Pointers
Chapter 8

Decision Procedures
An Algorithmic Point of View

D. Kroening  O. Strichman  
Revision 1.0
Outline

1. Introduction
   - Pointers and Their Applications
   - Dynamic Memory Allocation
   - Analysis of Programs with Pointers

2. A Simple Pointer Logic
   - Syntax
   - Semantics
   - Axiomatization of the Memory Model
   - Adding Structure Types

3. Modeling
   - Lists
   - Trees

4. Using the Semantic Translation
   - Applying the Memory Model Axioms
   - Pure Variables
   - Partitioning the Memory
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.
Motivation

- 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
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.
**Formalization**

**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** $M : A \rightarrow D$

(this is a continuous, uniform address space)
Arrays and Structs

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* $L : V \rightarrow A$ 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).
```c
int var_a, var_b, var_c;
struct {
    int x;
    int y;
} S;
int array[4];
int *p = &var_c;

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

### Memory Layout:

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>var_a</td>
</tr>
<tr>
<td>1</td>
<td>var_b</td>
</tr>
<tr>
<td>2</td>
<td>var_c</td>
</tr>
<tr>
<td>3</td>
<td>S.x</td>
</tr>
<tr>
<td>4</td>
<td>S.y</td>
</tr>
<tr>
<td>5</td>
<td>array[0]</td>
</tr>
<tr>
<td>6</td>
<td>array[1]</td>
</tr>
<tr>
<td>7</td>
<td>array[2]</td>
</tr>
<tr>
<td>8</td>
<td>array[3]</td>
</tr>
<tr>
<td>9</td>
<td>p</td>
</tr>
</tbody>
</table>
Dynamic Memory Allocation

- 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.
Program analysis tools often need to reason about pointers

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

    for (i=0; i<10; i++)
        *sum = *sum + array[i];
}
```
Program analysis tools often need to reason about pointers

```c
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.
A Simple Pointer Logic

### Definition (Pointer Logic)

**Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>formula</code></td>
<td>`formula ∧ formula</td>
</tr>
<tr>
<td><code>atom</code></td>
<td>`pointer = pointer</td>
</tr>
<tr>
<td></td>
<td>`pointer &lt; pointer</td>
</tr>
<tr>
<td><code>pointer</code></td>
<td>`pointer − identifier</td>
</tr>
<tr>
<td></td>
<td>`&amp;identifier</td>
</tr>
<tr>
<td><code>term</code></td>
<td>`identifier</td>
</tr>
<tr>
<td></td>
<td>`integer − constant</td>
</tr>
<tr>
<td><code>op</code></td>
<td>`+</td>
</tr>
</tbody>
</table>

**Warning:** `=` is equality here, not assignment
Let \( p, q \) denote pointer identifiers, and let \( i, j \) denote integer identifiers.

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

- \( *(p + i) = 1 \),
- \( *(p + *p) = 0 \),
- \( p = q \land *p = 5 \),
- \( ** ** *p = 1 \),
- \( p < q \).
The following formulas are not permitted by the grammar:

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

\[
\begin{align*}
[f_1 \land f_2] &= \llbracket f_1 \rrbracket \land \llbracket f_2 \rrbracket \\
[\neg f] &= \neg \llbracket f \rrbracket
\end{align*}
\]

Predicates:

\[
\begin{align*}
[p_1 = p_2] &= \llbracket p_1 \rrbracket = \llbracket p_2 \rrbracket & \text{where } p_1, p_2 \text{ are pointer expressions} \\
[p_1 < p_2] &= \llbracket p_1 \rrbracket < \llbracket p_2 \rrbracket & \text{where } p_1, p_2 \text{ are pointer expressions} \\
[t_1 = t_2] &= \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket & \text{where } t_1, t_2 \text{ are terms} \\
[t_1 < t_2] &= \llbracket t_1 \rrbracket < \llbracket t_2 \rrbracket & \text{where } t_1, t_2 \text{ are terms}
\end{align*}
\]
Non-pointer terms:

\[
\begin{align*}
[v] & = M[L[v]] & \text{where } v \in V \text{ is a variable with } \sigma(v) = 1 \\
[t_1 \text{ op } t_2] & = [t_1] \text{ op } [t_2] & \text{where } t_1, t_2 \text{ are terms} \\
[c] & = c & \text{where } c \text{ is an integer constant} \\
[v[t]] & = M[L[v] + [t]] & \text{where } v \text{ is an array identifier, } t \text{ is a term}
\end{align*}
\]
Pointer-related expressions:

\[
\begin{align*}
[p] & = M[L[p]] & \text{where } p \text{ is a pointer identifier} \\
