Program Transformations as Abstract Interpretation

MPRI — Cours 2.6 "Interprétation abstraite :
application à la vérification et à l'analyse statique"

Xavier Rival

INRIA, ENS, CNRS

Jan, 27th. 2016

Program transformations and static analysis

Previous lectures: **focus on static analysis techniques**, *i.e.*

1. take **one program as argument**
2. compute some **semantic properties** of the program
   e.g., compute an over-approximation of the reachable states
   e.g., verify the absence of runtime errors

Today: we consider **program transformations**

- functions that **compute a program from another program**
- thus, we will consider not a single program but **two**
- different set of **issues**
  - abstract interpretation to reason about and **verify the transformation**
  - static analysis to **enable the transformation**

Compilation

- **Transforms programs in high level languages** (OCaml, C, Java) **into assembly**
- **Verifies** (*e.g.*, types) and **Optimizes**

Source code:

```
int f( int a, int b ){
  int x0 = a - b;
  if( x0 > 0 )
    return x0 * (a + b);
  else return 0;
}
```

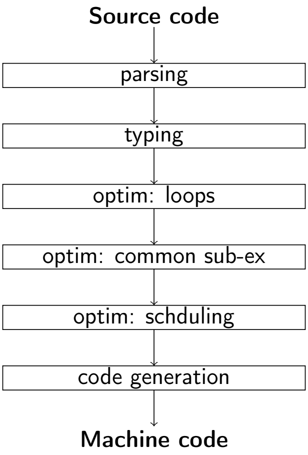
Compiled code:

```
.file "foo.c"
.text
.globl f
.type f, @function
f:
.LFB0:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
subl $16, %esp
```

```
movl 12(%ebp), %eax
movl 8(%ebp), %edx
movl %edx, %ecx
subl %eax, %ecx
movl %ecx, %eax
movl %eax, -4(%ebp)
cmpl $0, -4(%ebp)
jle .L2
movl 12(%ebp), %eax
movl 8(%ebp), %edx
addl %edx, %eax
imull -4(%ebp), %eax
jmp .L3
```

```
.L2:
movl $0, %eax
.L3:
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size f, .-f
.ident
"GCC: (Gentoo 4.7.3-r1 p1.4, pie-0.5
.section
.note.GNU-stack,"",@progbits
```

Compilation phases



- **Parsing:** can be considered a static analysis
- **Typing:** static analysis
- **Optimizations:** enabled by static analysis
  e.g., code removed if proved dead
  e.g., expressions shared if common
- **Code generation:**
  by induction on syntax...

Slicing

Slice extraction

- a slice $\mathcal{S}$ is a **syntactic subset of a program** $\mathcal{P}$
- it is usually extracted following a **criterion**
  that describes **an observation of the program that is under study**
- there are **many** definitions of slicing criteria: a specific statement, a specific variable, the conjunction of both...

Applications:

- **program understanding:**
  you are given a program, and need to understand how it works...
- **program debugging:**
  a bug was identified, where `x` stores an unexpected value at line N...
- **program maintenance:**
  a legacy code needs to be extended; what will intended changes do ?

Slicing

Example: slice to **understand the value of a at line 5**

1 : **input**(x);	1 : **input**(x);
2 : **input**(y);	2 : **input**(y);
3 : $a = 4 * x + 8$;	3 : $a = 4 * x + 8$;
4 : $b = 3 - 2 * y + a$;	4 : $b = 3 - 2 * y + a$;
5 : $c = a + b$;	5 : $c = a + b$;

$\rightarrow$

Algorithm:

1. **compute dependences:** usually, a dependence graph describes what x *immediately* depends on, at line *N*
2. **extract a set of slice dependences** from the slicing criterion
3. **collect the corresponding statements** and produce the slice

Effectively, 1 and 2 **are a static analysis**

Partial evaluation

Specialization and optimization of programs

- start from a **very general program**
- + possibly some **assumptions on the input values**
- compute a program that **behaves similarly on those programs that satisfy the inputs**
- **partial evaluation** of all program statements that can be, but may also involve unrolling of loop, duplication of functions...

Applications:

- practical:
  design a software for several products,
  and specialize it for each product
- theoretical: Futamura's projections
  compilation = specialization of an interpreter to a program

Partial evaluation

```
while(c){                                    if(c){
  if(b){                                       x = 1;
    x = 1;                                     while(c){
  }else{           hyp: b = true                 x = f(x);
    x = f(x);           ⟶                      }
  }                                          }
  b = false;
}
```

1. unfolding of the loop for a number of iterations
2. propagation of the value of b through the loop
3. simplification of conditions and removal of b

Questions about program transformations

Soundness:

- in **what sense** can we say a transformation is sound ?
- what properties should it preserves ?
  what properties should it modify ?
- **how to semantically specify a transformation** ?

Use of semantic information:

- transformations often need **semantic properties of programs**, to decide what code to generate...
  e.g., for compiler optimizations, dependence information...
- in some cases the transformation itself may be potentially non terminating, and **require a widening** for convergence
  e.g., partial evaluation

Example: semantics of C volatile variables

From the ANSI C'99 / C'11 standards

For every read from or write to a volatile variable
that would be performed by a straightforward interpreter for C,
exactly one load or store from/to the memory location allocated to the variable should be performed.

In other words:

- volatile variables should be assumed to be **modifiable** by the external world **at any time** (this is a worst case assumption)
- multiple accesses to a single volatile variable **should never be optimized into a single read**
  (this is a very strong constraint on the optimizers)

Do compilers follow this semantics ? NO...

Example: C compiler and volatile variables

Study by E. Eide and J. Regher, "Volatiles are Mis-compiled, and What to Do about it" (EMSOFT'2008)

- **13 compilers tested**
- **none** of them is **exempt of volatile bugs**
- possible **consequences:**
  - **incorrect computations**
  - more serious crashes, such as **system hangs**
- one example on the next slide, more in the paper...

Since then, the **CompCert compiler** was tested free of volatile bugs using the same technique...

Example: C compiler and volatile variables

Compiler: LLVM GCC 2.2 (IA 32)

Buggy optimization:

```
volatile int a;
void foo(void){
    int i;
    for(i = 0; i < 3; i++){
        a += 7;
    }
}
```

```
foo:
  movl  a, %eax
  leal  7(%eax), %ecx
  movl  %ecx, a
  leal  14(%eax), %ecx
  movl  %ecx, a
  addl  $21, %eax
  movl  %eax, a
  ret
```

Only ONE load to a

- loop unrolled three times
- three stores (correct), but only one load (incorrect)

Main points of the lecture

Formalize soundness of program transformations:

- **compare the semantics of two programs**
- **select** the semantics to be compared by **abstraction**

Consider some verification techniques:

- **invariant verification** approach
- **local equivalence proof**...

These are **partly inspired from static analysis techniques**

Outline

Formalizing correctness: assumptions

Source language: C like imperative language

- very **simplified**: no procedure, library functions, etc

Assembly language: RISC style (similar to Power-PC)

- **registers:** differentiated dep. on types (floating-point, integers)
- **memory access:** direct, indirect, stack-based
- **condition register:**
  Tests and branchings are **separate** operations
  Conditional branching: tests the value of the condition register

Compiler:

- the lecture is not about showing a compiler...
- we first assume no optimization and consider optimizations later

Transition systems

We assume a (source or compiled) program is a **transition system** $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_{\mathcal{I}})$:

- $\mathbb{S} = \mathbb{L} \times \mathbb{M}$ is the set of **states**, where $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$
- $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$ is the **transition relation**
- $\mathbb{S}_{\mathcal{I}} \subseteq \mathbb{S}$ is the set of **initial states**

We consider their **finite traces semantics**:

- $[\![\mathcal{S}]\!] = \{\langle s_0, \ldots, s_n \rangle \in \mathbb{S}^\star \mid s_0 \in \mathbb{S}_{\mathcal{I}} \wedge \forall i,\, s_i \rightarrow s_{i+1}\}$
- it can be defined as a **least fix-point**: $[\![\mathcal{S}]\!] = \mathbf{lfp}\, F$

$$
\begin{array}{rlcl}
F : & \mathcal{P}(\mathbb{S}^\star) & \longrightarrow & \mathcal{P}(\mathbb{S}^\star) \\
 & X & \longmapsto & \{\langle s_0 \rangle \mid s \in \mathbb{S}_{\mathcal{I}}\} \\
 & & & \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle \\
 & & & \quad \mid \langle s_0, \ldots, s_n \rangle \in X \wedge s_n \rightarrow s_{n+1}\}
\end{array}
$$

(exercise)

A very minimal imperative language

$\mathtt{l}$	$::=$	**l-valules**	
	$\mid$	$\mathtt{x}$	$(\mathtt{x} \in \mathbb{X})$
$\mathtt{e}$	$::=$	**expressions**	
	$\mid$	c	$(c \in \mathbb{V})$
	$\mid$	$\mathtt{l}$	(l-value)
	$\mid$	$\mathtt{e} \oplus \mathtt{e}$	(arith operation, comparison)
$\mathtt{s}$	$::=$	**statements**	
	$\mid$	$\mathtt{l} = \mathtt{e}$	(assignment)
	$\mid$	$\mathtt{s}; \ldots \mathtt{s};$	(sequence)
	$\mid$	**if**$(\mathtt{e})\{\mathtt{s}\}$	(condition)
	$\mid$	**while**$(\mathtt{e})\{\mathtt{s}\}$	(loop)

Other extensions, not considered at this stage:

- functions
- collection of arithmetic data types, structures, unions, pointers
- compilation units...

