How C differs from Java for Symbolic Program Execution

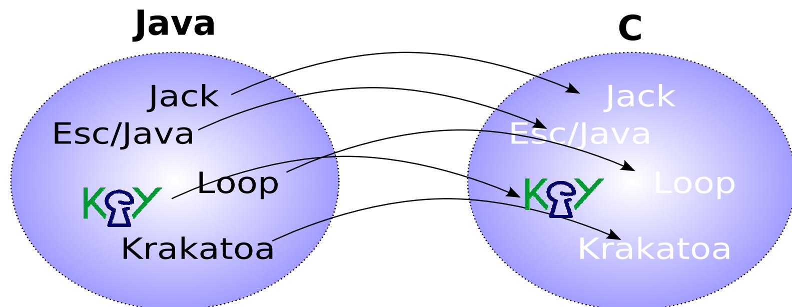
Christoph Gladisch

www.key-project.org

C/C++ Verification Workshop 2007

Oxford, United Kingdom
July 2, 2007

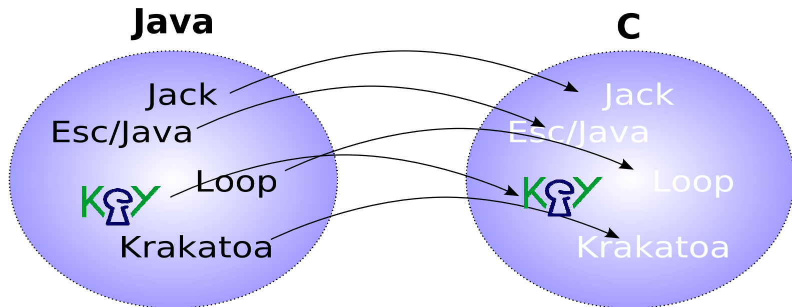
Motivation



KeY System

- Automatic and Interactive
- Verification System and Test Generator
- 100% JavaCard

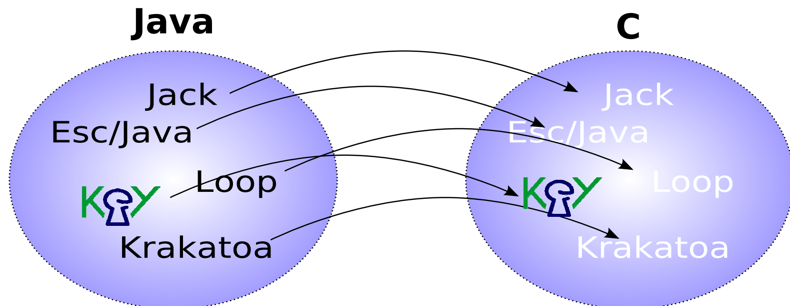
Motivation



KeY System

- Automatic and Interactive
- Verification System and Test Generator
- 100% JavaCard

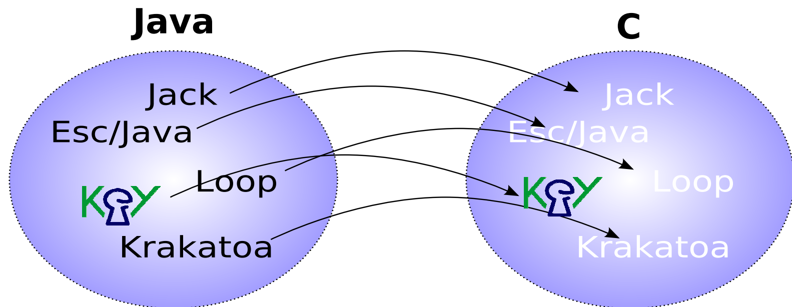
Motivation



KeY System

- Automatic and Interactive
- Verification System and Test Generator
- 100% JavaCard

Motivation



KeY System

- Automatic and Interactive
- Verification System and Test Generator
- 100% JavaCard

Level of Comparison

Equivalent

C	$\longleftrightarrow$	Java
if(..), a=b, m(s,a)		if(..), a=b, s.m(a)
+,-,*,/		+,-,*,/

Different

if((int) a)		if(..)
a= &b		a=m() //m() throws Ex
range(int)	$\neq$	range(int)

Level of Comparison

Equivalent

C	$\longleftrightarrow$	Java
if(..), a=b, m(s,a)		if(..), a=b, s.m(a)
+,-,*,/		+,-,*,/

Different

if((int) a)		if(..)
a= &b		a=m() //m() throws Ex
range(int)	$\neq$	range(int)

Level of Comparison

Equivalent

C	$\longleftrightarrow$	Java
if(..), a=b, m(s,a)		if(..), a=b, s.m(a)
+,-,*,/		+,-,*,/

