Abstract interpretation for the heap

Computer Science and Artificial Intelligence Laboratory
MIT

Nov 18, 2011
Consider the following program

```c
struct Node{
    int v;
    Node * nxt;
}

void foo(){
    head = new Node();
    n = head;
    i=0;
    while(i<10){
        n->nxt = new Node();
        n = n->nxt;
    }
}
```

How can we reason about a program like this?
Questions to ask

Is the language typesafe?
- Can we assume the only way to get the value of field nxt is by reading something->nxt?
- Is there pointer arithmetic? Do we care?

How much precision do we need?
- flow sensitive vs. flow insensitive
- field sensitive vs. field insensitive
- context sensitive vs. context insensitive
- what properties do we care about?
  - aliasing, null checking, structural properties (e.g. reachability)

Today we’ll focus on flow-sensitive, field-sensitive for typesafe languages
Modeling the heap

The heap is a mapping from an address and a field to a value: \( h(addr, field) \rightarrow val \)
- Address fields can be represented as a graph
  - fields correspond to edge labels

\[
\langle s, (\sigma, h) \rangle \rightarrow (\epsilon, h') \quad \langle e, (\sigma, h) \rangle \rightarrow v
\]

```
struct Tnode{
    Tnode* left;
    Tnode* right;
}
```
Program state

Often we only care about pointers in the program

- heap is then $h(addr, field) \rightarrow addr$.
  - Sometimes also represented as a set of triples $E_s = \{(addr, field, addr)\}$
  - fields are sometimes called selectors
- the stack is then $\sigma(var) \rightarrow addr$
  - Sometimes also represented as a set of pairs $E_v = \{(var, addr)\}$
Abstraction

Key challenge: represent an infinite set of graphs with a single structure
- the heaps that can arise at a particular program point are unbounded

General strategy: collect arbitrary sets of nodes into a single summary node
- Shape analysis is all about determining how to do this grouping
Approach 1: static abstraction

Collect nodes based on some statically determined criteria

- Common candidates include: type, allocation site, allocation site + bounded context for interprocedural analysis
Precision loss

What can we tell from this graph?
- x.left != x
- x.right != null → x.right != x.left

What can we not tell?
- x and z were not aliased
- x.right is not null
- x.left is not null
Destructive updates

We can not do destructive updates on abstract nodes
With this form of abstraction, all nodes are abstract

expression evaluates to a set of abstract addresses

\[
\begin{align*}
\langle x, (\overline{\sigma}, \overline{h}) \rangle &\rightarrow s & \langle e, (\overline{\sigma}, \overline{h}) \rangle &\rightarrow b \\
\langle x. f := e, (\overline{\sigma}, \overline{h}) \rangle &\rightarrow (\overline{\sigma}, \overline{h}[\forall a \in s. (a, f) \rightarrow b \sqcup \overline{h}(a, f)])
\end{align*}
\]

\[
\begin{align*}
\langle x := \text{new } X_i, (\overline{\sigma}, \overline{h}) \rangle &\rightarrow (\overline{\sigma}[x \rightarrow \{i\}], \overline{h}[\forall f (i, f) \rightarrow \text{something} \sqcup \overline{h}(i, f)])
\end{align*}
\]
Why you can’t do destructive updates

t = null;
x = null;
while(*){
    t = x;
    x = new X(); // a1
    x.y = null;
    if(t != null){
        t.y = x;
        assert x.y != null;
    }
}

Approach 2: dynamic abstraction

Create abstract nodes dynamically.

Simple strategy:
- For every allocation site, have 2 abstract nodes
  - one for the last allocation $i_{\text{fresh}}$,
  - one for everything that came before $i_{\text{old}}$

\[
\overline{\sigma}' = ?? \quad \overline{h}' = ?? \\
\langle x := \text{new } X_i, (\overline{\sigma}, \overline{h}) \rangle \rightarrow (\overline{\sigma}'[x \rightarrow \{i_{\text{fresh}}\}], \overline{h}'[\forall f \ (i_{\text{fresh}}, f) \rightarrow \text{something}])
\]

\[
\langle x, (\overline{\sigma}, \overline{h}) \rangle \rightarrow s \quad \langle e, (\overline{\sigma}, \overline{h}) \rangle \rightarrow b \\
\langle x.f := e, (\overline{\sigma}, \overline{h}) \rangle \rightarrow (\overline{\sigma}, \overline{h}[\forall a_{\text{old}} \in s. \ (a_{\text{old}}, f) \rightarrow b \cup \overline{h}(a_{\text{old}}, f)], \forall a_{\text{new}} \in s. \ (a_{\text{new}}, f) \rightarrow b)
\]
Approach 2: dynamic abstraction

Create abstract nodes dynamically.

Simple strategy:
- For every allocation site, have 2 abstract nodes
  - one for the last allocation,
  - one for everything that came before

\[ \forall \in \rightarrow \\subseteq \rightarrow (\forall) \]

\[ \approx = \approx \rightarrow \{\{\}} \]

If we think of \( \bar{h} \) as a relation consisting of triples \((s,f,d)\),
then \( \bar{h}' \) is a relation defined as follows

\[ \{(s, f, d) \mid (s, f, d) \in \bar{h}, s \neq i_{\text{new}}, d \neq i_{\text{new}}\} \cup \]
\[ \{(s, f, i_{\text{old}}) \mid (s, f, i_{\text{new}}) \in \bar{h}, s \neq i_{\text{new}}\} \cup \]
\[ \{(i_{\text{old}}, f, d) \mid (i_{\text{new}}, f, d) \in \bar{h}, d \neq i_{\text{new}}\} \cup \]
\[ \{(i_{\text{old}}, f, i_{\text{old}}) \mid (i_{\text{new}}, f, i_{\text{new}}) \in \bar{h}\} \]

\[ \bar{\sigma}' = \bar{\sigma} [\forall x_{\text{s.t.}} \bar{\sigma}[x] = i_{\text{fresh}}, x \rightarrow i_{\text{old}}] \]
\[ \bar{h}' = ?? \]
\[ \langle x := \text{new } X_i, (\bar{\sigma}, \bar{h}) \rangle \rightarrow (\bar{\sigma}'[x \rightarrow \{i_{\text{fresh}}\}], \bar{h}'[\forall f (i_{\text{fresh}}, f) \rightarrow \text{something}]) \]

\[ \langle x, (\bar{\sigma}, \bar{h}) \rangle \rightarrow s \quad \langle e, (\bar{\sigma}, \bar{h}) \rangle \rightarrow b \]
\[ \langle x. f := e, (\bar{\sigma}, \bar{h}) \rangle \rightarrow (\bar{\sigma}, \bar{h}[\forall a_{\text{old}} \in s. (a_{\text{old}}, f) \rightarrow b \cup \bar{h}(a_{\text{old}}, f)]) \]
\[ \forall a_{\text{new}} \in s. (a_{\text{new}}, f) \rightarrow b \]
t=null;
x=null;
while(*){
    t = x;
    x = new X(); // a1
    x.y = null;
    if(t != null){
        t.y = x;
        assert x.y != null;
    }
}

More Flexible Abstraction

Solving Shape-Analysis Problems in Languages with Destructive Updating
Sagiv, Reps and Wilhelm, POPL 93

Interesting idea:
- Locations pointed to by a variable should stay concrete
- A single abstract location represents everything else
Example

Concrete

Abstract
The node n2 is materialized from the abstract node. Left or right can point to either the abstract null or node