A basic, PPC-like assembly language: principles

We now consider a (very simplified) **assembly language**

- machine integers: sequences of 32-bits (set: $\mathbb{B}^{32}$)
- instructions are encoded over 32-bits (set: $\mathbb{I}_{\mathrm{MIPS}}$) and stored into the same space as data (*i.e.*, $\mathbb{I}_{\mathrm{MIPS}} \subseteq \mathbb{B}^{32}$)
- loads and store instructions, with relative addressing instructions
- conditional branching is indirect:
  comparison instruction sets condition register **cr** (comparison flag)
  conditional branching instruction reads **cr** and branches accordingly

Memory locations

- **program counter** **pc** (current instruction address)
- **general purpose registers** $\mathbf{r}_0, \ldots, \mathbf{r}_{31}$
- **main memory** (RAM) **Addrs** $\rightarrow \mathbb{B}^{32}$ where **Addrs** $\subseteq \mathbb{B}^{32}$
- **condition register** **cr**

Then: $\mathbb{X}^c = \{\mathbf{pc}, \mathbf{cr}, \mathbf{r}_0, \ldots, \mathbf{r}_{31}\} \uplus \mathbf{Addrs}$

A basic, PPC-like assembly language: instruction set

Instruction encoded into 32-bits words:

Instruction set

$$v, dst, o \in \mathbb{B}^{32}, \quad \mathbf{cr} \in \{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\}$$

$$
\begin{array}{rcll}
i & ::= & (\in \mathbb{I}_{\mathrm{MIPS}}) & \\
& | & \mathbf{li}\ \mathbf{r}_d, v & \text{load } v \in \mathbb{B}^{32} \\
& | & \mathbf{add}\ \mathbf{r}_d, \mathbf{r}_{s0}, \mathbf{r}_{s1} & \text{addition} \\
& | & \mathbf{addi}\ \mathbf{r}_d, \mathbf{r}_{s0}, v & \text{add. } v \in \mathbb{V}' \subset \mathbb{B}^{32} \\
& | & \mathbf{sub}\ \mathbf{r}_d, \mathbf{r}_{s0}, \mathbf{r}_{s1} & \text{subtraction} \\
& | & \mathbf{cmp}\ \mathbf{r}_{s0}, \mathbf{r}_{s1} & \text{comparison} \\
& | & \mathbf{b}\ dst & \text{branch} \\
& | & \mathbf{blt}\langle cr \rangle\ dst & \text{cond. branch} \\
& | & \mathbf{ld}\ \mathbf{r}_d, o & \text{absolute load} \\
& | & \mathbf{st}\ \mathbf{r}_d, o & \text{absolute store} \\
& | & \mathbf{ldx}\ \mathbf{r}_d, o, \mathbf{r}_x & \text{relative load (aka indeXed load)} \\
& | & \mathbf{stx}\ \mathbf{r}_d, o, \mathbf{r}_x & \text{relative store (aka indeXed store)}
\end{array}
$$

A basic, PPC-like assembly language: states

Definition: state

A state is a tuple $s = (pc, \rho, cr, \mu)$ which comprises:

- a **program counter** value $pc \in \mathbb{B}^{32}$
- a function mapping each **general purpose register** to its value $\rho : \{0, \ldots, 31\} \rightarrow \mathbb{B}^{32}$
- a **condition register** value $cr \in \{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\}$
- a function mapping each **memory cell** to its value μ : **Addrs** $\rightarrow \mathbb{B}^{32}$

Equivalently, we can also write $s = (\ell, m)$, where

- the **control state** ℓ is the **current** pc **value**
- the **memory state** m is the triple (ρ, cr, μ)

(we use both notations in the following)

A basic, PPC-like assembly language: instruction set

We assume a state $s = (pc, \rho, cr, \mu)$ and that $\mu(pc) = i$.

Then:

- if $i = \mathbf{li}\ \mathbf{r}_d, v$, then:

$$s \rightarrow (pc + 4, \rho[d \mapsto v], cr, \mu)$$

- if $i = \mathbf{add}\ \mathbf{r}_d, \mathbf{r}_{s0}, \mathbf{r}_{s1}$, then:

$$s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) + \rho(s1))], cr, \mu)$$

- if $i = \mathbf{addi}\ \mathbf{r}_d, \mathbf{r}_{s0}, v$, then:

$$s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) + v)], cr, \mu)$$

- if $i = \mathbf{sub}\ \mathbf{r}_d, \mathbf{r}_{s0}, \mathbf{r}_{s1}$, then:

$$s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) - \rho(s1))], cr, \mu)$$

A basic, PPC-like assembly language: instruction set

We assume a state $s = (pc, \rho, cr, \mu)$ and that $\mu(pc) = i$.

Then:

- if $i = \mathbf{cmp}\ \mathbf{r}_{s0}, \mathbf{r}_{s1}$, then:

$$s \rightarrow \begin{cases} (pc+4, \rho, \mathrm{LT}, \mu) & \text{if } \rho(s0) < \rho(s1) \\ (pc+4, \rho, \mathrm{EQ}, \mu) & \text{if } \rho(s0) = \rho(s1) \\ (pc+4, \rho, \mathrm{GT}, \mu) & \text{if } \rho(s0) > \rho(s1) \end{cases}$$

- if $i = \mathbf{blt}\langle cond \rangle\ dst$, then:

$$s \rightarrow \begin{cases} (dst, \rho, \mathbf{cr}, \mu) & \text{if } cr = cond \\ (pc+4, \rho, \mathbf{cr}, \mu) & \text{otherwise} \end{cases}$$

- if $i = \mathbf{b}\ dst$, then:

$$s \rightarrow (dst, \rho, cr, \mu)$$

A basic, PPC-like assembly language: instruction set

We assume a state $s = (pc, \rho, cr, \mu)$ and that $\mu(pc) = i$.

Then:

- if $i = \textbf{ldx}\ \mathbf{r}_d, o, \mathbf{r}_x$, then:

$$s \rightarrow \begin{cases} (pc + 4, \rho[d \mapsto \mu(\rho(x) + o)], \mathbf{cr}, \mu) & \text{if } \mu(\rho(x) + o) \text{ is defined} \\ \Omega & \text{otherwise} \end{cases}$$

- if $i = \textbf{ld}\ \mathbf{r}_d, o$, then:

$$s \rightarrow \begin{cases} (pc + 4, \rho[d \mapsto \mu(o)], \mathbf{cr}, \mu) & \text{if } \mu(o) \text{ is defined} \\ \Omega & \text{otherwise} \end{cases}$$

- if $i = \textbf{stx}\ \mathbf{r}_d, o, \mathbf{r}_x$, then:

$$s \rightarrow \begin{cases} (pc + 4, \rho, \mathbf{cr}, \mu[\rho(x) + o) \mapsto \rho(d)]) & \text{if } \mu(\rho(x) + o) \text{ is defined} \\ \Omega & \text{otherwise} \end{cases}$$

- if $i = \textbf{ld}\ \mathbf{r}_d, o$, then effect can be deduced from the above cases

Output of a non optimizing compiler

Assumptions and conventions:

- t is an array of integers initialized to $t = \{0; 1; 4; -1\}$
- i, x are integer variables
- in the assembly, $\underline{x}$ denotes the address of x

source code		compiled code	
l_0^s	$i := i + 1;$	l_0^c	$\mathbf{ld}\ r_0, \underline{i}$
		l_1^c	$\mathbf{addi}\ r_0, r_0, 1$
		l_2^c	$\mathbf{st}\ r_0, \underline{i}$
l_1^s	$x := x + t[i];$	l_3^c	$\mathbf{ld}\ r_0, \underline{x}$
		l_4^c	$\mathbf{ld}\ r_1, \underline{i}$
		l_5^c	$\mathbf{ldx}\ r_2, \underline{t}, r_1$
		l_6^c	$\mathbf{add}\ r_0, r_0, r_2$
		l_7^c	$\mathbf{st}\ r_0, \underline{x}$
l_2^s	$\ldots$	l_8^c	$\ldots$

Is it sound ? What property does it preserve ?

A source level execution

$$\left\langle \left(\ell_0^s, \begin{array}{l} \mathtt{i} \mapsto 1; \\ \mathtt{x} \mapsto 1; \\ \mathtt{t}[0] \mapsto 0; \\ \mathtt{t}[1] \mapsto 1; \\ \mathtt{t}[2] \mapsto 4; \\ \mathtt{t}[3] \mapsto -1; \end{array} \right), \left(\ell_1^s, \begin{array}{l} \mathtt{i} \mapsto 2; \\ \mathtt{x} \mapsto 1; \\ \mathtt{t}[0] \mapsto 0; \\ \mathtt{t}[1] \mapsto 1; \\ \mathtt{t}[2] \mapsto 4; \\ \mathtt{t}[3] \mapsto -1; \end{array} \right), \left(\ell_2^s, \begin{array}{l} \mathtt{i} \mapsto 2; \\ \mathtt{x} \mapsto 5; \\ \mathtt{t}[0] \mapsto 0; \\ \mathtt{t}[1] \mapsto 1; \\ \mathtt{t}[2] \mapsto 4; \\ \mathtt{t}[3] \mapsto -1; \end{array} \right), \right\rangle$$

Correctness of compilation:

- we cannot find the **same** execution in the assembly: as memory locations are not the same at all
- thus, we expect a **"similar"** trace

Corresponding assembly level execution

l_0^c	**ld** $\mathbf{r}_0, \underline{\mathtt{i}}$	l_4^c	**ld** $\mathbf{r}_1, \underline{\mathtt{i}}$
l_1^c	**addi** $\mathbf{r}_0, \mathbf{r}_0, 1$	l_5^c	**ldx** $\mathbf{r}_2, \underline{\mathtt{t}}, \mathbf{r}_1$
l_2^c	**st** $\mathbf{r}_0, \underline{\mathtt{i}}$	l_6^c	**add** $\mathbf{r}_0, \mathbf{r}_0, \mathbf{r}_2$
l_3^c	**ld** $\mathbf{r}_0, \underline{\mathtt{x}}$	l_7^c	**st** $\mathbf{r}_0, \underline{\mathtt{x}}$

We consider an assembly level trace starting from a similar state:

state s_i^c	s_0^c	s_1^c	s_2^c	s_3^c	s_4^c	s_5^c	s_6^c	s_7^c	s_8^c
control state pc_i	l_0^c	l_1^c	l_2^c	l_3^c	l_4^c	l_5^c	l_6^c	l_7^c	l_8^c
register state $\rho_i(0)$	45	1	2	2	1	1	1	5	5
register state $\rho_i(1)$	-5	-5	-5	-5	-5	2	2	2	2
register state $\rho_i(2)$	89	89	89	89	89	89	4	4	4
memory state $\mu_i(\underline{\mathtt{i}})$	1	1	1	2	2	2	2	2	2
memory state $\mu_i(\underline{\mathtt{x}})$	1	1	1	1	1	1	1	1	5
memory state $\mu_i(\underline{\mathtt{t}} + 0)$	0	0	0	0	0	0	0	0	0
memory state $\mu_i(\underline{\mathtt{t}} + 1)$	1	1	1	1	1	1	1	1	1
memory state $\mu_i(\underline{\mathtt{t}} + 2)$	4	4	4	4	4	4	4	4	4
memory state $\mu_i(\underline{\mathtt{t}} + 3)$	-1	-1	-1	-1	-1	-1	-1	-1	-1

Source and assembly executions compared

state s_i^s	s_0^s	s_1^s	s_2^s
control state l_i^s	l_0^s	l_1^s	l_2^s
memory state $m_i^s(\mathtt{i})$	1	2	2
memory state $m_i^s(\mathtt{x})$	1	1	5
memory state $m_i^s(\mathtt{t}[0])$	0	0	0
memory state $m_i^s(\mathtt{t}[1])$	1	1	1
memory state $m_i^s(\mathtt{t}[2])$	4	4	4
memory state $m_i^s(\mathtt{t}[3])$	-1	-1	-1

Much more information in the assembly trace:

- registers values
- more control states

state s_i^c	s_0^c	s_1^c	s_2^c	s_3^c	s_4^c	s_5^c	s_6^c	s_7^c	s_8^c
control state pc_i	l_0^c	l_1^c	l_2^c	l_3^c	l_4^c	l_5^c	l_6^c	l_7^c	l_8^c
register state $\rho_i(0)$	45	1	2	2	1	1	1	5	5
register state $\rho_i(1)$	-5	-5	-5	-5	-5	2	2	2	2
register state $\rho_i(2)$	89	89	89	89	89	89	4	4	4
memory state $\mu_i(\underline{\mathtt{i}})$	1	1	1	2	2	2	2	2	2
memory state $\mu_i(\underline{\mathtt{x}})$	1	1	1	1	1	1	1	1	5
memory state $\mu_i(\underline{\mathtt{t}}+0)$	0	0	0	0	0	0	0	0	0
memory state $\mu_i(\underline{\mathtt{t}}+1)$	1	1	1	1	1	1	1	1	1
memory state $\mu_i(\underline{\mathtt{t}}+2)$	4	4	4	4	4	4	4	4	4
memory state $\mu_i(\underline{\mathtt{t}}+3)$	-1	-1	-1	-1	-1	-1	-1	-1	-1

An abstraction approach

state s_i^s	s_0^s			s_1^s					s_2^s
control state l_i^s	l_0^s			l_1^s					l_2^s
memory state $m_i^s(\mathtt{i})$	1			2					2
memory state $m_i^s(\mathtt{x})$	1			1					5
memory state $m_i^s(\mathtt{t}[0])$	0			0					0
memory state $m_i^s(\mathtt{t}[1])$	1			1					1
memory state $m_i^s(\mathtt{t}[2])$	4			4					4
memory state $m_i^s(\mathtt{t}[3])$	-1			-1					-1

state s_i^c	s_0^c	s_1^c	s_2^c	s_3^c	s_4^c	s_5^c	s_6^c	s_7^c	s_8^c
control state pc_i	l_0^c	l_1^c	l_2^c	l_3^c	l_4^c	l_5^c	l_6^c	l_7^c	l_8^c
register state $\rho_i(0)$	45	1	2	2	1	1	1	5	5
register state $\rho_i(1)$	-5	-5	-5	-5	-5	2	2	2	2
register state $\rho_i(2)$	89	89	89	89	89	89	4	4	4
memory state $\mu_i(\underline{\mathtt{i}})$	1	1	1	2	2	2	2	2	2
memory state $\mu_i(\underline{\mathtt{x}})$	1	1	1	1	1	1	1	1	5
memory state $\mu_i(\underline{\mathtt{t}}+0)$	0	0	0	0	0	0	0	0	0
memory state $\mu_i(\underline{\mathtt{t}}+1)$	1	1	1	1	1	1	1	1	1
memory state $\mu_i(\underline{\mathtt{t}}+2)$	4	4	4	4	4	4	4	4	4
memory state $\mu_i(\underline{\mathtt{t}}+3)$	-1	-1	-1	-1	-1	-1	-1	-1	-1

- We can **abstract away intermediate control states**

An abstraction approach

state s_i^s	s_0^s			s_1^s					s_2^s
control state l_i^s	l_0^s			l_1^s					l_2^s
memory state m_i^s(i)	1			2					2
memory state m_i^s(x)	1			1					5
memory state m_i^s(t[0])	0			0					0
memory state m_i^s(t[1])	1			1					1
memory state m_i^s(t[2])	4			4					4
memory state m_i^s(t[3])	-1			-1					-1
state s_i^c	s_0^c	s_1^c	s_2^c	s_3^c	s_4^c	s_5^c	s_6^c	s_7^c	s_8^c
control state pc_i	l_0^c	l_1^c	l_2^c	l_3^c	l_4^c	l_5^c	l_6^c	l_7^c	l_8^c
register state $\rho_i(0)$	45	1	2	2	1	1	1	5	5
register state $\rho_i(1)$	-5	-5	-5	-5	-5	2	2	2	2
register state $\rho_i(2)$	89	89	89	89	89	89	4	4	4
memory state $\mu_i(\underline{\text{i}})$	1	1	1	2	2	2	2	2	2
memory state $\mu_i(\underline{\text{x}})$	1	1	1	1	1	1	1	1	5
memory state $\mu_i(\underline{\text{t}} + 0)$	0	0	0	0	0	0	0	0	0
memory state $\mu_i(\underline{\text{t}} + 1)$	1	1	1	1	1	1	1	1	1
memory state $\mu_i(\underline{\text{t}} + 2)$	4	4	4	4	4	4	4	4	4
memory state $\mu_i(\underline{\text{t}} + 3)$	-1	-1	-1	-1	-1	-1	-1	-1	-1

- Intermediate control states abstracted; we can **forget registers**

An abstraction approach

state s_i^s	s_0^s			s_1^s					s_2^s
control state l_i^s	l_0^s			l_1^s					l_2^s
memory state $m_i^s(\mathtt{i})$	1			2					2
memory state $m_i^s(\mathtt{x})$	1			1					5
memory state $m_i^s(\mathtt{t}[0])$	0			0					0
memory state $m_i^s(\mathtt{t}[1])$	1			1					1
memory state $m_i^s(\mathtt{t}[2])$	4			4					4
memory state $m_i^s(\mathtt{t}[3])$	-1			-1					-1

state s_i^c	s_0^c	s_1^c	s_2^c	s_3^c	s_4^c	s_5^c	s_6^c	s_7^c	s_8^c
control state pc_i	l_0^c	l_1^c	l_2^c	l_3^c	l_4^c	l_5^c	l_6^c	l_7^c	l_8^c
register state $\rho_i(0)$	45	1	2	2	1	1	1	5	5
register state $\rho_i(1)$	-5	-5	-5	-5	-5	2	2	2	2
register state $\rho_i(2)$	89	89	89	89	89	89	4	4	4
memory state $\mu_i(\underline{\mathtt{i}})$	1	1	1	2	2	2	2	2	2
memory state $\mu_i(\underline{\mathtt{x}})$	1	1	1	1	1	1	1	1	5
memory state $\mu_i(\underline{\mathtt{t}}+0)$	0	0	0	0	0	0	0	0	0
memory state $\mu_i(\underline{\mathtt{t}}+1)$	1	1	1	1	1	1	1	1	1
memory state $\mu_i(\underline{\mathtt{t}}+2)$	4	4	4	4	4	4	4	4	4
memory state $\mu_i(\underline{\mathtt{t}}+3)$	-1	-1	-1	-1	-1	-1	-1	-1	-1

- Registers and intermediate control states removed
  We get very similar traces !

Syntactic relations

What we did remove:

- **intermediate control states**
- **memory locations** associated to **registers**

What we did preserve:

- **control states in correspondence:**

$$l_0^s \leftrightarrow l_0^c \qquad l_1^s \leftrightarrow l_3^c \qquad l_2^s \leftrightarrow l_8^c$$

- **memory location in correspondence:**

$$\begin{array}{ccc} \mathtt{i} \leftrightarrow \underline{\mathtt{i}} & \mathtt{x} \leftrightarrow \underline{\mathtt{x}} & \mathtt{i} \leftrightarrow \underline{\mathtt{i}} \\ \mathtt{t}[0] \leftrightarrow \underline{\mathtt{t}} + 0 & \mathtt{t}[1] \leftrightarrow \underline{\mathtt{t}} + 1 & \mathtt{t}[2] \leftrightarrow \underline{\mathtt{t}} + 2 \\ \mathtt{t}[3] \leftrightarrow \underline{\mathtt{t}} + 3 & & \end{array}$$

Intuitively, we did apply an abstraction (to a single trace)

Syntactic relations

Definition

We define two **syntactic mappings**:

- **Between control points:** $\pi_l : \mathbb{L}'_s \rightarrow \mathbb{L}'_c$ (where $\mathbb{L}'_i \subseteq \mathbb{L}_i$)
- **Between memory locations:** $\pi_x : \mathbb{X}'_s \rightarrow \mathbb{X}'_c$ (where $\mathbb{X}'_i \subseteq \mathbb{X}_i$)

We consider only **subsets** $\mathbb{X}', \ldots$ of $\mathbb{X}, \ldots$. For instance:

- Some variables in the source code may be removed
- Registers in P_c may not correspond to variables of P_s
- One statement in P_s corresponds to several instructions in P_c

In practice, π_l, π_x are **provided by the compiler**:

- **Linking information**
- **Line table**
- **Debugging information:** Stabs, COFF...

Syntactic relations

Definition

We define two **syntactic mappings**:

- **Between control points:** $\pi_l : \mathbb{L}'_s \rightarrow \mathbb{L}'_c$ (where $\mathbb{L}'_i \subseteq \mathbb{L}_i$)
- **Between memory locations:** $\pi_x : \mathbb{X}'_s \rightarrow \mathbb{X}'_c$ (where $\mathbb{X}'_i \subseteq \mathbb{X}_i$)

For our **example:**

- **Control points:**
  - $\mathbb{L}'_s = \{l^s_0, l^s_1, l^s_2\}$ and $\mathbb{L}'_c = \{l^c_0, l^c_3, l^c_7\}$
  - $\pi_l : l^s_0 \mapsto l^c_0;\ l^s_1 \mapsto l^c_3;\ l^s_2 \mapsto l^c_7$
- **Memory locations:**
  - $\mathbb{X}'_s = \{\mathtt{i}, \mathtt{x}, \mathtt{t}[0], \mathtt{t}[1], \mathtt{t}[2], \mathtt{t}[3]\}$ and $\mathbb{X}'_c = \{\underline{\mathtt{i}}, \underline{\mathtt{x}}, \underline{\mathtt{t}}, \underline{t} + 1, \underline{\mathtt{t}} + 2, \underline{t} + 3\}$
  - $\pi_x : \begin{cases} \mathtt{i} & \mapsto & \underline{i} \\ \mathtt{x} & \mapsto & \underline{x} \\ \mathtt{t}[n] & \mapsto & \underline{t} + n \end{cases}$

State observational abstraction

We now formalize the process to **project out irrelevant behaviors**:

- in **states**
- in **traces**
- in **the semantics**

We consider the assembly level first:

Definition: state abstraction

We let the **compiled code-level memory state abstraction** $\pi_c^{\mathbf{m}}$ be defined by:

$$\begin{array}{rccc} \pi_c^{\mathbf{m}}: & (\mathbb{X}_c \to \mathbb{V}) & \longrightarrow & (\mathbb{X}'_c \to \mathbb{V}) \\ & m & \longmapsto & \lambda(x \in \mathbb{X}'_c) \cdot m(x) \end{array}$$

Similar definition at the source level...
(though no variable needs to be abstracted at this point, we will make use of that possibility further in this course)

State observational abstraction: example

We recall that

$$\begin{array}{rcl} \mathbb{X}'_s & = & \{\mathtt{i}, \mathtt{x}, \mathtt{t}[0], \mathtt{t}[1], \mathtt{t}[2], \mathtt{t}[3]\} \\ \mathbb{X}'_c & = & \{\underline{\mathtt{i}}, \underline{\mathtt{x}}, \underline{\mathtt{t}}, \underline{t} + 1, \underline{\mathtt{t}} + 2, \underline{t} + 3 \end{array}$$