Different

if((int) a)		if(..)
a= &b		a=m() //m() throws Ex
range(int)	$\neq$	range(int)

Level of Comparison

Equivalent

C	$\longleftrightarrow$	Java
if(..), a=b, m(s,a)		if(..), a=b, s.m(a)
+,-,*,/		+,-,*,/

Different

if((int) a)		if(..)
a= &b		a=m() //m() throws Ex
range(int)	$\neq$	range(int)

Level of Comparison

Equivalent

C	$\longleftrightarrow$	Java
if(..), a=b, m(s,a)		if(..), a=b, s.m(a)
+,-,*,/		+,-,*,/

Different

if((int) a)		if(..)
a= &b		a=m() //m() throws Ex
range(int)	$\neq$	range(int)

Level of Comparison

Equivalent

C	⟷	Java
if(..), a=b, m(s,a)		if(..), a=b, s.m(a)
+,-,*,/		+,-,*,/

Different

if((int) a)		if(..)
a= &b		a=m() //m() throws Ex
range(int)	$\neq$	range(int)

Differences Overview

- reference and dereference operators (ANSI C, MISRA C, C0)
- assignment of structures (ANSI C, MISRA C, C0)
- evaluation order of expressions and function arguments (ANSI C only)
- explicit object deallocation (ANSI C only)
- unions, pointer arithmetics, and type casts (ANSI C only)
- goto-statement (ANSI C only)

ANSI C

The Specification of ANSI C ...

- ...gives room for interpretation and customization of a C implementation.
  Undefined-, Unspecified-, Implementation-defined- behavior
- ...is hard to understand in detail.
  Books on C often contain mistakes due to misunderstanding of the Spec.

ANSI C

The Specification of ANSI C ...

- ...gives room for interpretation and customization of a C implementation.
  Undefined-, Unspecified-, Implementation-defined- behavior
- ...is hard to understand in detail.
  Books on C often contain mistakes due to misunderstanding of the Spec.

MISRA C

- Restrictions on the construction of expressions (evaluation order, types)
- No direct or indirect recursive functions
- Single point of exist at the end of the function
- No dynamic memory (allocation, deallocation)
- Pointerarithmethic restricted to use for array indexing
- No goto, continue, comma-operator, unions

MISRA C

- Restrictions on the construction of expressions (evaluation order, types)
- No direct or indirect recursive functions
- Single point of exist at the end of the function
- No dynamic memory (allocation, deallocation)
- Pointerarithmethic restricted to use for array indexing
- No goto, continue, comma-operator, unions

MISRA C

- Restrictions on the construction of expressions (evaluation order, types)
- No direct or indirect recursive functions
- Single point of exist at the end of the function
- No dynamic memory (allocation, deallocation)
- Pointerarithmethic restricted to use for array indexing
- No goto, continue, comma-operator, unions

MISRA C

- Restrictions on the construction of expressions (evaluation order, types)
- No direct or indirect recursive functions
- Single point of exist at the end of the function
- No dynamic memory (allocation, deallocation)
- Pointerarithmethic restricted to use for array indexing
- No goto, continue, comma-operator, unions

MISRA C

- Restrictions on the construction of expressions (evaluation order, types)
- No direct or indirect recursive functions
- Single point of exist at the end of the function
- No dynamic memory (allocation, deallocation)
- Pointerarithmethic restricted to use for array indexing
- No goto, continue, comma-operator, unions

MISRA C

- Restrictions on the construction of expressions (evaluation order, types)
- No direct or indirect recursive functions
- Single point of exist at the end of the function
- No dynamic memory (allocation, deallocation)
- Pointerarithmethic restricted to use for array indexing
- No goto, continue, comma-operator, unions

C0 Overview

- No side effects in expression, no ambiguities of evaluation
- Size of arrays is staticaly fixed
- No pointers to local memories
- Two's complement representation
- Strictly typed

C0 Overview

- No side effects in expression, no ambiguities of evaluation
- Size of arrays is staticaly fixed
- No pointers to local memories
- Two's complement representation
- Strictly typed

C0 Overview

- No side effects in expression, no ambiguities of evaluation
- Size of arrays is staticaly fixed
- No pointers to local memories
- Two's complement representation
- Strictly typed

C0 Overview

- No side effects in expression, no ambiguities of evaluation
- Size of arrays is staticaly fixed
- No pointers to local memories
- Two's complement representation
- Strictly typed

C0 Overview

- No side effects in expression, no ambiguities of evaluation
- Size of arrays is staticaly fixed
- No pointers to local memories
- Two's complement representation
- Strictly typed

