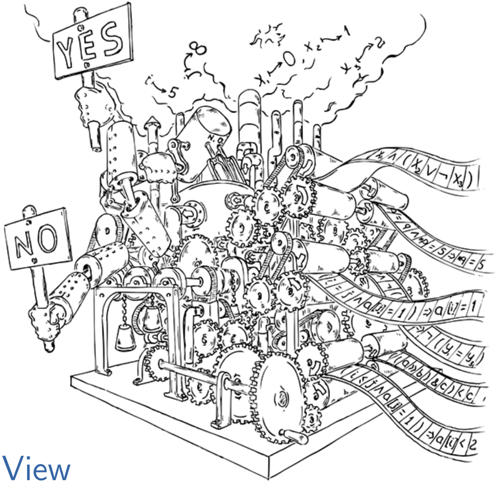
Pointers

Chapter 8



Decision Procedures

An Algorithmic Point of View

D.Kroening O.Strichman

Revision 1.0

Outline

Pointer: a program variable that refers to some other program construct

This other construct may be

- another variable, including a pointer,
- a function or method.

- Pointers to other variables allow code fragments to operate on different sets of data
- This avoids inefficient copying of data
- Pointers enable dynamic data structures
- But: Many bugs relate to the (ab-)use of pointers

Implementation

- Memory cells of a computer have *addresses*,
  i.e., each cell has a unique number

- The value of a pointer is such a number

- **memory model**: the way the memory cells are addressed

Definition (Our Memory Model)

- Set of addresses A is a subinterval of the integers $\{0, \ldots, N-1\}$
- Each address corresponds to a memory cell that is able to store one data word.
- The set of data words is denoted by D.
- *Memory valuation* $M : A \longrightarrow D$

(this is a continuous, uniform address space)

A variable may require more than one data word to be stored in memory

Examples:

- structs,
- arrays,
- double-precision floating-point

A variable may require more than one data word to be stored in memory

Examples:

- structs,
- arrays,
- double-precision floating-point

Let $\sigma(v)$ with $v \in V$ denote the size (in data words) of v.

Let V denote the set of variables.

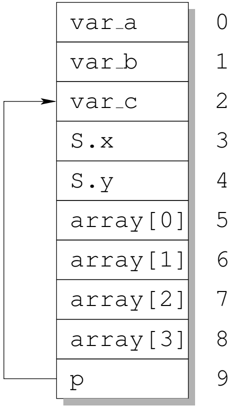
Definition (memory layout)

A *memory layout* $L : V \longrightarrow A$ is a mapping from V to an address A. The address of $v \in V$ is also called the *memory location* of v.

- The memory locations of the statically allocated variables are usually *non-overlapping*
- The memory layout is not necessarily continuous (e.g., due to alignment restrictions)

```
int var_a, var_b, var_c;
struct { int x; int y; } S;
int array[4];
int *p = &var_c;

int main() {
  *p=100;
}
```



- There is an area of memory (called heap) for objects that are created at run time
- A library maintains a list of the memory regions that are unused
- Some function allocates a memory region of a given size and returns a pointer to it
  - `malloc()` in C,
  - **`new`** in C++, C#, and Java.

Example from Program Analysis

Program analysis tools often need to reason about pointers

```
void f(int *sum) {
  *sum = 0;

  for(i=0; i<10; i++)
    *sum = *sum + array[i];
}
```

Example from Program Analysis

Program analysis tools often need to reason about pointers

```
void f(int *sum) {
  *sum = 0;

  for(i=0; i<10; i++)
    *sum = *sum + array[i];
}
```

- This program does not obey the obvious specification if the address held by `sum` is equal to the address of `i`
- Aliasing not anticipated by the programmer is a common source of problems

Definition (Pointer Logic)

Syntax:

$$
\begin{array}{rcl}
formula & : & formula \wedge formula \mid \neg formula \mid (formula) \mid atom \\
atom & : & pointer \ = \ pointer \mid term \ = \ term \mid \\
 & & pointer \ < \ pointer \mid term \ < \ term \\
pointer & : & pointer - identifier \mid pointer + term \mid (pointer) \mid \\
 & & \&identifier \mid \& * pointer \mid * pointer \mid \mathrm{NULL} \\
term & : & identifier \mid * pointer \mid term \ op \ term \mid (term) \mid \\
 & & integer - constant \mid identifier \ [\ term \ ] \\
op & : & + \mid -
\end{array}
$$

Warning: $=$ is equality here, not assignment

Example

Let p, q denote pointer identifiers, and let i, j denote integer identifiers.