Then $\pi_c^{\mathbf{m}} : (pc, \rho, \mathbf{cr}, \mu) \longmapsto \mu$

So, in particular:

$$\pi_c^{\mathbf{m}} : \left(\begin{array}{lllll} pc & & \mapsto & l_0^c \\ \rho : & 0 & \mapsto & 45 \\ & 1 & \mapsto & -5 \\ & 2 & \mapsto & 4 \\ \mu : & \underline{\mathtt{i}} & \mapsto & 1 \\ & \underline{\mathtt{x}} & \mapsto & 1 \\ & \underline{\mathtt{t}} + 0 & \mapsto & 0 \\ & \underline{\mathtt{t}} + 1 & \mapsto & 1 \\ & \underline{\mathtt{t}} + 2 & \mapsto & 4 \\ & \underline{\mathtt{t}} + 3 & \mapsto & -1 \end{array} \right) \longmapsto \left(\begin{array}{llll} \mu : & \underline{\mathtt{i}} & \mapsto & 1 \\ & \underline{\mathtt{x}} & \mapsto & 1 \\ & \underline{\mathtt{t}} + 0 & \mapsto & 0 \\ & \underline{\mathtt{t}} + 1 & \mapsto & 1 \\ & \underline{\mathtt{t}} + 2 & \mapsto & 4 \\ & \underline{\mathtt{t}} + 3 & \mapsto & -1 \end{array} \right)$$

Trace observational abstraction

We can now lift the same abstraction principle to traces:

Definition: trace abstraction

We let the **compiled code-level trace abstraction** $\pi_c^{\mathbf{tr}}$ be defined by:

$$\begin{array}{rrcl} \pi_c^{\mathbf{tr}}: & (\mathbb{L}_c \times (\mathbb{X}_c \rightarrow \mathbb{V}))^\star & \longrightarrow & (\mathbb{L}'_c \times (\mathbb{X}'_c \rightarrow \mathbb{V}))^\star \\ & \langle (l_0, m_0), \ldots, (l_n, m_n) \rangle & \longmapsto & \langle (l_{k_0}, \pi_c^{\mathbf{m}}(m_{k_0})), \ldots, (l_{k_m}, \pi_c^{\mathbf{m}}(m_{k_m})) \rangle \end{array}$$

$$\text{where: } \left\{ \begin{array}{l} \{k_0, \ldots, k_m\} = \{k \mid 0 \leq k \leq n \wedge l_k \in \mathbb{L}'_c\} \\ k_0 < \ldots < k_m \end{array} \right.$$

Similar definition at the source level...
(though no control state / variable needs to be abstracted at this point, we will make use of that possibility further in this course)

Trace observational abstraction: example

$\pi^{\mathbf{tr}}$:

control state pc_i	l_0^c	l_1^c	l_2^c	l_3^c	l_4^c	l_5^c	l_6^c	l_7^c	l_8^c
register state $\rho_i(0)$	45	1	2	2	1	1	1	5	5
register state $\rho_i(1)$	-5	-5	-5	-5	-5	2	2	2	2
register state $\rho_i(2)$	89	89	89	89	89	89	4	4	4
memory state $\mu_i(\underline{\mathtt{i}})$	1	1	1	2	2	2	2	2	2
memory state $\mu_i(\underline{\mathtt{x}})$	1	1	1	1	1	1	1	1	5
memory state $\mu_i(\underline{\mathtt{t}}+0)$	0	0	0	0	0	0	0	0	0
memory state $\mu_i(\underline{\mathtt{t}}+1)$	1	1	1	1	1	1	1	1	1
memory state $\mu_i(\underline{\mathtt{t}}+2)$	4	4	4	4	4	4	4	4	4
memory state $\mu_i(\underline{\mathtt{t}}+3)$	-1	-1	-1	-1	-1	-1	-1	-1	-1

$\longmapsto$

control state pc_i	l_0^c	l_3^c	l_8^c
memory state $\mu_i(\underline{\mathtt{i}})$	1	2	2
memory state $\mu_i(\underline{\mathtt{x}})$	1	1	5
memory state $\mu_i(\underline{\mathtt{t}}+0)$	0	0	0
memory state $\mu_i(\underline{\mathtt{t}}+1)$	1	1	1
memory state $\mu_i(\underline{\mathtt{t}}+2)$	4	4	4
memory state $\mu_i(\underline{\mathtt{t}}+3)$	-1	-1	-1

Observable behaviors inclusions

Applying this systematically to all traces results into **an abstraction**:

Result: compiled code observational abstraction

We let α_c^r be the **compiled code observational abstraction**:

$$\begin{array}{rlcl} \alpha_c^r : & \mathcal{P}((\mathbb{L}_c \times (\mathbb{X}_c \rightarrow \mathbb{V}))^\star) & \longrightarrow & \mathcal{P}((\mathbb{L}_c' \times (\mathbb{X}_c' \rightarrow \mathbb{V}))^\star) \\ & \mathcal{E} & \longmapsto & \{\pi_c^{\mathbf{tr}}(\sigma) \mid \sigma \in \mathcal{E}\} \end{array}$$

It defines a **Galois connection** with an adjoint **concretization** γ_c^r:

$$(\mathcal{P}((\mathbb{L}_c \times (\mathbb{X}_c \rightarrow \mathbb{V}))^\star), \subseteq) \underset{\alpha_c^r}{\overset{\gamma_c^r}{\leftrightarrows}} (\mathcal{P}((\mathbb{L}_c' \times (\mathbb{X}_c' \rightarrow \mathbb{V}))^\star), \subseteq)$$

- α_c^r is monotone and the concrete domain is a complete lattice; the concretization function follows and is defined by $\gamma_c^r(\mathcal{E}') = \bigcup_{\mathcal{E}} \{\mathcal{E} \mid \alpha_c^r(\mathcal{E}) \subseteq \mathcal{E}'\} = \{\sigma \mid \pi^{\mathbf{tr}}(\sigma) \in \mathcal{E}'\}$
- The **observational semantics is defined by:** $[\![P_c]\!]_{\mathbf{obs}} = \alpha_c^r([\![P_c]\!])$

Correctness by semantic equivalence

- The same construction holds at the source level
- The resulting traces are **very similar**, up-to a **basic renaming**
- To define it, we assume the **syntactic mappings** π_l, π_x are **bijective**

Memory state renaming

We let the **memory state renaming function** be defined by:

$$\begin{array}{rccc} \pi_m : & (\mathbb{X}'_s \to \mathbb{V}) & \longrightarrow & (\mathbb{X}'_c \to \mathbb{V}) \\ & m & \longmapsto & m \circ \pi_x^{-1} \end{array}$$

Trace renaming

We let the **trace renaming function** be defined by:

$$\begin{array}{rccc} \pi_t : & \mathbb{L}'_s \times (\mathbb{X}'_s \to \mathbb{V}) & \longrightarrow & \mathbb{L}'_c \times (\mathbb{X}'_c \to \mathbb{V}) \\ & \langle (l_0, m_0), \ldots, (l_n, m_n) \rangle & \longmapsto & \langle (\pi_l(l_0), \pi_m(m_0)), \ldots, (\pi_l(l_n), \pi_m(m_n)) \rangle \end{array}$$

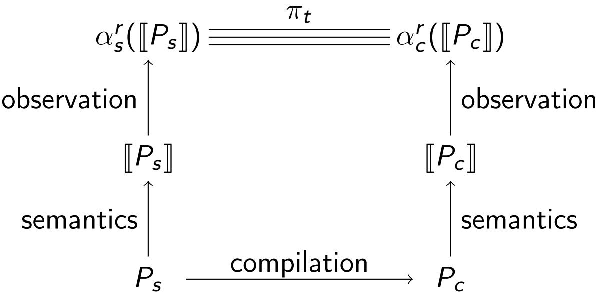
Correctness by semantic equivalence

We can now state the **compilation correctness definition**

Definition: compilation correctness

Compilation of P_s into P_c is correct *with respect to* π_l, π_x if and only if π_t establishes a bijection between $\alpha^r_s([\![P_s]\!])$ and $\alpha^r_c([\![P_c]\!])$.

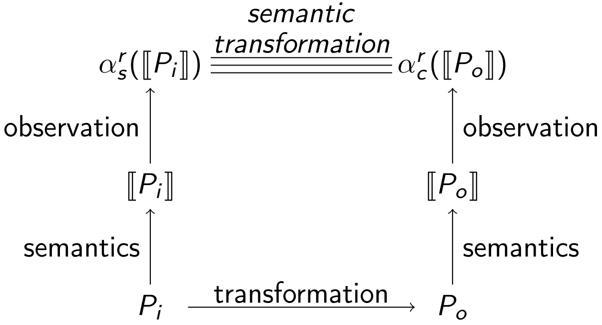
This definition can be illustrated by the diagram:



Correctness by semantic equivalence

This approach generalizes to other program transformations

This definition can be illustrated by the diagram:



Choice of another concrete semantics: consequences

New compilation correctness definition

$$\forall \rho \in \mathbb{M},\ [\![P_c]\!]_{\mathrm{rel}} \equiv [\![P_s]\!]_{\mathrm{rel}} \textbf{ modulo } \pi_l, \pi_x$$

This new definition is much weaker:

- **Correctness** assumes **no relation** about
  - intermediate control states
  - non terminating executions
- **More compilers** are considered correct
- **Weaker relation** between source and compiled programs
  This new definition really **misses something**, and impedes verification

Ways to **circumvent the limitation:**

1. Include **the whole trace into the final state!**
   Back to the previous definition, hard to formalize,
   says nothing about ∞...
2. Better way: **get it right first** and choose the right semantics!

Choice of another concrete semantics

We have built our definition of compilation correctness upon **operational (trace) semantics**.
What if we abstracted into **another observational semantics** ?

Alternate choice: let us consider a **more abstract semantics**

For instance, **relational semantics** (equivalent to denotational semantics)

- Notation for**initial** (resp. **final**) control states: $\ell_\vdash$ (resp. $\ell_\dashv$)
- Notation for **non-termination** written ∞;
- **Observational semantics:** relations between $\mathbb{M}$ and $\mathbb{M} \cup \{\infty\}$
- **Observational abstraction** defined by collecting for all traces:

$$\begin{array}{rcl} \langle(\ell_\vdash, \rho), \ldots, (\ell_\dashv, \rho')\rangle & \mapsto & (\rho, \rho') \\ \sigma = \langle(\ell_\vdash, \rho), \ldots\rangle & \mapsto & (\rho, \infty) \text{ if } \sigma \text{ infinite} \end{array}$$

- **Denotational semantics** defined by:

$$[\![P]\!]_{\mathrm{rel}} = \{(\rho, \rho') \mid \ldots\} \uplus \{(\rho, \infty) \mid \ldots\}$$

Outline

Optimizations

Until now we focused on **non-optimizing compilation**

In practice, compilers perform various **optimizations**

- **Elimination:** dead code, dead variables...
- **Instruction scheduling:** Instruction-Level-Parallelism...
- **Global transformations:** Propagation of common expressions...
- **Structural transformations:** Loop unrolling...

Consequences: π_l, π_x, $\mathbb{L}'_i$, $\mathbb{X}'_i$ **may not be defined**

Framework extension:

- **Redefine** the "**most precise observation preserved by compilation**"
- Would be **more difficult** with **bissimulations**
- **Next slides:** consider a few optimizations...