Dynamic Logic - Details

Program-modal-operators + FOL = DL

$\langle \texttt{p} \rangle \phi \qquad \{a := b..\}\phi$

Program Variable = non-rigid Function Symbol

$$(prog.var.)\ \texttt{a} = a \text{ (logic const.)}$$
$$\texttt{o.a} = a(o)$$

Updates

$$\langle \texttt{o.a = t;} \rangle \phi \rightsquigarrow \{a(o) := t\}\phi$$
$$\langle \texttt{o.a = t;u.b = s} \rangle \phi \rightsquigarrow \{a(o) := t \,||\, b(u') := s'\}\phi$$

Replace $\langle \texttt{prog} \rangle \phi$ **by** $\{upd\}\phi$

Dynamic Logic - Details

Program-modal-operators + FOL = DL

$\langle \texttt{p} \rangle \phi \qquad \{a := b..\}\phi$

Program Variable = non-rigid Function Symbol

$$(prog.var.)\ \texttt{a} = a \text{ (logic const.)}$$
$$\texttt{o.a} = a(o)$$

Updates

$$\langle \texttt{o.a = t;} \rangle \phi \rightsquigarrow \{a(o) := t\}\phi$$
$$\langle \texttt{o.a = t;u.b = s} \rangle \phi \rightsquigarrow \{a(o) := t \,||\, b(u') := s'\}\phi$$

Replace $\langle \texttt{prog} \rangle \phi$ **by** $\{upd\}\phi$

Dynamic Logic - Details

Program-modal-operators + FOL = DL

$\langle \texttt{p} \rangle \phi$ $\{a := b..\}\phi$

Program Variable = non-rigid Function Symbol

$(prog.var.)$ `a`	=	a (logic const.)
`o.a`	=	$a(o)$

Updates

$\langle$`o.a = t;`$\rangle\phi$	$\rightsquigarrow$	$\{a(o) := t\}\phi$
$\langle$`o.a = t;u.b = s`$\rangle\phi$	$\rightsquigarrow$	$\{a(o) := t \mid\mid b(u') := s'\}\phi$
Replace $\langle$`prog`$\rangle\phi$	**by**	$\{upd\}\phi$

Dynamic Logic - Details

Program-modal-operators + FOL = DL

$$\langle \texttt{p} \rangle \phi \qquad \{a := b..\}\phi$$

Program Variable = non-rigid Function Symbol

$$(\textit{prog.var.})\ \texttt{a} \quad = \quad a \text{ (logic const.)}$$
$$\texttt{o.a} \quad = \quad a(o)$$

Updates

$$\langle \texttt{o.a = t;} \rangle \phi \quad \rightsquigarrow \quad \{a(o) := t\}\phi$$
$$\langle \texttt{o.a = t;u.b = s} \rangle \phi \quad \rightsquigarrow \quad \{a(o) := t \,||\, b(u') := s'\}\phi$$

Replace $\langle \texttt{prog} \rangle \phi$ **by** $\{upd\}\phi$

Dynamic Logic - Details

Program-modal-operators + FOL = DL

$$\langle \texttt{p} \rangle \phi \qquad \{a := b..\}\phi$$

Program Variable = non-rigid Function Symbol

$$(\textit{prog.var.})\ \texttt{a} \quad = \quad a \text{ (logic const.)}$$
$$\texttt{o.a} \quad = \quad a(o)$$

Updates

$$\langle \texttt{o.a = t;} \rangle \phi \quad \rightsquigarrow \quad \{a(o) := t\}\phi$$
$$\langle \texttt{o.a = t;u.b = s} \rangle \phi \quad \rightsquigarrow \quad \{a(o) := t \,||\, b(u') := s'\}\phi$$

Replace $\langle \texttt{prog} \rangle \phi$ **by** $\{upd\}\phi$

Dynamic Logic - Details

Program-modal-operators + FOL = DL

$$\langle \mathtt{p} \rangle \phi \qquad \{a := b..\}\phi$$

Program Variable = non-rigid Function Symbol

$$(\mathit{prog.var.})\ \mathtt{a} \quad = \quad a \text{ (logic const.)}$$

$$\mathtt{o.a} \quad = \quad a(o)$$

Updates

$$\langle \mathtt{o.a = t;} \rangle \phi \quad \rightsquigarrow \quad \{a(o) := t\}\phi$$

$$\langle \mathtt{o.a = t;u.b = s} \rangle \phi \quad \rightsquigarrow \quad \{a(o) := t \,||\, b(u') := s'\}\phi$$

Replace $\langle \mathtt{prog} \rangle \phi$ **by** $\{upd\}\phi$

Dynamic Logic - Details

Program-modal-operators + FOL = DL

$$\langle \texttt{p} \rangle \phi \qquad \{a := b..\}\phi$$

Program Variable = non-rigid Function Symbol

$$\begin{array}{rcl} (\textit{prog.var.})\ \texttt{a} & = & a \text{ (logic const.)} \\ \texttt{o.a} & = & a(o) \end{array}$$