The following formulas are well-formed according to the grammar:

- $*(p + i) = 1$,
- $*(p + *p) = 0$,
- $p = q \land *p = 5$,
- $*****p = 1$,
- $p < q$.

Example

The following formulas are not permitted by the grammar:

- $p + i$,
- $p = i$,
- $*(p + q)$,
- $*1 = 1$,
- $p < i$.

Semantics

- We define the semantics by referring to a specific memory layout L and a specific memory valuation M.

- Pointer logic formulas are predicates on M, L pairs

- We obtain a reduction to integer arithmetic and array logic

We define a semantics using the function

$$[\![\cdot]\!] : \mathcal{L}_P \longrightarrow \mathcal{L}_D$$

$\mathcal{L}_P$: language of pointer expressions
$\mathcal{L}_D$: expressions over variables with values from D

Defined recursively.

Boolean connectives:

$$\begin{array}{rcl} [\![f_1 \wedge f_2]\!] & = & [\![f_1]\!] \wedge [\![f_2]\!] \\ [\![\neg f]\!] & = & \neg [\![f]\!] \end{array}$$

Predicates:

$$\begin{array}{rcll} [\![p_1 = p_2]\!] & = & [\![p_1]\!] = [\![p_2]\!] & \text{where } p_1,\ p_2 \text{ are pointer expressions} \\ [\![p_1 < p_2]\!] & = & [\![p_1]\!] < [\![p_2]\!] & \text{where } p_1,\ p_2 \text{ are pointer expressions} \\ [\![t_1 = t_2]\!] & = & [\![t_1]\!] = [\![t_2]\!] & \text{where } t_1,\ t_2 \text{ are terms} \\ [\![t_1 < t_2]\!] & = & [\![t_1]\!] < [\![t_2]\!] & \text{where } t_1,\ t_2 \text{ are terms} \end{array}$$

Non-pointer terms:

$$
\begin{array}{rcll}
[\![v]\!] & = & M[L[v]] & \text{where } v \in V \text{ is a variable with } \sigma(v) = 1 \\
[\![t_1 \; op \; t_2]\!] & = & [\![t_1]\!] \; op \; [\![t_2]\!] & \text{where } t_1, t_2 \text{ are terms} \\
[\![c]\!] & = & c & \text{where } c \text{ is an integer constant} \\
[\![v[t]]\!] & = & M[L[v] + [\![t]\!]] & \text{where } v \text{ is an array identifier, } t \text{ is a term}
\end{array}
$$

Pointer-related expressions:

$$\begin{array}{rcll}
[\![p]\!] & = & M[L[p]] & \text{where } p \text{ is a pointer identifier} \\
[\![p+t]\!] & = & [\![p]\!] + [\![t]\!] & \text{where } p \text{ is a pointer expression, } t \text{ is a term} \\
[\![\&v]\!] & = & L[v] & \text{where } v \in V \text{ is a variable} \\
[\![\& * p]\!] & = & [\![p]\!] & \text{where } p \text{ is a pointer expression} \\
[\![\mathrm{NULL}]\!] & = & 0 & \\
[\![*p]\!] & = & M[[\![p]\!]] & \text{where } p \text{ is a pointer expression}
\end{array}$$

Notation

- A pointer p points to a variable x if $M[L[p]] = L[x]$
- Shorthand: $p \hookrightarrow z$ for $*p = z$