[p + t] & = [p] + [t] & \text{where } p \text{ is a pointer expression, } t \text{ is a term} \\
& [\& v] & = L[v] & \text{where } v \in V \text{ is a variable} \\
& [\& * p] & = [p] & \text{where } p \text{ is a pointer expression} \\
\text{NULL} & = 0 \\
*p & = M[[p]] & \text{where } p \text{ is a pointer expression}
\end{align*}
\]
A pointer $p$ points to a variable $x$ if $M[L[p]] = L[x]$

Shorthand: $p \rightarrow z$ for $*p = z$

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

Let $a$ be an array identifier:

$$\ast((\&a) + 1) = a[1]$$

The definition expands as follows:

$$[\ast((\&a) + 1) = a[1]]$$
Example I

Let \( a \) be an array identifier:

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

The definition expands as follows:

\[
\llbracket \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \! \!
Example I

Let $a$ be an array identifier:

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

The definition expands as follows:

$$[*((&a) + 1) = a[1]] \iff [*[(&a) + 1)] = [a[1]]$$
$$\iff M[[(&a) + 1]] = M[L[a] + [1]]$$

The last formula is obviously valid.
Example I

Let $a$ be an array identifier:

$$\ast((\&a) + 1) = a[1]$$

The definition expands as follows:

$$\llbracket\ast((\&a) + 1) = a[1]\rrbracket \iff \llbracket\ast((\&a) + 1)\rrbracket = \llbracket a[1]\rrbracket$$

$$\iff M[\llbracket(\&a) + 1\rrbracket] = M[L[a] + 1]$$

The last formula is obviously valid.
Let \( a \) be an array identifier:

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

The definition expands as follows:

\[
\begin{align*}
\left[ *((\&a) + 1) = a[1] \right] & \iff \left[ *((\&a) + 1) \right] = \llbracket a[1] \rrbracket \\
& \iff M[\llbracket (\&a) + 1 \rrbracket] = M[L[a] + \llbracket 1 \rrbracket] \\
& \iff M[\llbracket \&a \rrbracket + \llbracket 1 \rrbracket] = M[L[a] + 1] \\
& \iff M[L[a] + 1] = M[L[a] + 1]
\end{align*}
\]

The last formula is obviously valid.
The translated formula must evaluate to true for any $L$ and $M$!

The following formula is not valid:

$$*p = 1 \quad \rightarrow \quad x = 1$$

For $p \neq &x$, this formula evaluates to false.
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.
On most architectures, the following two formulas are valid:

1. \( \&x \neq \text{NULL} \)
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”)

\[ \forall v \in V. \ L[v] \neq 0 \]
Overlapping Objects

How do we address (2)?

Suggestion:

\[ \forall v_1, v_2 \in V. \ v_1 \neq v_2 \rightarrow L[v_1] \neq L[v_2] \]
Memory Axioms 2 and 3

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")

\[ \forall v_1, v_2 \in V. v_1 \neq v_2 \quad \rightarrow \quad \{L[v_1], \ldots, L[v_1] + \sigma(v_1) - 1\} \cap \{L[v_2], \ldots, L[v_2] + \sigma(v_2) - 1\} = \emptyset. \]
Some code relies on additional, architecture-specific guarantees, e.g.,

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

Some program analysis tools allow adding such rules.
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 $s$
Mapping to Array Types

- Each field of the structure is assigned a unique offset: \( o(f) \)
- Meaning of \( s.f \):
  \[
  s.f \equiv \star((\&s) + o(f))
  \]
- Following PASCAL and ANSI-C syntax:
  \[
  p\rightarrow f \equiv (\star p).f
  \]
- Adopted from separation logic:
  \[
  p \leftarrow a, b, c, \ldots \quad \equiv \quad \star (p + 0) = a \quad \land \\
  \star (p + 1) = b \quad \land \\
  \star (p + 2) = c \quad \ldots .
  \]
Modeling Lists

- Simplest dynamically allocated data structure

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

\[
p \rightarrow 't', p_1 \\
\land p_1 \rightarrow 'e', p_2 \\
\land p_2 \rightarrow 'x', p_3 \\
\land p_3 \rightarrow 't', \text{NULL} .
\]
Define a recursive shorthand for the \( i \)-th member of a list:

\[
\begin{align*}
\text{list-elem}(p, 0) & \triangleq p, \\
\text{list-elem}(p, i) & \triangleq \text{list-elem}(p, i - 1) \rightarrow n \quad \text{for } i \geq 1
\end{align*}
\]

We use '\( n \)' as the next pointer field.

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

\[
\text{list}(p, l) \triangleq \text{list-elem}(p, l) = \text{NULL}
\]
A linked list is **cyclic** if the pointer of the last element points to the first one:

```
. . . .
  e
  t
  t
  p
```

Would this work?

```
my-list (p, l) = list-elem (p, l) = p
```
A linked list is *cyclic* if the pointer of the last element points to the first one:

Would this work?

\[
\text{my-list}(p, l) \doteq \text{list-elem}(p, l) = p.
\]
Unfortunately, the following satisfies $\text{my-list}(p, 4)$:

\[ p \rightarrow 't' \]

→ need to rule out sharing
Cyclic Lists

Define a shorthand ‘overlap’ as follows:

\[
\text{overlap}(p, q) \equiv p = q \lor p + 1 = q \lor p = q + 1
\]

Use to state that all list elements are pairwise disjoint:

\[
\text{list-disjoint}(p, 0) \equiv \text{TRUE},
\]
\[
\text{list-disjoint}(p, l) \equiv \text{list-disjoint}(p, l - 1) \land \\
\forall 0 \leq i < l - 1. \neg \text{overlap} (\text{list-elem}(p, i), \text{list-elem}(p, l - 1))
\]

Grows quadratically in \( l \)!
Goal: model binary search tree

- Pointer to the left-hand child: \( l \)
- Pointer to the right-hand child: \( r \)
Idea:

\[(n.l \neq \text{NULL} \rightarrow n.l \rightarrow x < n.x) \land (n.r \neq \text{NULL} \rightarrow n.r \rightarrow x > n.x)\].
Idea:

\[
(n.l \neq \text{NULL} \rightarrow n.l\rightarrow x < n.x) \\
\land (n.r \neq \text{NULL} \rightarrow n.r\rightarrow x > n.x)
\] .

Not strong enough for \(O(h)\) lookup!
Let us first define the transitive closure of a relation \( R \):

\[
\begin{align*}
TC^1_R(p, q) & \equiv R(p, q) \\
TC^i_R(p, q) & \equiv \exists p'. \ TC^{-1}_R(p', p') \land R(p', q) \\
TC(p, q) & \equiv \exists i. \ TC^i_R(p, q)
\end{align*}
\]
Now define a predicate $\text{tree-reach}(p, q)$:

$$\text{tree-reach}(p, q) \doteq p \neq \text{NULL} \land q \neq \text{NULL} \land (p = q \lor p\rightarrow l = q \lor p\rightarrow r = q)$$

Use the transitive closure:

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

\[(\forall p. \text{tree-reach}^*(n.l, p) \rightarrow p \rightarrow x < n.x) \land \ (\forall p. \text{tree-reach}^*(n.r, p) \rightarrow p \rightarrow x > n.x)\].
Using the Semantic Translation

$\llbracket \cdot \rrbracket$ is a decision procedure!

1. Define $\varphi' = \llbracket \varphi \rrbracket$
2. Pass $\varphi'$ to procedure for integers and arrays
Let $x$ be a variable, and $p$ be a pointer.

\[ p = \&x \land x = 1 \longrightarrow *p = 1 \]
Example I

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

$p = \&x \land x = 1 \implies *p = 1$

Use semantic definition:

$[[p = \&x \land x = 1 \implies *p = 1]]$