Updates

$$\begin{array}{rcl} \langle \texttt{o.a = t;} \rangle \phi & \rightsquigarrow & \{a(o) := t\}\phi \\ \langle \texttt{o.a = t;u.b = s} \rangle \phi & \rightsquigarrow & \{a(o) := t \,||\, b(u') := s'\}\phi \\ \textbf{Replace} \ \langle \texttt{prog} \rangle \phi & \textbf{by} & \{upd\}\phi \end{array}$$

Differences Overview

- reference and dereference operators (ANSI C, MISRA C, C0)
- assignment of structures (ANSI C, MISRA C, C0)
- evaluation order of expressions and function arguments (ANSI C only)
- explicit object deallocation (ANSI C only)
- unions, pointer arithmetics, and type casts (ANSI C only)
- goto-statement (ANSI C only)

Reference and Dereference Operators (ANSI C, MISRA C, C0)

Indirect access

- $\langle$`p=&a; *p=1;`$\rangle a = 1$
- $\{*(p) := a\}\{*(*(p)) := 1\} * (a) = 1$

Leverage program variables to object level

- a Pointer to the program variable "a"
- $* : Pointer \rightarrow Value$

Reference and Dereference Operators (ANSI C, MISRA C, C0)

Indirect access

- $\langle$`p=&a; *p=1;`$\rangle a = 1$
- $\{*(p) := a\}\{*(*(p)) := 1\} * (a) = 1$

Leverage program variables to object level

- a Pointer to the program variable "a"
- $* : \mathit{Pointer} \rightarrow \mathit{Value}$

Reference and Dereference Operators (ANSI C, MISRA C, C0)

Indirect access

- $\langle$`p=&a; *p=1;`$\rangle a = 1$
- $\{*(p) := a\}\{*(*(p)) := 1\} * (a) = 1$

Leverage program variables to object level

- a Pointer to the program variable "a"
- $*$: *Pointer* $\longrightarrow$ *Value*

Reference and Dereference Operators (ANSI C, MISRA C, C0)

Indirect access

- $\langle \texttt{p=\&a; *p=1;} \rangle a = 1$
- $\{*(p) := a\}\{*(*(p)) := 1\} * (a) = 1$

Leverage program variables to object level

- a Pointer to the program variable “a”
- $* : \mathit{Pointer} \rightarrow \mathit{Value}$

Reference and Dereference Operators (ANSI C, MISRA C, C0)

Step 1: Expression to Pointer

$$\begin{aligned}
\text{etp}(\mathsf{a}) &\leadsto \mathsf{a} \\
\text{etp}(X.m) &\leadsto {}^{.m}(\text{etp}(X)) \\
\text{etp}(X[i]) &\leadsto {}^{[]}(\text{etp}(X), \text{ett}(i)) \\
\text{etp}(*X) &\leadsto \text{ett}(X)
\end{aligned}$$

Step 2: Dereference Pointer

$$\begin{aligned}
\text{ett}(X) &\leadsto {}^{*}(\text{etp}(X)) \\
\text{ett}(\&X) &\leadsto \text{etp}(X)
\end{aligned}$$

Expression	`a`	`&a`	`a.m`	`*a`	`(*a).m`
Term	${}^{*}(a)$	a	${}^{*}({}^{.m}(a))$	${}^{*}({}^{*}(a))$	${}^{*}({}^{.m}({}^{*}(a)))$

Expression	`*(a.m)`	`a[i]`
Term	${}^{*}({}^{*}({}^{.m}(a)))$	${}^{*}({}^{[]}(a, {}^{*}(i)))$

Reference and Dereference Operators (ANSI C, MISRA C, C0)

Step 1: Expression to Pointer

$$
\begin{aligned}
\text{etp}(\text{a}) &\rightsquigarrow \text{a} \\
\text{etp}(X.m) &\rightsquigarrow {}^{.m}(\text{etp}(X)) \\
\text{etp}(X[i]) &\rightsquigarrow {}^{[]}(\text{etp}(X), \text{ett}(i)) \\
\text{etp}(*X) &\rightsquigarrow \text{ett}(X)
\end{aligned}
$$

Step 2: Dereference Pointer

$$
\begin{aligned}
\text{ett}(X) &\rightsquigarrow {}^{*}(\text{etp}(X)) \\
\text{ett}(\&X) &\rightsquigarrow \text{etp}(X)
\end{aligned}
$$