Warning: the meaning $p + i$ does not depend on the type of p

Let a be an array identifier:

$$*((\&a) + 1) \; = \; a[1]$$

The definition expands as follows:

$$[\![*((\&a) + 1) \; = \; a[1]]\!]$$

Example I

Let a be an array identifier:

$$*((\&a) + 1) \; = \; a[1]$$

The definition expands as follows:

$$[\![*((\&a) + 1) \; = \; a[1]]\!] \quad \iff \quad [\![*((\&a) + 1)]\!] = [\![a[1]]\!]$$

Example I

Let a be an array identifier:

$$*((\&a) + 1) \; = \; a[1]$$

The definition expands as follows:

$$\begin{aligned} [\![*((\&a) + 1) \; = \; a[1]]\!] \quad &\iff \quad [\![*((\&a) + 1)]\!] \; = \; [\![a[1]]\!] \\ &\iff \quad M[[\![(\&a) + 1]\!]] \; = \; M[L[a] + [\![1]\!]] \end{aligned}$$

Example I

Let a be an array identifier:

$$*((\&a) + 1) \;=\; a[1]$$

The definition expands as follows:

$$\begin{aligned} [\![*((\&a) + 1) \;=\; a[1]]\!] \quad &\iff \quad [\![*((\&a) + 1)]\!] \;=\; [\![a[1]]\!] \\ &\iff \quad M[[\![(\&a) + 1]\!]] \;=\; M[L[a] + [\![1]\!]] \\ &\iff \quad M[[\![\&a]\!] + [\![1]\!]] \;=\; M[L[a] + 1] \end{aligned}$$

Example I

Let a be an array identifier:

$$*((\&a)+1) \; = \; a[1]$$

The definition expands as follows:

$$\begin{aligned}
[\![*((\&a)+1) \; = \; a[1]]\!] \quad & \iff \quad [\![*((\&a)+1)]\!] \; = \; [\![a[1]]\!] \\
& \iff \quad M[[\![(\&a)+1]\!]] \; = \; M[L[a]+[\![1]\!]] \\
& \iff \quad M[[\![\&a]\!]+[\![1]\!]] \; = \; M[L[a]+1] \\
& \iff \quad M[L[a]+1] \; = \; M[L[a]+1]
\end{aligned}$$

The last formula is obviously valid.

Example II

The translated formula must evaluate to true for any L and M!

The following formula is not valid:

$$*p = 1 \longrightarrow x = 1$$

For $p \neq \&x$, this formula evaluates to false.

Axiomatization of the Memory Model

- It is possible to exploit assumptions made about the memory model.
- Depends highly on the architecture!
- Here: we formalize properties that *most* architectures comply with.

Example

On most architectures, the following two formulas are valid:

1. $\&x \neq \mathrm{NULL}$
2. $\&x \neq \&y$

(1) translates into $L[x] \neq 0$ and relies on the fact that no object has address 0.

(2) relies on non-overlapping addresses

Memory Axiom 1

Memory Model Axiom ("No object has address 0")

$$\forall v \in V.\ L[v] \neq 0$$

How do we address (2)?

Suggestion:

$$\forall v_1, v_2 \in V.\ v_1 \neq v_2 \longrightarrow L[v_1] \neq L[v_2]$$

The following two conditions together are stronger:

Memory Model Axiom ("Objects have size at least one")

$$\forall v \in V.\ \sigma(v) \geq 1$$

Memory Model Axiom ("Objects do not overlap")

$$\forall v_1, v_2 \in V.\ v_1 \neq v_2 \longrightarrow \begin{array}{l} \{L[v_1], \ldots, L[v_1] + \sigma(v_1) - 1\} \cap \\ \{L[v_2], \ldots, L[v_2] + \sigma(v_2) - 1\} = \emptyset\ . \end{array}$$

More?

Some code relies on additional, architecture-specific guarantees, e.g.,

- byte ordering
- endianness,
- alignment,
- structure layout.

Some program analysis tools allow adding such rules.