Dead-code elimination definition

Principle

Do not compile statements of the source program that provably never are executed

- This **saves space** as smaller executables get generated
- It also **improves runtime** as some tests may be removed (when they always produce the same result)

Example:

source code		compiled code	
l_0^s	$\mathrm{x} := 4;$	l_0^c	$\mathbf{li}\ \mathbf{r}_0, 4$
		l_1^c	$\mathbf{st}\ \mathbf{r}_0, \underline{\mathrm{x}}$
l_1^s	$\mathbf{if}(\mathrm{x} < 0)\{$		%% no code generated
l_2^s	$\quad \mathrm{x} = -\mathrm{x};$		%% no code generated
l_3^s	$\}$		%% no code generated
l_4^s	$\mathrm{x} = \mathrm{x} + 1$	l_2^c	$\mathbf{ld}\ \mathbf{r}_1, \underline{\mathrm{x}}$

Dead-code elimination correctness

How to set up a formal definition of compilation, that considers dead-code elimination correct ?

- we have to abstract away **all labels removed by the optimizations**
- this is **trivial**:
  we should simply **not include them in $\mathbb{L}'_s$**
- thus, our previous definition of compilation correctness **already accommodates dead-code elimination**

Compilation correctness in presence of dead-code elimination

Same definition as before

Dead-variable elimination definition

Principle

Discard entirely the variables that are never used anymore
(the compiler may reuse cells of dead local variables as well)

- This obviously both **saves space** and **improves runtime**
- **There is a caveat though: this may change the error semantics** indeed, expressions may be optimized away, so a program that normally fails (*e.g.*, on a division by zero) may not fail after optimization

```
...
x := y;
while(i < 10){
    x := x + 1;
    y := y - x - 1;
    i := i + 1;
}
use(x);
```

- x **read after the loop, but not** y
- thus, y **can be removed** with no observable change
- the purple statement disappears
- but y **does not disappear everywhere**

Dead-variable elimination correctness

How to set up a formal definition of compilation, that considers dead-variable elimination correct ?

- variables may need be removed **at certain program points**
- it is not possible to simply remove the dead variables from $\mathbb{X}_s$ altogether: in the example, this would not be correct, as y would be completely lost
- thus, π_x should be relational

Compilation correctness in presence of variable-code elimination

Similar definition as before, but with $\pi_x : \mathbb{L}_s \times \mathbb{X}_s \to \mathbb{X}_c$ instead.

Exercise: formalize the new definition, inspired from the previous one, and with $\pi_x : \mathbb{L}_s \times \mathbb{X}_s \to \mathbb{X}_c$ instead

Path modifying optimizations

Some optimization **deeply modify the control flow paths:**

- **loop unrolling**
- **loop exchange**
- **loop tiling**
- **loop interchange**
- flattening of **conditions**

Gains:

- more efficient code, due to **fewer conditions** (unrolling, tiling)
- enabling of **other optimizations**, *e.g.*, vectorization (tiling, interchange...)

In the next few slides, we consider the case of **loop unrolling**

Loop unrolling example

Assumption: a for loop run an even number of times
(loop unrolling may also apply to loops run a non statically known number of times, but it is more complex in that case)

source code

$$
\begin{array}{ll}
l_0^s & \mathtt{i} := 0; \\
l_1^s & \textbf{while}(\mathtt{i} < 1000) \\
l_2^s & \quad \mathtt{x} := \mathtt{x} * \mathtt{y}; \\
l_3^s & \quad \mathtt{y} := \mathtt{y} - 1; \\
l_4^s & \quad \mathtt{i} := \mathtt{i} + 1; \\
l_5^s & \}
\end{array}
$$

optimized code

$$
\begin{array}{ll}
l_0^o & \mathtt{i} := 0; \\
l_1^o & \textbf{while}(\mathtt{i} < 1000) \\
l_2^o & \quad \mathtt{x} := \mathtt{x} * \mathtt{y}; \\
l_3^o & \quad \mathtt{y} := \mathtt{y} - 1; \\
l_4^o & \quad \mathtt{x} := \mathtt{x} * \mathtt{y}; \\
l_5^o & \quad \mathtt{y} := \mathtt{y} - 1; \\
l_6^o & \quad \mathtt{i} := \mathtt{i} + 2; \\
l_7^o & \}
\end{array}
$$

Control state correspondence π_l is clearly broken:

$$
\pi_l : \left\{ \begin{array}{lcl} l_2^s & \leftrightarrow & l_2^o \\ l_2^s & \leftrightarrow & l_4^o \end{array} \right.
$$

Loop unrolling source and assembly traces

We consider executions in the source and the optimized code, and only display **control states at the assignment to** $\mathtt{x}$ and the **values of** $\mathtt{i}, \mathtt{y}$:

- At the **source code level:**

control state	l_2^s	l_2^s	l_2^s	l_2^s
value of $\mathtt{i}$	0	1	2	3
value of $\mathtt{y}$	1200	1199	1198	1197

- At the **compiled code level:**

control state	l_2^o	l_4^o	l_2^o	l_4^o
value of $\mathtt{i}$	0	0	2	2
value of $\mathtt{y}$	1200	1199	1198	1197

As expected:

- **the correlation** between the values of $\mathtt{i}$ and the other variables is **lost**
- the real correspondence is between **values of other variables** and **iterations even-ness**

Loop unrolling observational abstractions

How to set up a formal definition of compilation, that accepts loop unrolling as correct ?

- the loop counter variable `i` should be **excluded from** $\mathbb{X}_s, \mathbb{X}_o$
- each control state in the **source loop** should be **divided into a pair of labels**, that carry an **even-ness tab**:

$$\begin{array}{ccc} l_2^s & \mapsto & l_2^{s,e}, l_2^{s,o} \\ l_3^s & \mapsto & l_3^{s,e}, l_3^{s,o} \\ \ldots & \mapsto & \ldots \end{array}$$

- the trace abstraction function $\pi_s^{\mathbf{tr}}$ should map each loop body state into a state with a **consistent iteration even-ness**

This amounts to doing an even-ness based trace partitioning

Loop unrolling observational abstractions

We can consider the traces again:

source code	control state	l_2^s	l_2^s	l_2^s	l_2^s
	value of `i`	0	1	2	3
	value of `y`	1200	1199	1198	1197
source code, abstract	control state	$l_2^{s,e}$	$l_2^{s,o}$	$l_2^{s,e}$	$l_2^{s,o}$
	value of `i`	0	1	2	3
	value of `y`	1200	1199	1198	1197
optimized code	control state	l_2^o	l_4^o	l_2^o	l_4^o
	value of `i`	0	0	2	2
	value of `y`	1200	1199	1198	1197

We observe the **following control state correspondence:**

$$\pi_l : \begin{array}{ccc} l_2^{s,e} & \longmapsto & l_2^o \\ l_4^{s,o} & \longmapsto & l_4^o \end{array}$$

Loop unrolling correctness

Then, the definition follows a **very similar form** as before:

Compilation correctness in presence of loop unrolling

Similar definition as before, but with:

- **trace partitioning α_s^r abstraction**
- a mapping π_l that **preserves even-ness**

Instruction scheduling: instruction level parallelism

We now consider optimizations that modify the code **locally**, and take **instruction scheduling** as an example.

Instruction-level parallelism is a feature of modern processors:

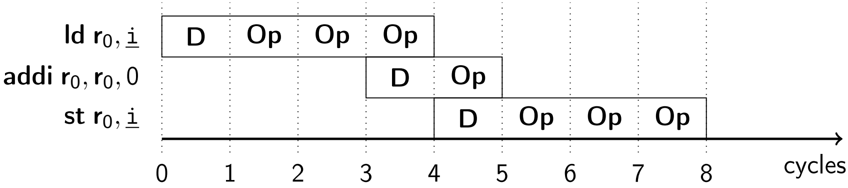
- **one** instruction = **one or several** cycles
  - memory typically slow: **load, store** take several cycles
    speed depends on the content of cache (hit/miss); can be 100 cycles!
  - arithmetic operations are usually faster
- **Pipeline:** run several instructions in parallel
- Some instructions **cannot be evaluated in parallel** due to dependences
- **Scheduling: re-ordering of instructions**
  so as to limit the number of *stall* cycles

Instruction level parallelism example

Assumptions:

- **arith. instructions**: 1 cycle instruction decoding, 1 cycle op.
- **load/store instructions**: 1 cycle instruction decoding, 3 cycle op.

We consider the code below:



Then, we observe **a two cycles stall after the load**

Consequence of this observation: instruction scheduling

More efficient code is generated if there are more instructions between load/store instruction and uses of the values loaded/stored

Instruction scheduling example

source code

l_0^s $\mathtt{i} := \mathtt{i} + 1;$

l_1^s $\mathtt{x} := \mathtt{x} + \mathtt{t}[\mathtt{i}];$

l_2^s $\ldots$

non optimized code

l_0^a ld $r_0, \underline{\mathtt{i}}$
l_1^a addi $r_0, r_0, 1$
l_2^a st $r_0, \underline{\mathtt{i}}$
l_3^a ld $r_1, \underline{\mathtt{x}}$
l_4^a ldx $r_2, \underline{\mathtt{t}}, r_0$
l_5^a add r_1, r_1, r_2
l_6^a st $r_1, \underline{\mathtt{x}}$
l_7^a $\ldots$

optimized code

l_0^o ld $r_0, \underline{\mathtt{i}}$
l_1^o ld $r_1, \underline{\mathtt{x}}$
l_2^o addi $r_0, r_0, 1$
l_3^o ldx $r_2, \underline{\mathtt{t}}, r_0$
l_4^o st $r_0, \underline{\mathtt{i}}$
l_5^o add r_1, r_1, r_2
l_6^o st $r_1, \underline{\mathtt{x}}$
l_7^o $\ldots$

Without optimization:
4 stall cycles, 14 cycles total

$l_0^s \leftrightarrow l_0^a$
$l_1^s \leftrightarrow l_3^a$
$l_2^s \leftrightarrow l_7^a$

Without optimization:
2 stall cycles, 12 cycles total

$l_0^s \leftrightarrow l_0^o$
$l_1^s \leftrightarrow$???
$l_2^s \leftrightarrow l_7^o$

Instruction scheduling observational abstractions

Issues to fix our definition:

- **Instructions execution order modified:**
  $l_1^a \rightarrow l_2^a$ and $l_2^a \rightarrow l_3^a$ are postponed
- **Mapping π_l is broken:**
  - The intermediate state l_1^s **has no clear counterpart in the assembly**
  - For `i`, it corresponds to l_5^o
  - For `x`, it corresponds to l_1^o
  - *In general:* this happens for all control points!
    (except for initial points, final points)

Thus, we need a **relational mapping** (π_l, π_x),
i.e., a single function taking care of both variables and control states:

Relational syntactic mapping

A **relational syntactic mapping** is defined by an injective function

$$\pi_{\mathbb{X}\times\mathbb{X}} : (\mathbb{L}'_s \times \mathbb{X}'_s) \longrightarrow (\mathbb{L}_c \times \mathbb{X}_c)$$