Expression	`a`	`&a`	`a.m`	`*a`	`(*a).m`
Term	${}^{*}(a)$	a	${}^{*}({}^{.m}(a))$	${}^{*}({}^{*}(a))$	${}^{*}({}^{.m}({}^{*}(a)))$

Expression	`*(a.m)`	`a[i]`
Term	${}^{*}({}^{*}({}^{.m}(a)))$	${}^{*}({}^{[]}(a, {}^{*}(i)))$

Reference and Dereference Operators (ANSI C, MISRA C, C0)

Step 1: Expression to Pointer

$$\begin{aligned}
\mathrm{etp}(\mathtt{a}) &\leadsto \mathtt{a}\\
\mathrm{etp}(X.m) &\leadsto {}^{.m}(\mathrm{etp}(X))\\
\mathrm{etp}(X[i]) &\leadsto {}^{[]}(\mathrm{etp}(X), \mathrm{ett}(i))\\
\mathrm{etp}(*X) &\leadsto \mathrm{ett}(X)
\end{aligned}$$

Step 2: Dereference Pointer

$$\begin{aligned}
\mathrm{ett}(X) &\leadsto {}^{*}(\mathrm{etp}(X))\\
\mathrm{ett}(\&X) &\leadsto \mathrm{etp}(X)
\end{aligned}$$

Expression	`a`	`&a`	`a.m`	`*a`	`(*a).m`
Term	${}^{*}(a)$	a	${}^{*}({}^{.m}(a))$	${}^{*}({}^{*}(a))$	${}^{*}({}^{.m}({}^{*}(a)))$

Expression	`*(a.m)`	`a[i]`
Term	${}^{*}({}^{*}({}^{.m}(a)))$	${}^{*}({}^{[]}(a, {}^{*}(i)))$

Assignment of Structs (ANSI C, MISRA C, C0)

a,b∈C struct	a,b∈C struct**pointer**	a,b∈Java class

`b.m = d;`
`a = b;`
`b.m = e;`
	`b->m = d;`
	`a = b;`
	`b->m = e;`
		`b.m = d;`
		`a = b;`
		`b.m = e;`

$a.m =^{\cdot} d$	`a->m` $=^{\cdot} e$	$a.m =^{\cdot} e$

```
struct S {
int a,b,c,d,e;
int n[];
int m[100];
}s1,s2;
```

$$\frac{s1 = s2;}{\begin{array}{rcl} s1.a & = & s2.a \\ & \vdots & \\ s1.e & = & s2.e \\ s1.n & = & s2.n \\ s1.m[0] & = & s2.m[0] \\ & \vdots & \\ s1.m[99] & = & s2.m[99] \end{array}}$$

Assignment of Structs (ANSI C, MISRA C, C0)

a,b∈C struct	a,b∈C struct**pointer**	a,b∈Java class

`b.m = d;`
`a = b;`
`b.m = e;`
	`b->m = d;`
	`a = b;`
	`b->m = e;`
		`b.m = d;`
		`a = b;`
		`b.m = e;`

$a.m =^{\cdot} d$	`a->m` $=^{\cdot} e$	$a.m =^{\cdot} e$

```
struct S {
int a,b,c,d,e;
int n[];
int m[100];
}s1,s2;
```

$$\frac{s1 = s2;}{\begin{array}{rcl} s1.a & = & s2.a \\ & \vdots & \\ s1.e & = & s2.e \\ s1.n & = & s2.n \\ s1.m[0] & = & s2.m[0] \\ & \vdots & \\ s1.m[99] & = & s2.m[99] \end{array}}$$

Assignment of Structs (ANSI C, MISRA C, C0)

a,b∈C struct	a,b∈C struct**pointer**	a,b∈Java class

`b.m = d;`
`a = b;`
`b.m = e;`
	`b->m = d;`
	`a = b;`
	`b->m = e;`
		`b.m = d;`
		`a = b;`
		`b.m = e;`

$a.m =^{\cdot} d$	`a->m` $=^{\cdot} e$	$a.m =^{\cdot} e$

```
struct S {
int a,b,c,d,e;
int n[];
int m[100];
}s1,s2;
```

$$
\begin{array}{rcl}
s1 & = & s2; \\
\hline
s1.a & = & s2.a \\
 & \vdots & \\
s1.e & = & s2.e \\
s1.n & = & s2.n \\
s1.m[0] & = & s2.m[0] \\
 & \vdots & \\
s1.m[99] & = & s2.m[99]
\end{array}
$$

Assignment of Structs (ANSI C, MISRA C, C0)

a,b∈C struct	a,b∈C struct**pointer**	a,b∈Java class

`b.m = d;`
`a = b;`
`b.m = e;`
	`b->m = d;`
	`a = b;`
	`b->m = e;`
		`b.m = d;`
		`a = b;`
		`b.m = e;`

