦ at ③:

α

\[ \text{list} \]

\[ \text{list} \]

\[ β \]

\[ \text{c list} \]

and \( α \neq β \)

♦ at ④:

α

\[ \text{list} \]

\[ \text{list} \]

\[ β \]

\[ \text{c list} \]

and \( β = 0 \)

• How to infer these invariants?
Outline

- An Abstract Domain for Shapes
- Unfolding Edges: Local Concretization
- Abstraction of Segments

√ Infering Shape Invariants
- Relations among Shape Properties
- Need for a Reduction Operator
- Conclusion
Need for Folding (2)

• First iterates in the loop:
  ♦ at iteration 0 (before entering the loop):

  ![Diagram](attachment:diagram1.png)

  ♦ at iteration 1:

  ![Diagram](attachment:diagram2.png)

  ♦ at iteration 2:

  ![Diagram](attachment:diagram3.png)

• How to infer the loop invariant?
  How to introduce a segment edge?

• This is the purpose of widening!
Widening Algorithm

- **Widening** should achieve two things:
  - soundness, i.e. compute a common over-approximation
  - enforce termination

- **Separation:** If $\forall i \in \{0, 1\}$, $\forall s \in \{\text{lft, rgh}\}$, $\gamma(S^\#_{i,s}) \subseteq \gamma(S^\#_s)$, then:
  - $\forall s, \gamma(S^\#_{0,s} * S^\#_{1,s}) \subseteq \gamma(S^\#_{\text{lft}} * S^\#_{\text{rgh}})$
  - we can choose $S^\#_{\text{lft}} * S^\#_{\text{rgh}}$ as our widening:

![Diagram](image)

- **join algorithm:**
  - split both graphs into regions, thanks to $\Psi$
    - maintain a node-to-node mapping relation $\Psi$ (address mapping)
  - find a common abstraction for well-chosen pairs of regions
Rule:

\[
\begin{align*}
S_{\text{left}} & \quad \alpha \quad S_{\text{right}} \quad \beta_0 \\
\Psi & \quad \Psi \\
\beta_0 \quad S_{\text{left}} \quad S_{\text{right}} \quad \beta_1 & \subseteq \beta_0 \quad \beta_1
\end{align*}
\]

if \( S_{\text{left}} \nabla S_{\text{right}} \) then

\[
\begin{align*}
S_{\text{left}} \nabla S_{\text{right}} & = \gamma_0 \quad \gamma_1 \\
(\alpha, \beta_0) & \leftrightarrow \gamma_0 \\
(\alpha, \beta_1) & \leftrightarrow \gamma_1
\end{align*}
\]

Application to list traversal, at the end of iteration 1:

- before iteration 0:
  - before iteration 0:
  - end of iteration 0:
  - join, before iteration 1:
Segment Edges Extension

- Rule:

\[
\begin{align*}
S^\#_{\text{lf}} & \nabla S^\#_{\text{rgh}} = \\
(\alpha_0, \beta_0) & \leftrightarrow \gamma_0 \\
(\alpha_1, \beta_1) & \leftrightarrow \gamma_1
\end{align*}
\]

- Application to list traversal, at the end of iteration 1:

  ♦ previous invariant before iteration 1:

  ♦ end of iteration 1:

  ♦ join, before iteration 1:

\[
\begin{align*}
\Psi(\alpha_0, \beta_0) & = \gamma_0 \\
\Psi(\alpha_1, \beta_2) & = \gamma_1
\end{align*}
\]
Comparison Operator

- **Algorithm structure:** quite similar to join...
  - based on separation and local rules:
    \[ \gamma(S_0^\#) \subseteq \gamma(S_1^\#) \implies \gamma(S_0^\# \ast S^\#) \subseteq \gamma(S_1^\# \ast S^\#) \]
  - use of a counterpart for \( \Psi \)