Instruction scheduling observational abstractions

Intuition

A source control state l^s corresponds to a **fictitious control state** where values of corresponding locations are gathered at different points in the execution of the optimized, compiled code

source code

l_0^s $\mathtt{i} := \mathtt{i} + 1;$

l_1^s $\mathtt{x} := \mathtt{x} + \mathtt{t}[\mathtt{i}];$

l_2^s $\ldots$

optimized code

l_0^o **ld** $\mathbf{r}_0, \underline{\mathtt{i}}$
l_1^o **ld** $\mathbf{r}_1, \underline{\mathtt{x}}$
l_2^o **addi** $\mathbf{r}_0, \mathbf{r}_0, 1$
l_3^o **ldx** $\mathbf{r}_2, \underline{\mathtt{t}}, \mathbf{r}_0$
l_4^o **st** $\mathbf{r}_0, \underline{\mathtt{i}}$
l_5^o **add** $\mathbf{r}_1, \mathbf{r}_1, \mathbf{r}_2$
l_6^o **st** $\mathbf{r}_1, \underline{\mathtt{x}}$
l_7^o $\ldots$

We then have:

$$\pi_{\mathbb{X}\times\mathbb{X}} : \begin{array}{rcl} (l_0^s, \mathtt{i}) & \mapsto & (l_0^o, \underline{\mathtt{i}}) \\ (l_0^s, \mathtt{x}) & \mapsto & (l_0^o, \underline{\mathtt{x}}) \\ (l_1^s, \mathtt{i}) & \mapsto & (l_5^o, \underline{\mathtt{i}}) \\ (l_1^s, \mathtt{x}) & \mapsto & (l_1^o, \underline{\mathtt{x}}) \\ (l_2^s, \mathtt{i}) & \mapsto & (l_7^o, \underline{\mathtt{i}}) \\ (l_2^s, \mathtt{x}) & \mapsto & (l_7^o, \underline{\mathtt{x}}) \end{array}$$

Instruction scheduling correctness

The source level observational abstraction is unchanged.

Optimized level observational abstraction

Optimized code observational abstraction α_s^r abstracts traces into **sequences of states observed at fictitious points**

We now obtain:

Compilation correctness in presence of instruction scheduling

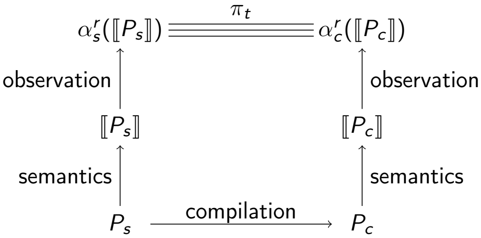
Similar definition as before, but with:

- **optimized code observational abstraction** α_s^r derived from $\pi_{\mathbb{X}\times\mathbb{X}}$
- **semantic mapping** π_t derived from $\pi_{\mathbb{X}\times\mathbb{X}}$

Compilation correctness

Definition: compilation correctness

Compilation of P_s into P_c is correct *with respect to* π_l, π_x (*resp.*, $\pi_{\mathbb{X}\times\mathbb{X}}$) if and only if π_t establishes a bijection between $\alpha_s^r(\llbracket P_s \rrbracket)$ and $\alpha_c^r(\llbracket P_c \rrbracket)$.



Main idea: optimizations handled as standard compilation, but with more complex mappings, and observational abstractions

On the formalization of program transformations

Methodology:

1. Set up the **standard semantics**
2. Define the **observation preserved by the transformation**
3. Derive the corresponding **abstractions**
4. Establish the correctness at the abstract level

Advantages of this approach:

- The framework **can be extended** (*e.g.*, with more complex abstractions)
- Abstract Interpretation theorems apply (*e.g.*, fix-point transfers)

Other extensions:

- **Define** the transformation **at the semantic level**
- **Derive** an implementation of the transformation, from the definition

Outline

Verifying compiled code

Kinds of properties:

- **safety** (no runtime errors, no overflows, no NaN...)
- **security** (no undesired information flow, in the sense of non-interference)

Two benefits:

- of course, verifying the generated code...
- but also, that **the compiler does not turn a correct (already verified) program into an incorrect assembly one**...

In the following, we consider **safety properties and invariants**

The invariant translation approach

Process

1. **Analyze the source** program P_s and compute an invariant $\mathcal{I}_s$
2. **Translate** $\mathcal{I}_s$ into assembly level candidate invariant $\mathcal{I}_t$
3. Perform an **assembly level check** of $\mathcal{I}_t$

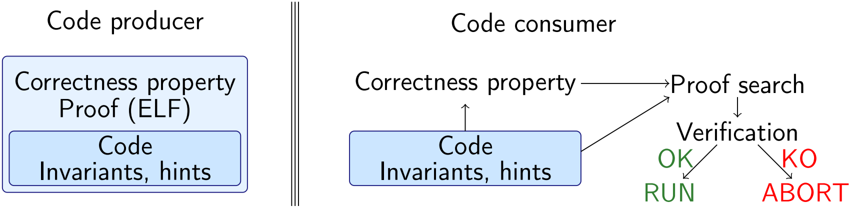
Motivation:

- inferring invariants is **hard** in general...
- and **even more so at the assembly level**
  due to an important loss of structure at compile time
  (data-structures flattened, control flow more complex, additional steps to perform an arithmetic assignment –with separate load and store– or a test –with separate test and branching instructions)

Example 1: Proof Carrying Codes (PCC)

Principle:

- **"Code producer":** provides code and **proof annotations** in binaries (*i.e.*, proof of correctness),
- **"Code consumer":** checks the safety of the code
  1. consistence of annotations: very quick proof search, from invariants
  2. annotations ⇒ the safety property we wish to enforce



Context: execution of **non-trusted** code downloaded in the Internet
e.g., it could contain a **security bug** (information leak, buffer overflow)

Example 2: TAL, compiled code certification by abstract interpretation

Typed and type safe assembly language:

- **Java bytecode:** interpreted (rather slow at runtime)
- **TALx86:** annotations for an assembly language closed to Intel **80x86**
- Removing types ⇒ executable code
- A specific compiler **translate** source level types

Advantages:

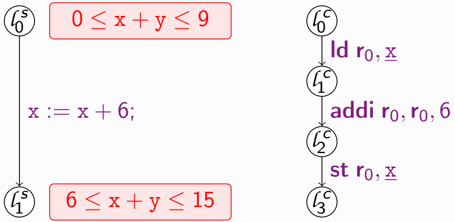
- Ensure the safety of **linkage** thanks to types
  Linkage of object files usually *not* sound
- Improve the reliability of **optimizations**
  Constraint: they should *preserve* types!
- Compilation of type-safe versions of C (CCured, CClone)

Certification of assembly code

Principle similar to PCC and TAL

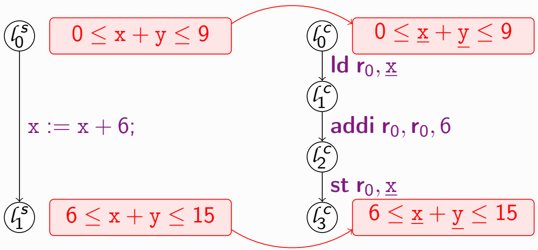
but computation of invariants by abstract interpretation

Assembly level verification of invariants



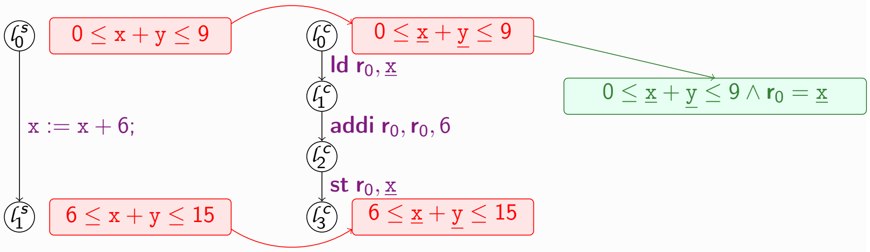
- Start with **invariants on the source code**

Assembly level verification of invariants



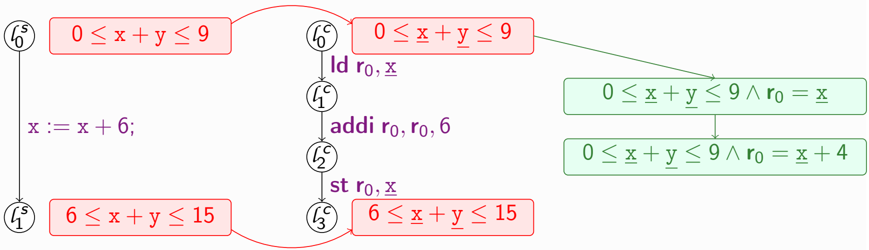
- **Translates those invariants**
  but not all control states are decorated

Assembly level verification of invariants



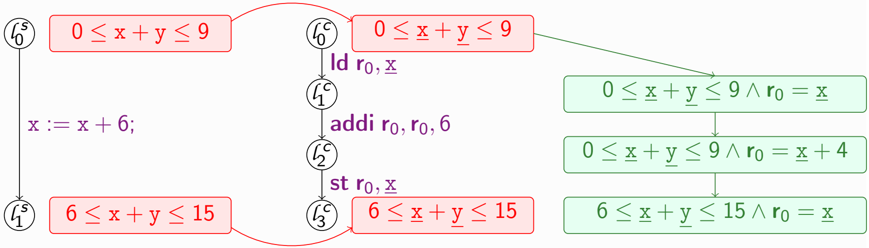
- **Propagates** the invariants and **computes refined local invariants**

Assembly level verification of invariants



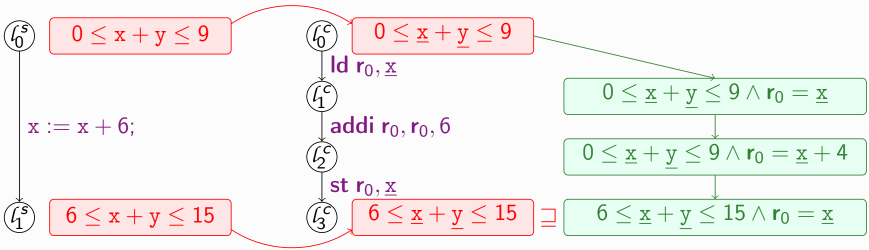
- **Propagates** the invariants and **computes refined local invariants**

Assembly level verification of invariants



- **Propagates** the invariants and **computes refined local invariants**

Assembly level verification of invariants



- **Checks invariance** at the end of the computation

Source static analysis: assumptions

- We assume an **abstraction of sets of stores** defined by **an abstraction function for sets of stores**

$$\alpha_{\mathrm{num}} : (\mathcal{P}(\mathbb{M}_s), \subseteq) \rightarrow (\mathbb{D}^\sharp_{\mathrm{num}}, \sqsubseteq)$$