$a.m \doteq d$	`a->m` $\doteq e$	$a.m \doteq e$

```
struct S {
int a,b,c,d,e;
int n[];
int m[100];
}s1,s2;
```

$$\begin{array}{rcl} s1 & = & s2; \\ \hline s1.a & = & s2.a \\ & \vdots & \\ s1.e & = & s2.e \\ s1.n & = & s2.n \\ s1.m[0] & = & s2.m[0] \\ & \vdots & \\ s1.m[99] & = & s2.m[99] \end{array}$$

Evaluation order of Expressions and Function Arguments (ANSI C only)

Example

$\langle$`i = 1; i = 2* ++i + 3*i;`$\rangle i = 7$

$\langle$`i = 1; i = 2* ++i + 3*i;`$\rangle i = 10$

Rule

Replace $\langle$`stmt(expr_1,..,expr_n);`$\rangle$ **by**
$\langle$`tmp_1=expr_1;...;tmp_n=expr_n; stmt(tmp_1,...,tmp_n);`$\rangle$

Evaluation order of Expressions and Function Arguments (ANSI C only)

Example

⟨`i = 1; i = 2* ++i + 3*i;`⟩$i = 7$

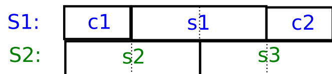
⟨`i = 1; i = 2* ++i + 3*i;`⟩$i = 10$

Rule

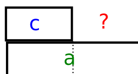
Replace ⟨`stmt(expr_1,..,expr_n);`⟩ **by**
⟨`tmp_1=expr_1;...;tmp_n=expr_n; stmt(tmp_1,...,tmp_n);`⟩

unions, pointer arithmetics, and type casts (ANSI C only)

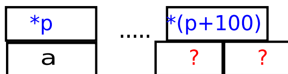
```
union U {
  struct S1{
    unsigned char c1;
    unsigned short s1;
    unsigned char c2;
  }s1;
  struct S2{
    unsigned short s2;
    unsigned short s3;
  }s2;
};
```




```
char c= -1;
short a= *((short*)&c);
```



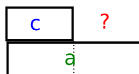

```
int* p=&a;
int b = *(p+100);
```



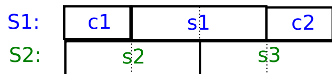
unions, pointer arithmetics, and type casts (ANSI C only)

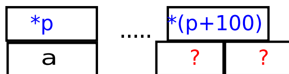
```
union U {
  struct S1{
    unsigned char c1;
    unsigned short s1;
    unsigned char c2;
  }s1;
  struct S2{
    unsigned short s2;
    unsigned short s3;
  }s2;
};
```

```
char c= -1;
short a= *((short*)&c);
```



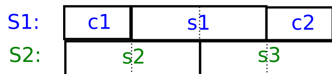

```
int* p=&a;
int b = *(p+100);
```



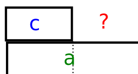


unions, pointer arithmetics, and type casts (ANSI C only)

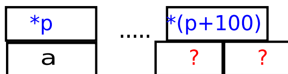
```
union U {
  struct S1{
    unsigned char c1;
    unsigned short s1;
    unsigned char c2;
  }s1;
  struct S2{
    unsigned short s2;
    unsigned short s3;
  }s2;
};
```




```
char c= -1;
short a= *((short*)&c);
```

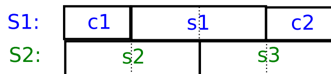



```
int* p=&a;
int b = *(p+100);
```

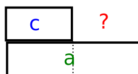


unions, pointer arithmetics, and type casts (ANSI C only)

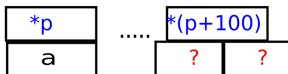
```
union U {
  struct S1{
    unsigned char c1;
    unsigned short s1;
    unsigned char c2;
  }s1;
  struct S2{
    unsigned short s2;
    unsigned short s3;
  }s2;
};
```




```
char c= -1;
short a= *((short*)&c);
```




```
int* p=&a;
int b = *(p+100);
```



Explicit object deallocation (ANSI C only)

Example Dangling Pointers

```
int* pi= (int*) malloc (sizeof(int));
free(pi);
*pi=52; //access unallocated memory
```

Symbolically Executing Allocation in Java

Replace $\langle$`a = new T();`$\rangle$ **by**

$$\{a := \mathrm{obj}_T(next_T) \| \mathrm{next}_T := \mathrm{next}_T + 1\}$$

Explicit object deallocation (ANSI C only)

Example Dangling Pointers

```
int* pi= (int*) malloc (sizeof(int));
free(pi);
*pi=52; //access unallocated memory
```