- **A set of structural rules** such as:
  - segment splitting:
    - \[ S_0^\# \sqsubseteq \begin{array}{c} \alpha \end{array}_c \implies S_0^\# \ast \begin{array}{c} \beta \end{array}_c \sqsubseteq \begin{array}{c} \beta \end{array}_c \]
  - full checker folding:
    - \[ S_0^\# \sqsubseteq S_0^\# \implies S_0^\# \sqsubseteq \begin{array}{c} \alpha \end{array}_c \]

- **Correctness:**
  - \[ S_0^\# \sqsubseteq S_1^\# \implies \gamma(S_0^\#) \subseteq \gamma(S_1^\#) \]
Widening Properties

- **Soundness**: ensured by construction

\[
\begin{align*}
\gamma(S_0^\#) & \subseteq \gamma(S_0^\# \nabla S_1^\#) \\
\gamma(S_1^\#) & \subseteq \gamma(S_0^\# \nabla S_1^\#)
\end{align*}
\]

- **Termination**: The following sequence is ultimately stable

\[
S_0^\# \quad \text{abstraction at loop entrance}
\]

\[
S_{n+1}^\# = S_n^\# \nabla F^\#(n^\#)
\]

where \( F^\# \) interprets the loop body in the abstract level
Outline

- An Abstract Domain for Shapes
- Unfolding Edges: Local Concretization
- Abstraction of Segments
- Infering Shape Invariants

√ Relations among Shape Properties
- Need for a Reduction Operator
- Conclusion
Beyond Lists and Trees: Back Pointers

- **Trees:** quite similar to lists...

- What about doubly-linked lists?

  - new issue: back pointers

  \[ x \neq \text{NULL} \land x \rightarrow \text{next} \neq \text{NULL} \implies x \rightarrow \text{next} \rightarrow \text{prev} = x \]

  - inductive definition: needs a parameter, to check pointer relations
Inductive Definitions with Parameters

- Inductive checker:
  - $\alpha \text{ dll}(\beta)$ unfolds into two disjuncts
  - $\alpha = 0 \lor \alpha \neq 0 \land \alpha \text{ @ next} \mapsto \gamma \land \alpha \text{ @ prev} \mapsto \beta \land \gamma \text{ dll}(\alpha)$

- Note that:
  - Parameter $\beta$ specifies where the back-edge is pointing to
  - Parameter for next unfold is $\alpha$ itself
  - Head of the list: parameter is NULL

- Concretization:
  - Defined as usual, by inductive unfolding
Unfolding Segments Backwards

- **Issue** with unfolding:
  - ♦ **backward traversal** of doubly-linked lists does **not** match the “built-in” induction scheme
  - ♦ **example:**


```
assume(l points to a dll)
c = l;
while(c != NULL && condition)
  c = c → next;
if(c != 0 && c → prev != 0) c = c → prev → prev;
```

- In fact, **two** separate issues need be solved:
  - ♦ **how to unfold a segment?** (semantic point of view)
  - ♦ **discover automatically when** the analysis needs it
Segment Splitting

- Segment edges:
  - defined by induction over lengths of call chains
  - unfolding: requires decomposing the segments

- Splitting lemma:
  \[
  \alpha_{c} \xrightarrow{i+j} \alpha'_{c'} \]
  describes the same set of stores as:
  \[
  \alpha_{c} \xrightarrow{i} \alpha''_{c''} \xrightarrow{j} \alpha'_{c'}
  \]

- Backward unfolding algorithm:
  - treat the case of the segment of length 0 separately
    assert equalities of nodes, parameters...
  - for a segment of length \(j + 1\), split it into
    - a segment of length \(j\);
    - a segment of length 1, which is trivial to unfold
We focus on the non empty segment case:

1. Segment splitting:

2. Unfolding of the 1-segment:

3. Unfolding of the 0-segment, and reducing equalities ($\alpha_1 = \alpha_3$, $\alpha_2 = \beta_1$):

4. Assignment $c = c \rightarrow \text{prev}$ can be analyzed
Forward unfolding of segments works similarly disjunction for the empty segment, splitting, unfolding of a 1-segment

Useful in the following case:

```
assume(l points to a dll)
c = l;
while(c != NULL && condition)
    c = c -> next;
if(c != l && condition)
    c = c -> prev;
```

Results in:
When to Perform Backward Unfolding

- An abstraction:

- to expose \(c \rightarrow \text{prev}\): unfold the full checker
- but then, how to unfold \(c \rightarrow \text{prev} \rightarrow \text{prev}\) or \(c \rightarrow \text{prev} \rightarrow \text{next}\)?