- We derive an **abstraction for sets of executions**:

$$\begin{array}{rlcl} \alpha_{i,s} : & \mathcal{P}(\mathbb{S}_P{}^\star) & \longrightarrow & \mathbb{L}_s \rightarrow \mathbb{D}^\sharp_{\mathrm{num}} \\ & X & \longmapsto & (\ell \in \mathbb{L}_s) \mapsto \alpha_{\mathrm{num}}(\{m \mid \langle \ldots, (\ell, m), \ldots \rangle \in X\}) \end{array}$$

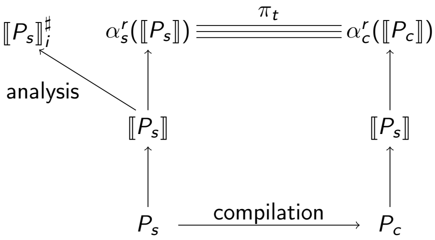
- We assume also a **source code static analysis**, that computes a sound over-approximation of the behaviors of the program:

$$\alpha_{i,s}(\llbracket P_s \rrbracket) \sqsubseteq \llbracket P_s \rrbracket^\sharp_i$$

Abstract invariant translation

Two abstractions have been defined:

- Abstraction for **static analysis** of P_s
- Abstraction for **defining compilation correctness**



Those abstractions are in general not comparable

Abstract invariant translation

We can derive **another abstraction**, more abstract than both α_s^r and $\alpha_{i,s}$:

- **theoretical result**: Galois-connections of a concrete domain form a **lattice**
- in practice, this common abstraction should **abstract away all the elements that are not in $\mathbb{L}_s', \mathbb{X}_s'$**:
  e.g., all dead variables, all unreachable control states...
  e.g., in case of loop unrolling, it should perform the same trace partitioning

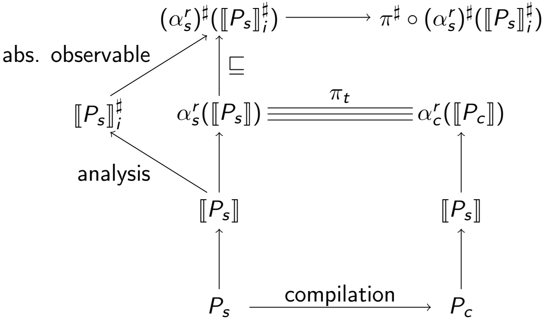
Moreover, π_l, π_x induce a **safe abstract invariant translation function** $\pi^\sharp : (\mathbb{L}_s' \to \mathbb{D}^\sharp_{\mathrm{num}}) \to (\mathbb{L}_c' \to \mathbb{D}^\sharp_{\mathrm{num}})$

- for each pair of control points in correspondence in π_l
- it maps numerical invariants among variables of P_s into numerical invariants among variables of P_c

Abstract invariant translation

Invariant translation process:

1. **Apply** $\pi^\sharp$ to an abstract invariant $[\![P_s]\!]_i^\sharp$ computed for P_s
2. Result: **a candidate invariant $\pi^\sharp([\![P_s]\!]_i^\sharp)$ for P_c**



Invariant translation: soundness

Soundness lemma

If:

- the **compilation $P_s \rightarrow P_c$ is sound** with respect to π_I, π_x;
- the **analysis of P_s computes a sound $[\![P_s]\!]_i^\sharp$** $\alpha_{i,s}([\![P_s]\!]) \sqsubseteq [\![P_s]\!]_i^\sharp$

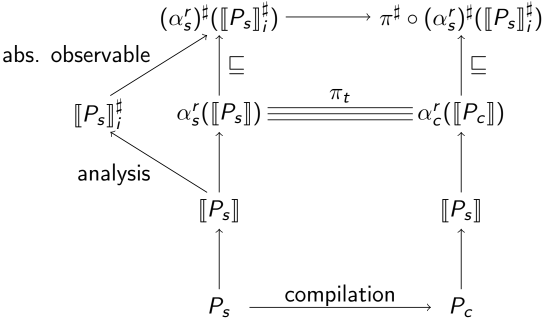
Then, $\pi^\sharp((\alpha_s^r)^\sharp([\![P_s]\!]_i^\sharp))$ **is a sound approximation of $[\![P_c]\!]$:**

$$\alpha_{i,r,c}([\![P_c]\!]) \sqsubseteq \pi^\sharp((\alpha_s^r)^\sharp([\![P_s]\!]_i^\sharp))$$

Consequence of the choice of another observational semantics for compilation correctness:
If $\alpha_s^r([\![P_s]\!])$, $\alpha_c^r([\![P_c]\!])$ are **weakened**, then the invariants that can be translated **are also weakened**

Invariant translation: soundness

Proof summarized:



Assumptions are very strong:

compilation, analysis, translation need to be correct

We need an independent verification of translated invariants

Independent verification of translated invariants

Principle of invariant checking: post-fixpoint checking

Theorem: invariant verification

Using a concretization function γ,

- **The *concrete* function F is monotone**,
- $F \circ \gamma \subseteq \gamma \circ F^\sharp$,
- $F^\sharp(x) \sqsubseteq x$,

Then, **lfp** $F \sqsubseteq \gamma(x)$

Proof left as exercise

- **Only the verifier needs to be sound** even if the assumptions of the translation soundness lemma are not met
  i.e., we can have an incorrect compiler, translate an incorrect invariant, and still obtain and check a correct translated invariant !
- In turn, **invariant checking is incomplete**

Independent verification of translated invariants

Principle of invariant checking: post-fixpoint checking

Theorem: invariant verification

Using a concretization function γ,

- **The *concrete* function F is monotone**,
- $F \circ \gamma \subseteq \gamma \circ F^{\sharp}$,
- $F^{\sharp}(x) \sqsubseteq x$,

Then, $\mathbf{lfp}\, F \sqsubseteq \gamma(x)$

Invariant checking refines abstract predicates:

this phase also produces more precise abstract properties about:

- **memory locations** in $\mathbb{X}_c \setminus \mathbb{X}'_c$
- **program points** in $\mathbb{L}_c \setminus \mathbb{L}'_c$

In practice, **every cycle of the compiled code control flow graph should contain an element of** $\mathbb{X}_s$

Invariant checking and difficulties

We consider the verification of invariants **around a condition test**

Assumptions:

- $\mathrm{x} \in [0, 12]$ at the entry point;
- we wish to **verify the assert in the compiled code**;
- we use a **non relational abstract domain: intervals**

Source code:

```
if(x ≤ 5){
    assert(x ≤ 5);
    ...
}else{
    ...
}
```

Compiled code:

```
0    ld r0, x
4    li r1, 5
8    cmp r0, r1
12   blt⟨GT⟩ ℓ     # (jump point)
16     ...# true branch contents
ℓ :  # false branch contents
```

Invariant checking and difficulties

0 : $\underline{\mathtt{x}} \in [0, 12]$

ld $\mathbf{r}_0, \underline{\mathtt{x}}$

4 :

li $\mathbf{r}_1, 5$

8 :

cmp $\mathbf{r}_0, \mathbf{r}_1$

12 :

blt$\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

16 :

Invariant checking and difficulties

$0:\quad \underline{\mathtt{x}} \in [0,12]$

$\quad\textbf{ld } \mathbf{r}_0, \underline{\mathtt{x}}$

$4:\quad \underline{\mathtt{x}} \in [0,12] \wedge \mathbf{r}_0 \in [0,12]$

$\quad\textbf{li } \mathbf{r}_1, 5$

$8:$

$\quad\textbf{cmp } \mathbf{r}_0, \mathbf{r}_1$

$12:$

$\quad\textbf{blt}\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

$16:$

Invariant checking and difficulties

$0:$ $\quad \underline{\mathtt{x}} \in [0, 12]$

$\quad\quad \mathbf{ld}\ \mathbf{r}_0, \underline{\mathtt{x}}$

$4:$ $\quad \underline{\mathtt{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12]$

$\quad\quad \mathbf{li}\ \mathbf{r}_1, 5$

$8:$ $\quad \underline{\mathtt{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5]$

$\quad\quad \mathbf{cmp}\ \mathbf{r}_0, \mathbf{r}_1$

$12:$

$\quad\quad \mathbf{blt}\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

$16:$

Invariant checking and difficulties

$0: \quad \underline{\mathrm{x}} \in [0, 12]$

$\textbf{ld } \mathbf{r}_0, \underline{\mathrm{x}}$

$4: \quad \underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12]$

$\textbf{li } \mathbf{r}_1, 5$

$8: \quad \underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5]$

$\textbf{cmp } \mathbf{r}_0, \mathbf{r}_1$

$12: \quad \underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5] \wedge \mathbf{cr} \in \{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\}$

$\textbf{blt}\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

$16:$

Invariant checking and difficulties

0 : $\underline{x} \in [0, 12]$

ld $\mathbf{r}_0, \underline{x}$

4 : $\underline{x} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12]$

li $\mathbf{r}_1, 5$

8 : $\underline{x} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5]$

cmp $\mathbf{r}_0, \mathbf{r}_1$

12 : $\underline{x} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5] \wedge \mathbf{cr} \in \{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\}$

blt$\langle \mathrm{GT} \rangle$ ℓ `# (jump point)`

16 : $\underline{x} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5] \wedge \mathbf{cr} \in \{\mathrm{LT}, \mathrm{EQ}\}$

Invariant checking and difficulties

0 : $\underline{\mathrm{x}} \in [0, 12]$

ld $\mathbf{r}_0, \underline{\mathrm{x}}$

4 : $\underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12]$

li $\mathbf{r}_1, 5$

8 : $\underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5]$

cmp $\mathbf{r}_0, \mathbf{r}_1$

12 : $\underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5] \wedge \mathbf{cr} \in \{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\}$

blt$\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

16 : $\underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5] \wedge \mathbf{cr} \in \{\mathrm{LT}, \mathrm{EQ}\}$

The condition at the branch point is not precise

The range of x was not refined by the test:

- the test and branching are **independent**
  relations between test results and values need be tracked
- the test is made on **a copy of** x
  equalities between copies need be tracked by the verifier

Refinement of the verifier

Relation between test and branching:

- each value in $\{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\}$ should be bound to the ranges of the other location
- this is obtained by a **value partitioning**, based on the value of **cr**:

$$\begin{array}{rlcl} \gamma: & (\{\mathrm{LT}, \mathrm{EQ}, \mathrm{GT}\} \to \mathbb{D}^{\sharp}_{\mathrm{num}}) & \longrightarrow & \mathcal{P}(\mathbb{M}) \\ & \phi^{\sharp} & \longmapsto & \{m \mid m \in \gamma_{\mathrm{num}} \circ \phi^{\sharp} \circ m(\mathbf{cr})\} \end{array}$$

Equalities between copies, *e.g.*, of $\underline{\mathrm{x}}$ and $\mathbf{r}_0$:

- an **equality abstraction** abstracts partitions of $\mathbb{X}_c$
- replacement of $\mathbb{D}^{\sharp}_{\mathrm{num}}$ with **a reduced product of $\mathbb{D}^{\sharp}_{\mathrm{num}}$ and an equality abstraction**