Structure Types

- Convenient way to implement data structures
- We add this as a syntactic extension
- Notation: $s.f$ to denote the value of the field f in the structure s

- Each field of the structure is assigned a unique offset: $o(f)$
- Meaning of $s.f$:

$$s.f \quad \doteq \quad *((\&s) + o(f))$$

- Following PASCAL and ANSI-C syntax:

$$p\text{->}f \quad \doteq \quad (*p).f$$

- Adopted from separation logic:

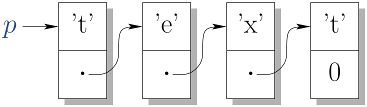
$$p \hookrightarrow a, b, c, \ldots \quad \doteq \quad \begin{array}{ll} *(p+0) = a & \wedge \\ *(p+1) = b & \wedge \\ *(p+2) = c & \ldots\,. \end{array}$$

Modeling Lists

- Simplest dynamically allocated data structure

- Realized by means of a structure type that contains fields for a next pointer

List Example



$$
\begin{array}{rrcl}
 & p & \hookrightarrow & \text{'t'},\, p_1 \\
\wedge & p_1 & \hookrightarrow & \text{'e'},\, p_2 \\
\wedge & p_2 & \hookrightarrow & \text{'x'},\, p_3 \\
\wedge & p_3 & \hookrightarrow & \text{'t'},\, \text{NULL}\ .
\end{array}
$$

Define a recursive shorthand for the i-th member of a list:

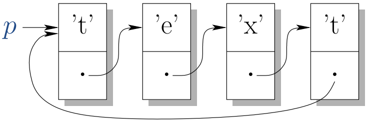
$$\begin{array}{rcl} \text{list-elem}(p, 0) & \doteq & p\ , \\ \text{list-elem}(p, i) & \doteq & \text{list-elem}(p, i-1)\text{->}n \quad \text{for } i \geq 1 \end{array}$$

We use 'n' as the next pointer field.

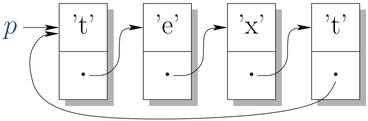
Now define the shorthand $\text{list}(p, l)$:

$$\text{list}(p, l) \quad \doteq \quad \text{list-elem}(p, l) = \text{NULL}$$

A linked list is cyclic if the pointer of the last element points to the first one:



A linked list is cyclic if the pointer of the last element points to the first one:

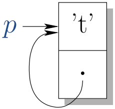


Would this work?

$$\text{my-list}(p, l) \quad \doteq \quad \text{list-elem}(p, l) = p \; .$$

Cyclic Lists

Unfortunately, the following satisfies my-list$(p, 4)$:



$\longrightarrow$ need to rule out sharing

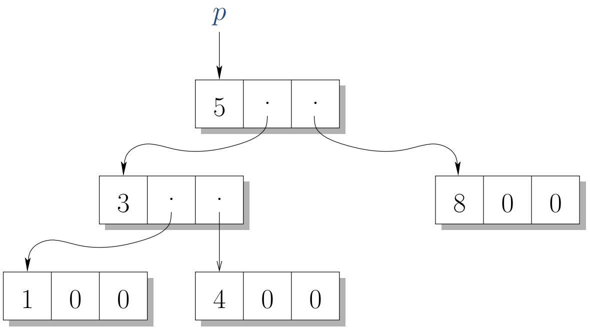
Define a shorthand ‘overlap’ as follows:

$$\text{overlap}(p,q) \doteq \; p = q \vee p+1 = q \vee p = q+1$$

Use to state that all list elements are pairwise disjoint:

$$\begin{array}{rcl} \text{list-disjoint}(p,0) & \doteq & \text{TRUE}\,, \\ \text{list-disjoint}(p,l) & \doteq & \text{list-disjoint}(p,l-1) \wedge \\ \multicolumn{3}{l}{\quad \forall 0 \le i < l-1.\ \neg\text{overlap}(\text{list-elem}(p,i), \text{list-elem}(p,l-1))} \end{array}$$

Grows quadratically in l!