$\iff [[p = \&x]] \land [[x = 1]] \implies [[*p = 1]]$

$\iff [[p]] = [[\&x]] \land [[x]] = 1 \implies [[*p]] = 1$

$\iff M[L[p]] = L[x] \land M[L[x]] = 1 \implies M[M[L[p]]] = 1.$

The last formula is obviously valid.
Example II

\[ [p \leftarrow x \rightarrow p = \&x] \]

\[ \iff \quad [p \leftarrow x] \rightarrow [p = \&x] \]

\[ \iff \quad [*p = x] \rightarrow [p] = [\&x] \]

\[ \iff \quad [*p] = [x] \rightarrow M[L[p]] = L[x] \]

\[ \iff \quad M[M[L[p]]] = M[L[x]] \rightarrow M[L[p]] = L[x] \]
Example II

\[[p \leftarrow x \longrightarrow p = &x]\]
\[\iff [p \leftarrow x] \longrightarrow [p = &x]\]
\[\iff [*p = x] \longrightarrow [p] = [&x]\]
\[\iff [*p] = [x] \longrightarrow M[L[p]] = L[x]\]
\[\iff M[M[L[p]]] = M[L[x]] \longrightarrow M[L[p]] = L[x]\]

Counterexample:

What if the formula relies on a memory model axiom?

Example:

\[
\sigma(x) = 2 \implies \&y \neq \&x + 1
\]

The semantic translation yields:

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

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 \(L[y] \neq L[x] + 1\)
\[ [x = y \longrightarrow y = x] \]
\[ \iff [x = y] \longrightarrow [y = x] \]
\[ \iff M[L[x]] = M[L[y]] \longrightarrow M[L[y]] = M[L[x]] . \]

Unnecessary burden for the array decision procedure!
Pure Variables

\[ [x = y \rightarrow y = x] \]
\[ \iff [x = y] \rightarrow [y = x] \]
\[ \iff M[L[x]] = M[L[y]] \rightarrow M[L[y]] = M[L[x]] . \]

Unnecessary burden for the array decision procedure!

Should have done:

\[ x = y \rightarrow y = x \]
Obvious idea:

if the address of a variable $x$ is not referred to, translate it to a new variable $\Upsilon_x$ 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

\[ M[\U p] = 1 \land M[\U q] = 1 \]

- \( p \) and \( q \) might alias
- But there is no reason why they have to!
- Let’s assume they don’t!
We partition $M$ into $M_1$ and $M_2$:

$$M_1[γ_p] = 1 \land M_2[γ_q] = 1$$

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

Cannot always be applied:

\[ p = q \implies *p = *q \]

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

\[ \Upsilon_p = \Upsilon_q \implies M_1[\Upsilon_p] = M_2[\Upsilon_q] \]
Deciding if the optimization is applicable is in general as hard as deciding $\varphi$ itself

$\rightarrow$ Do an approximation based on a syntactic test
Deciding if the optimization is applicable is in general as hard as deciding $\varphi$ itself.

→ 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 $\approx$!