- A concrete element:

- \(c\) is in checker call number 3
- \(c \rightarrow \text{prev}\) points into call number 2 i.e., \(3 - 1\)
- to expose \(c \rightarrow \text{prev} \rightarrow \text{prev}\) unfold previous call (end of the segment)

- Inference of field levels using a typing system
Benchmarks

- **Extensible Inductive Shape Analyzer** (http://xisa.cs.berkeley.edu/)
- **Analyzes** standard algorithms on
  - lists, doubly-linked lists, fixed level skip-lists
  - binary trees...
- A few **selected test cases**:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (sec)</th>
<th>Disjuncts</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>doubly-linked list insertion</td>
<td>0.0038</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>search tree with back pointers</td>
<td>0.0470</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>insert and back to root</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scull driver (without strings) ≡ 900 lines</td>
<td>9.710</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>
Outline

- An Abstract Domain for Shapes
- Unfolding Edges: Local Concretization
- Abstraction of Segments
- Infering Shape Invariants
- Relations among Shape Properties

✓ Need for a Reduction Operator
- Conclusion
Uniqueness of Abstraction

- There is **NO unique abstraction** in general:
  - existence of distinct ways to express a same property
  - may cause the analysis to fail

- **Forward vs backward** doubly-linked list checkers:

  ![Forward vs backward doubly-linked list checkers](image)

- Then the following two segments **have the same concretization**:

  ![Two segments have the same concretization](image)

  they represent stores such as:

  ![Stores representation](image)
Forward and Backward Inductive Definitions

• In certain cases, unfolding of segments fixes the problem:
  For instance, to uncover $c \rightarrow \text{prev}$:

  $\alpha_0 \xrightarrow{\text{FWD\_dll}(\alpha_0)} \alpha_1 \xrightarrow{\text{FWD\_dll}(\alpha_2)} \alpha_3 \xrightarrow{\text{BWD\_dll}(\alpha_1)} \alpha_4$

  **Unfold**

  $\alpha_0 \xrightarrow{\text{FWD\_dll}(\alpha_0)} \alpha_1 \xrightarrow{\text{FWD\_dll}(\alpha_2)} \alpha_3 \xrightarrow{\text{BWD\_dll}(\alpha_1)} \alpha_4$

  **Unfold**

  $\alpha_0 \xrightarrow{\text{FWD\_dll}(\alpha_0)} \alpha_1 \xrightarrow{\text{FWD\_dll}(\alpha_2)} \alpha_3 \xrightarrow{\text{BWD\_dll}(\alpha_1)} \alpha_4$

• However, folding definitely fails:
  Our widening cannot fold

  $\alpha_0 \xrightarrow{\text{BWD\_dll}(\alpha_0)} \alpha_1 \xrightarrow{\text{BWD\_dll}(\alpha_2)} \alpha_3 \xrightarrow{\text{FWD\_dll}(\alpha_1)} \alpha_4$

  Though, it abstracts the same stores as
What Needs to be Done?

- Automatically derive inclusions such as:
  \[ \beta_0 \xrightarrow{\text{BWD} \_ \text{dll}(\alpha_0)} \beta \\preceq \ \sigma \xrightarrow{\text{FWD} \_ \text{dll}(\beta)} \beta_1 \]

  ♦ we do not want to rely on user-written assumptions
  ♦ we could build a decision procedure
  ♦ but, we can also use our abstract domain here!

- Infer when such a “reduction” should be used

- How to set up an abstract interpretation of checkers?
  an operational semantics for inductive definitions is needed
An Alternate Concretization for Inductive Defs.