Goal: model binary search tree

- Pointer to the left-hand child: l
- Pointer to the right-hand child: r

Idea:

$$\begin{array}{ll} & (n.l \neq \mathrm{NULL} \longrightarrow n.l\text{->}x < n.x) \\ \wedge & (n.r \neq \mathrm{NULL} \longrightarrow n.r\text{->}x > n.x)\ . \end{array}$$

Idea:

$$\begin{array}{ll} & (n.l \neq \mathrm{NULL} \longrightarrow n.l\text{->}x < n.x) \\ \wedge & (n.r \neq \mathrm{NULL} \longrightarrow n.r\text{->}x > n.x)\ . \end{array}$$

Not strong enough for $O(h)$ lookup!

Let us first define the transitive closure of a relation R:

$$
\begin{aligned}
\mathit{TC}^1_R(p,q) &\doteq R(p,q) \\
\mathit{TC}^i_R(p,q) &\doteq \exists p'.\ \mathit{TC}^{i-1}_R(p,p') \wedge R(p',q) \\
\mathit{TC}(p,q) &\doteq \exists i.\ \mathit{TC}^i_R(p,q)
\end{aligned}
$$

Now define a predicate tree-reach(p, q):

$$\begin{aligned}\text{tree-reach}(p, q) \quad &\doteq \quad p \neq \text{NULL} \wedge q \neq \text{NULL} \wedge \\ &\quad\quad (p = q \vee p\text{->}l = q \vee p\text{->}r = q)\end{aligned}$$

Use the transitive closure:

$$\text{tree-reach*}(p, q) \iff \text{TC}_{\text{tree-reach}(p,q)}$$

New definition:

$$
\begin{array}{ll}
& (\forall p.\ \textsf{tree-reach*}(n.l, p) \longrightarrow p\text{->}x < n.x) \\
\wedge & (\forall p.\ \textsf{tree-reach*}(n.r, p) \longrightarrow p\text{->}x > n.x)\ .
\end{array}
$$

Using the Semantic Translation

$[\![\cdot]\!]$ is a decision procedure!

1. Define $\varphi' \doteq [\![\varphi]\!]$
2. Pass φ' to procedure for integers and arrays

Example I

Let x be a variable, and p be a pointer.

$$p = \&x \wedge x = 1 \longrightarrow *p = 1$$

Example I

Let x be a variable, and p be a pointer.

$$p = \&x \wedge x = 1 \longrightarrow *p = 1$$

Use semantic definition:

$$\begin{aligned}
&[\![p = \&x \wedge x = 1 \longrightarrow *p = 1]\!] \\
&\quad \Longleftrightarrow \quad [\![p = \&x]\!] \wedge [\![x = 1]\!] \longrightarrow [\![*p = 1]\!] \\
&\quad \Longleftrightarrow \quad [\![p]\!] = [\![\&x]\!] \wedge [\![x]\!] = 1 \longrightarrow [\![*p]\!] = 1 \\
&\quad \Longleftrightarrow \quad M[L[p]] = L[x] \wedge M[L[x]] = 1 \longrightarrow M[M[L[p]]] = 1\ .
\end{aligned}$$

The last formula is obviously valid.

Example II

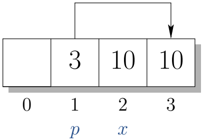
$$
\begin{aligned}
&[\![p \hookrightarrow x \longrightarrow p = \&x]\!] \\
&\qquad \Longleftrightarrow \quad [\![p \hookrightarrow x]\!] \longrightarrow [\![p = \&x]\!] \\
&\qquad \Longleftrightarrow \quad [\![*p = x]\!] \longrightarrow [\![p]\!] = [\![\&x]\!] \\
&\qquad \Longleftrightarrow \quad [\![*p]\!] = [\![x]\!] \longrightarrow M[L[p]] = L[x] \\
&\qquad \Longleftrightarrow \quad M[M[L[p]]] = M[L[x]] \longrightarrow M[L[p]] = L[x]
\end{aligned}
$$