Symbolically Executing Allocation in Java

Replace ⟨`a = new T();`⟩ **by**

$$\{a := \mathrm{obj}_T(next_T) \| \mathrm{next}_T := \mathrm{next}_T + 1\}$$

Explicit object deallocation (ANSI C only)

Example Dangling Pointers

```
int* pi= (int*) malloc (sizeof(int));
free(pi);
*pi=52; //access unallocated memory
```

Symbolically Executing Allocation in Java

Replace $\langle$`a = new T();`$\rangle$ **by**

$\{a := \mathrm{obj}_T(next_T) \| \mathrm{next}_T := \mathrm{next}_T + 1\}$

Explicit object deallocation (ANSI C only)

Example Dangling Pointers

```
int* pi= (int*) malloc (sizeof(int));
free(pi);
*pi=52; //access unallocated memory
```

Symbolically Executing Allocation in Java

Replace ⟨`a = new T();`⟩ **by**

$\{a := \mathrm{obj}_T(next_T) \| \mathrm{next}_T := \mathrm{next}_T + 1\}$

Explicit object deallocation (ANSI C only)

Example Dangling Pointers

```
int* pi= (int*) malloc (sizeof(int));
free(pi);
*pi=52; //access unallocated memory
```

Symbolically Executing Allocation in Java

Replace $\langle$`a = new T();`$\rangle$ **by**

$$\{a := \mathrm{obj}_T(next_T) \| \mathrm{next}_T := \mathrm{next}_T + 1\}$$

Explicit object deallocation (ANSI C only)

Example Dangling Pointers

```
int* pi= (int*) malloc (sizeof(int));
free(pi);
*pi=52; //access unallocated memory
```

Symbolically Executing Allocation in Java

Replace $\langle$`a = new T();`$\rangle$ **by**

$$\{a := \mathrm{obj}_T(next_T) \| \mathrm{next}_T := \mathrm{next}_T + 1\}$$

Explicit object deallocation (ANSI C only)

Example Dangling Pointers

```
int* pi= (int*) malloc (sizeof(int));
free(pi);
*pi=52; //access unallocated memory
```

Symbolically Executing Allocation in C

Replace ⟨`a = (*T)malloc(sizeof(T));`⟩ **by**

$$\{^*(a) := obj_T(next_T) \|^{cr} (^*(a)) := \mathsf{true} \| next_T := next_T + 1\}$$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace ⟨`free(a);`⟩ **by** $\{^{cr}(^*(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$
\begin{array}{rcl}
obj_T(n) & : & \text{Root} \\
{}^{.m}(x),\ {}^{[]}(x,i) & : & \text{NonRoot } \textit{where } x \in \text{Root} \cup \text{NonRoot} \\
a,\ {}^{.m}(a),\ {}^{[]}(a,i) & : & \text{Static}
\end{array}
$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.\left((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o)\right)$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{^{cr}(^*(a)) := \mathit{false}\}$

`int a; free(&a);`

`struct S* s; ... ; free(&(s->m))`

Extending the Type System

$$
\begin{array}{rcl}
obj_T(n) & : & \text{Root} \\
{}^{.m}(x),\ {}^{[]}(x, i) & : & \text{NonRoot } \mathit{where}\ x \in \text{Root} \cup \text{NonRoot} \\
a,\ {}^{.m}(a),\ {}^{[]}(a, i) & : & \text{Static}
\end{array}
$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \land o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{^{cr}(^{*}(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$\begin{array}{rcl} obj_T(n) & : & \text{Root} \\ {}^{.m}(x),\ {}^{[]}(x,i) & : & \text{NonRoot } \textit{where } x \in \text{Root} \cup \text{NonRoot} \\ a,\ {}^{.m}(a),\ {}^{[]}(a,i) & : & \text{Static} \end{array}$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{{}^{cr}({}^{*}(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$\begin{aligned} obj_T(n) &: \text{Root} \\ {}^{.m}(x),\ {}^{[]}(x,i) &: \text{NonRoot}\ \mathit{where}\ x \in \text{Root} \cup \text{NonRoot} \\ a,\ {}^{.m}(a),\ {}^{[]}(a,i) &: \text{Static} \end{aligned}$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{{}^{cr}({}^{*}(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$\begin{aligned} obj_T(n) &: \text{Root} \\ {}^{.m}(x),\ {}^{[]}(x,i) &: \text{NonRoot } \textit{where } x \in \text{Root} \cup \text{NonRoot} \\ a,\ {}^{.m}(a),\ {}^{[]}(a,i) &: \text{Static} \end{aligned}$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{^{cr}(^{*}(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$
\begin{array}{rcl}
obj_T(n) & : & \text{Root} \\
{}^{.m}(x),\ {}^{[]}(x,i) & : & \text{NonRoot}\ \textit{where}\ x \in \text{Root} \cup \text{NonRoot} \\
a,\ {}^{.m}(a),\ {}^{[]}(a,i) & : & \text{Static}
\end{array}
$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{{}^{cr}({}^{*}(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$\begin{array}{rcl} obj_T(n) & : & \text{Root} \\ {}^{.m}(x),\ {}^{[]}(x,i) & : & \text{NonRoot } \mathit{where}\ x \in \text{Root} \cup \text{NonRoot} \\ a,\ {}^{.m}(a),\ {}^{[]}(a,i) & : & \text{Static} \end{array}$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{^{cr}(^{*}(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$\begin{array}{rcl} obj_T(n) & : & \text{Root} \\ ^{.m}(x),\ ^{[]}(x,i) & : & \text{NonRoot } \mathit{where}\ x \in \text{Root} \cup \text{NonRoot} \\ a,\ ^{.m}(a),\ ^{[]}(a,i) & : & \text{Static} \end{array}$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{^{cr}(^{*}(a)) := \mathit{false}\}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$\begin{array}{rcl} obj_T(n) & : & \text{Root} \\ {}^{.m}(x),\ {}^{[]}(x,i) & : & \text{NonRoot}\ \mathit{where}\ x \in \text{Root} \cup \text{NonRoot} \\ a,\ {}^{.m}(a),\ {}^{[]}(a,i) & : & \text{Static} \end{array}$$