- Decomposing our concretization for graphs:

  abstract graphs

  \[ \beta_0 \xleftarrow{\text{BWD_dll(} \alpha_0 \text{)}} \beta_1 \]

  set of "concrete" graphs (only p.t. edges)

  \[ \beta_0 \xrightarrow{\text{prev}} \beta_1 \xrightarrow{\text{prev}} \alpha_1 \]

  set of concrete stores (addresses mapped)

  \[ \gamma \mapsto \text{to addr}[\beta_0] \quad \text{to addr}[\alpha_0] \quad \text{to addr}[\beta_1] \quad \text{to addr}[\alpha_1] \]

- Main property: \( \gamma = \gamma_{\text{map}} \circ \gamma_{\text{oper}} \)

- Operational semantics:
  - defined as an \text{lfp}
  - abstract interpretation of it can be performed
Inductive Definitions as Transition Systems

- Computing the semantics of $S^\# = \begin{array}{c} \beta_0 \\ \text{BWD.dll(\alpha_0)} \end{array} \rightarrow \begin{array}{c} \beta_1 \\ \text{BWD.dll(\alpha_1)} \end{array}$

  ♦ configurations:
  
  \[
P^\#, (\beta_0, \alpha_0), (\beta_1, \alpha_1)
  \]

  ♦ initial states (empty segments):
  
  \[
  \mathcal{I}_{\text{init}} = \begin{array}{c} \alpha_0 \\ \beta_0 \end{array} (\beta_0, \beta_0), (\alpha_0, \alpha_0)
  \]

  ♦ transitions (inductive case):
  
  \[
  (\beta_0, \alpha_0), (\beta_1, \alpha_1) \rightarrow (\beta_0, \alpha_0), (\alpha_0, \alpha_2)
  \]

  ♦ Program generating $\gamma_{\text{oper}}(S^\#)$:

  \[
  \begin{align*}
  \mathcal{I} &= \mathcal{I}_{\text{init}}; \\
  \text{while}(\text{TRUE}) ; \\
  \mathcal{I} &= \{P^\#_{\text{nxt}} \mid P^\# \in \mathcal{I} \land P^\# \Rightarrow P^\#_{\text{nxt}} \}
  \end{align*}
  \]

  • The transition system is systematically derived from the inductive
Abstract Interpretation of An Inductive Def.

- Abstract interpretation:
  - abstraction: sets of stores, abstracted with our domain, with only FWD_dll as a parameter
  - computation: abstract interpretation of the transition system

- First abstract iterates:
  - at rank 0: $I_{init}$
  - at the end of iter 1:
  - widening: introduces the segment edge
Abstract Interpretation of An Inductive Def.

- End of the analysis:
  - next iterate:
    transition adds a pair of edges:
    \[ \alpha_0 = \beta_1 \]
    \[ FWD_{dll}(\beta_0) \quad FWD_{dll}(\beta_1) \]
    \[ (\beta_0, \alpha_0), (\beta_1, \alpha_2) \]
  - widening:
    \[ (\alpha_0, \alpha_2) \]
    \[ FWD_{dll}(\beta_0) \quad FWD_{dll}(\alpha_1) \]
    \[ (\beta_0, \alpha_0), (\beta_1, \alpha_2) \]
  - the above is stable!
    \[ \gamma_{\text{oper}}(\beta_0, \alpha_0) \subseteq \gamma_{\text{oper}}(\beta_0, \alpha_0) \]

- A similar approach works in other cases of conversions:
  - weakening structures: FD list → list
  - turning segments into full checkers, with additional parameters
Outline

- An Abstract Domain for Shapes
- Unfolding Edges: Local Concretization
- Abstraction of Segments
- Infering Shape Invariants
- Relations among Shape Properties
- Need for a Reduction Operator

✓ Conclusion
Towards Shape Analysis Abstract Domains

- Our main contributions:
  - a parametric shape analysis domain
    - structures to check are not built-in
    - a numerical abstraction can also be chosen
  - good support for local concretization (unfolding)
  - a widening operator
  - powerful abstraction: usable even to analyze its parameterization

- Long term goal:
  make a parametric shape analysis domain available for other analyses

- Many theoretical and practical issues yet to solve:
  - reduction
  - arrays
  - ...