Example II

$$\begin{aligned}
&\llbracket p \hookrightarrow x \longrightarrow p = \&x \rrbracket \\
&\quad \Longleftrightarrow \quad \llbracket p \hookrightarrow x \rrbracket \longrightarrow \llbracket p = \&x \rrbracket \\
&\quad \Longleftrightarrow \quad \llbracket *p = x \rrbracket \longrightarrow \llbracket p \rrbracket = \llbracket \&x \rrbracket \\
&\quad \Longleftrightarrow \quad \llbracket *p \rrbracket = \llbracket x \rrbracket \longrightarrow M[L[p]] = L[x] \\
&\quad \Longleftrightarrow \quad M[M[L[p]]] = M[L[x]] \longrightarrow M[L[p]] = L[x]
\end{aligned}$$

Counterexample:

$$L[p] = 1,\ L[x] = 2,\ M[1] = 3,\ M[2] = 10,\ M[3] = 10$$



What if the formula relies on a memory model axiom?

Example:

$$\sigma(x) = 2 \longrightarrow \&y \neq \&x + 1$$

The semantic translation yields:

$$\sigma(x) = 2 \longrightarrow L[y] \neq L[x] + 1$$

This needs the no-overlapping axiom:

$$\{L[x], \ldots, L[x] + \sigma(x) - 1\} \cap \{L[y], \ldots, L[y] + \sigma(y) - 1\} = \emptyset$$

Applying the Memory Model Axioms

1. Transform into linear arithmetic over the integers as follows:

$$(L[x] + \sigma(x) - 1 < L[y]) \lor (L[x] > L[y] + \sigma(y) - 1)$$

2. Using $\sigma(x) = 2$ and $\sigma(y) \geq 1$:

$$(L[x] + 1 < L[y]) \lor (L[x] > L[y])$$

3. Now strong enough to imply $L[y] \neq L[x] + 1$

Pure Variables

$$\begin{aligned}
&[\![x = y \longrightarrow y = x]\!] \\
&\qquad \iff \quad [\![x = y]\!] \longrightarrow [\![y = x]\!] \\
&\qquad \iff \quad M[L[x]] = M[L[y]] \longrightarrow M[L[y]] = M[L[x]]\ .
\end{aligned}$$

Unnecessary burden for the array decision procedure!

Pure Variables

$$
\begin{aligned}
& [\![x = y \longrightarrow y = x]\!] \\
& \qquad \iff \quad [\![x = y]\!] \longrightarrow [\![y = x]\!] \\
& \qquad \iff \quad M[L[x]] = M[L[y]] \longrightarrow M[L[y]] = M[L[x]] \; .
\end{aligned}
$$

Unnecessary burden for the array decision procedure!

Should have done:

$$x = y \longrightarrow y = x$$

Pure Variables

Obvious idea:

if the address of a variable x is not referred to,
translate it to a new variable Υ_x instead of $M[L[x]]$

Observation: the run time of a decision procedure for array logic depends on the number of different expressions that are used to index a particular array

Example

$$*p = 1 \land *q = 1$$

This is

$$M[\Upsilon_p] = 1 \land M[\Upsilon_q] = 1$$

- p and q might alias
- But there is no reason why they have to!

- Let's assume they don't!

Partitioning the Memory

We partition M into M_1 and M_2:

$$M_1[\Upsilon_p] = 1 \wedge M_2[\Upsilon_q] = 1$$

- This increases the number of array variables
- But: the number of different indices *per array* decreases!
- Typically improves performance

Cannot always be applied:

$$p = q \longrightarrow *p = *q$$

- Obviously valid
- If we partition as before, the translated formula is no longer valid:

$$\Upsilon_p = \Upsilon_q \longrightarrow M_1[\Upsilon_p] = M_2[\Upsilon_q]$$

A Partitioning Heuristic

- Deciding if the optimization is applicable is in general as hard as deciding φ itself

$\longrightarrow$ Do an approximation based on a syntactic test

- Deciding if the optimization is applicable is in general as hard as deciding φ itself

$\longrightarrow$ Do an approximation based on a syntactic test

Definition

Two pointer expressions p and q are *related* if both p and q are used inside the same relational expression

Write $p \approx q$ for TC_{related}

Partition according to $\approx$!