Quantification

$\forall o : T_1.\Phi(o)$

$\forall i.\forall o : T_1.((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

Explicit object deallocation (ANSI C only)

Symbolically Executing Deallocation

Replace $\langle$`free(a);`$\rangle$ **by** $\{ {}^{cr}({}^{*}(a)) := \mathit{false} \}$

```
int a; free(&a);
struct S* s; ... ; free(&(s->m))
```

Extending the Type System

$$\begin{array}{rcl} obj_T(n) & : & \text{Root} \\ {}^{.m}(x),\ {}^{[]}(x,i) & : & \text{NonRoot } \mathit{where}\ x \in \text{Root} \cup \text{NonRoot} \\ a,\ {}^{.m}(a),\ {}^{[]}(a,i) & : & \text{Static} \end{array}$$

Quantification

$\forall o : T_1 . \Phi(o)$

$\forall i . \forall o : T_1 . ((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$

goto-Statement (ANSI C only)

Example

```
int i;
for(i=0;i<2;i++)
{
   label1:
   printf("(i=%i) ",i);
}
if(i<4)goto label1;
```

Border Issues

Volatile program variables

- Prevents certain optimizations by an optimizing compiler
- Can be modified by external concurrently running programs

Function pointers

- An obtained function pointer must be “specified”
- Otherwise, no harder than polymorphism with unspecified classes

Border Issues

Volatile program variables

- Prevents certain optimizations by an optimizing compiler
- Can be modified by external concurrently running programs

Function pointers

- An obtained function pointer must be "specified"
- Otherwise, no harder than polymorphism with unspecified classes

Conclusion

- Most features of C are similar to those in Java
- Most additional feature can be handled with existing concepts
- Handling of type casts, unions, pointer arithmetics is infeasible
- Should we concentrate on instances of ANSI C, or ...
- ... should we concentrate on subsets of ANSI C?

Conclusion

- Most features of C are similar to those in Java
- Most additional feature can be handled with existing concepts
- Handling of type casts, unions, pointer arithmetics is infeasible
- Should we concentrate on instances of ANSI C, or ...
- ... should we concentrate on subsets of ANSI C?

Conclusion

- Most features of C are similar to those in Java
- Most additional feature can be handled with existing concepts
- Handling of type casts, unions, pointer arithmetics is infeasible
- Should we concentrate on instances of ANSI C, or ...
- ... should we concentrate on subsets of ANSI C?

Conclusion

- Most features of C are similar to those in Java
- Most additional feature can be handled with existing concepts
- Handling of type casts, unions, pointer arithmetics is infeasible
- Should we concentrate on instances of ANSI C, or ...
- ... should we concentrate on subsets of ANSI C?

Conclusion

- Most features of C are similar to those in Java
- Most additional feature can be handled with existing concepts
- Handling of type casts, unions, pointer arithmetics is infeasible
- Should we concentrate on instances of ANSI C, or ...
- ... should we concentrate on subsets of ANSI C?

Conclusion

- Most features of C are similar to those in Java
- Most additional feature can be handled with existing concepts
- Handling of type casts, unions, pointer arithmetics is infeasible
- Should we concentrate on instances of ANSI C, or ...
- ... should we concentrate on subsets of ANSI C?