Invariant checking: fixed

0 : $\underline{\mathrm{x}} \in [0, 12]$

ld $\mathbf{r}_0, \underline{\mathrm{x}}$

4 :

li $\mathbf{r}_1, 5$

8 :

cmp $\mathbf{r}_0, \mathbf{r}_1$

12 :

blt$\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

16 :

In general, invariant checking is incomplete...
It may require some refinement in the verifier

Invariant checking: fixed

0 : $\underline{\mathrm{x}} \in [0, 12]$

$\mathbf{ld}\ \mathbf{r}_0, \underline{\mathrm{x}}$

4 : $\underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \underline{\mathrm{x}} = \mathbf{r}_0$

$\mathbf{li}\ \mathbf{r}_1, 5$

8 :

$\mathbf{cmp}\ \mathbf{r}_0, \mathbf{r}_1$

12 :

$\mathbf{blt}\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

16 :

In general, invariant checking is incomplete...
It may require some refinement in the verifier

Invariant checking: fixed

0 : $\underline{\mathrm{x}} \in [0, 12]$

ld $\mathbf{r}_0, \underline{\mathrm{x}}$

4 : $\underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \underline{\mathrm{x}} = \mathbf{r}_0$

li $\mathbf{r}_1, 5$

8 : $\underline{\mathrm{x}} \in [0, 12] \wedge \mathbf{r}_0 \in [0, 12] \wedge \mathbf{r}_1 \in [5, 5] \wedge \underline{\mathrm{x}} = \mathbf{r}_0$

cmp $\mathbf{r}_0, \mathbf{r}_1$

12 :

blt$\langle \mathrm{GT} \rangle$ ℓ `# (jump point)`

16 :

In general, invariant checking is incomplete...
It may require some refinement in the verifier

Invariant checking: fixed

0 : $\underline{x} \in [0, 12]$

ld $r_0, \underline{x}$

4 : $\underline{x} \in [0, 12] \wedge r_0 \in [0, 12] \wedge \underline{x} = r_0$

li $r_1, 5$

8 : $\underline{x} \in [0, 12] \wedge r_0 \in [0, 12] \wedge r_1 \in [5, 5] \wedge \underline{x} = r_0$

cmp r_0, r_1

12 : $\begin{cases} cr = \mathrm{LT} \implies \underline{x} \in [0, 4] \wedge r_0 \in [0, 4] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \\ cr = \mathrm{EQ} \implies \underline{x} \in [5, 5] \wedge r_0 \in [5, 5] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \\ cr = \mathrm{GT} \implies \underline{x} \in [6, 12] \wedge r_0 \in [6, 12] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \end{cases}$

blt$\langle \mathrm{GT} \rangle\ \ell$ `# (jump point)`

16 :

In general, invariant checking is incomplete...
It may require some refinement in the verifier

Invariant checking: fixed

0 : $\underline{x} \in [0, 12]$

ld $r_0, \underline{x}$

4 : $\underline{x} \in [0, 12] \wedge r_0 \in [0, 12] \wedge \underline{x} = r_0$

li $r_1, 5$

8 : $\underline{x} \in [0, 12] \wedge r_0 \in [0, 12] \wedge r_1 \in [5, 5] \wedge \underline{x} = r_0$

cmp r_0, r_1

$$12 : \left\{ \begin{array}{lcl} cr = \mathrm{LT} & \Longrightarrow & \underline{x} \in [0, 4] \wedge r_0 \in [0, 4] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \\ cr = \mathrm{EQ} & \Longrightarrow & \underline{x} \in [5, 5] \wedge r_0 \in [5, 5] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \\ cr = \mathrm{GT} & \Longrightarrow & \underline{x} \in [6, 12] \wedge r_0 \in [6, 12] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \end{array} \right.$$

blt$\langle \mathrm{GT} \rangle$ ℓ `# (jump point)`

$$16 : \left\{ \begin{array}{lcl} cr = \mathrm{LT} & \Longrightarrow & \underline{x} \in [0, 4] \wedge r_0 \in [0, 4] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \\ cr = \mathrm{EQ} & \Longrightarrow & \underline{x} \in [5, 5] \wedge r_0 \in [5, 5] \wedge \underline{x} = r_0 \wedge r_1 \in [5, 5] \\ cr = \mathrm{EQ} & \Longrightarrow & \bot \end{array} \right.$$

In general, invariant checking is incomplete...
It may require some refinement in the verifier

Outline

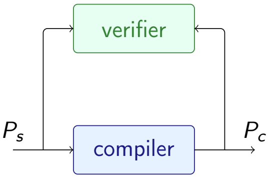
Verifying a compiler result

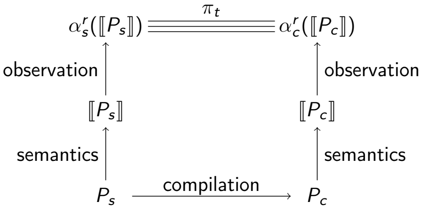
Principle: verify the semantic equivalence between source and compiled programs

Verification process: translation validation

1. **Establish mappings** π_l, π_x between source and compiled programs
2. **Prove (with a specialized prover) the semantic equivalence of each basic block**

Process:





A technique based on fixpoint transfer

Foundation: fixpoint transfer

Theorem

Let $F_s : \mathcal{P}(\mathbb{S}_s{}^\star) \to \mathcal{P}(\mathbb{S}_s{}^\star)$ and $F_c : \mathcal{P}(\mathbb{S}_c{}^\star) \to \mathcal{P}(\mathbb{S}_c{}^\star)$ and $\pi_t : \mathbb{S}_s{}^\star \to \mathbb{S}_c{}^\star$ (complete for join), such that:

- F_s, F_c **are monotone**
- $\pi_t(\emptyset) = \emptyset$ ($\emptyset$ least element);
- $\pi_t \circ F_s = F_c \circ \pi_t$

then both functions have a least fixpoint and:

$$\mathbf{lfp}\, F_c = \pi_t(\mathbf{lfp}\, F_s)$$

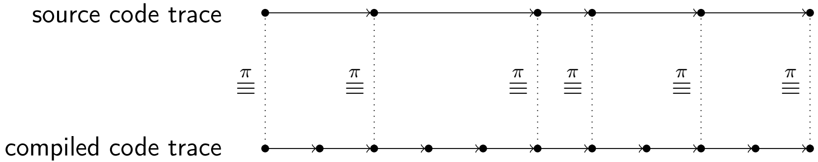
Proof: exercise

But the theorem does not apply directly:

source and compiled executions are not correlated step-by-step

A technique based on fixpoint transfer

Equivalence of source and assembly traces:



- **standard semantics** $[\![P_s]\!]$ and $[\![P_c]\!]$ are expressed as least fixpoints, but not directly correlated by π_x, π_l
- **observational semantics** $\alpha_s^r([\![P_s]\!])$ and $\alpha_c^r([\![P_c]\!])$ are directly correlated by not expressed as least fixpoint

We need fixpoint definitions for $\alpha_s^r([\![P_s]\!]), \alpha_c^r([\![P_c]\!])$ (*e.g.*, each basic block in the assembly code should be one computation step)

Symbolic transfer functions: definition

A language to describe the effect of a basic block

- basic blocks usually contain **series of assignment**:
  we **flatten sequences of assignments into parallel assignments**
- a basic block may branch to **several points** (often two)
- **no loop**: each cycle in the compiled code control flow graph is associated to at least one control state in the source

Symbolic transfer functions

Symbolic transfer functions are defined by the grammar:

$$\begin{array}{rcll} \delta(\in \mathbb{T}) & ::= & \square & \text{no transition (dead branch, error)} \\ & | & \lfloor \overrightarrow{x} \leftarrow \overrightarrow{e} \rfloor & \text{parallel assignment} \\ & | & \lfloor c \ ? \ \delta_0 \mid \delta_1 \rfloor & \text{conditional} \end{array}$$

Intuitively, a symbolic transfer function is a **store transformer**

Symbolic transfer functions: semantics

Semantic domain:

- $\bot$ corresponds to the absence of behavior (error, blocking)
- $[\![\delta]\!] \in \mathbb{M} \to \mathbb{M} \cup \{\bot\}$

Denotational Semantics:

- $[\![\square]\!](\rho) = \bot$
- $[\![\lfloor x \leftarrow e \rfloor]\!](\rho) = \rho[\forall i,\ [\![x_i]\!](\rho) \leftarrow [\![e_i]\!](\rho)]$
  if $\forall i,\ [\![x_i]\!](\rho) \neq \mathrm{error}$ and $\forall i,\ [\![e_i]\!](\rho) \neq \mathrm{error}$
  $[\![\lfloor x \leftarrow e \rfloor]\!](\rho) = \bot$ otherwise
- $$[\![\lfloor e\ ?\ \delta_0 \mid \delta_1 \rfloor]\!](\rho) = \begin{cases} [\![\delta_0]\!](\rho) & \text{if } [\![e]\!](\rho) = \textbf{true} \\ [\![\delta_1]\!](\rho) & \text{if } [\![e]\!](\rho) = \textbf{false} \\ \bot & \text{if } [\![e]\!](\rho) = \mathrm{error} \end{cases}$$

Note: observe the identity is described by $\iota = \lfloor \cdot \leftarrow \cdot \rfloor$ (parallel assignment, with empty support)

Symbolic transfer functions: example

Encoding of a few instructions:

- **"Addition"** $\ell_0 : \textbf{addi } \mathbf{r}_1, \mathbf{r}_1, v;\ \ell_1 : \ldots$:

$$\delta_{\ell_0,\ell_1} = \lfloor \mathbf{r}_0 \leftarrow \mathbf{r}_1 + v \rfloor$$

- **"Comparison"** $\ell_0 : \textbf{cmp } \mathbf{r}_0, \mathbf{r}_1;\ \ell_1 : \ldots$:

$$\begin{aligned}\delta_{\ell_0,\ell_1} = \lfloor \mathbf{r}_0 < \mathbf{r}_1 \ ?& \\ &\lfloor \mathbf{cr} \leftarrow \mathrm{LT} \rfloor \\ &\mid \lfloor \mathbf{r}_0 = \mathbf{r}_1 \ ?\ \lfloor \mathbf{cr} \leftarrow \mathrm{EQ} \rfloor \mid \lfloor \mathbf{cr} \leftarrow \mathrm{GT} \rfloor \rfloor \rfloor\end{aligned}$$

- **"Conditional branching"** $\ell_0 : \textbf{blt}\langle \mathrm{LT} \rangle\ \ell_1;\ \ell_2 : \ldots$:

$$\begin{aligned}\delta_{\ell_0,\ell_1} &= \lfloor \mathbf{cr} = \mathrm{LT} \ ?\ \iota \mid \square \rfloor \\ \delta_{\ell_0,\ell_2} &= \lfloor \mathbf{cr} = \mathrm{LT} \ ?\ \square \mid \iota \rfloor\end{aligned}$$

Symbolic transfer functions: example

Encoding of a few instructions:

- **"Load"** ℓ_0 : **ldx** $\mathbf{r}_d, o, \mathbf{r}_x$; ℓ_1 : ...:

$$\delta_{\ell_0,\ell_1} = \lfloor \mathbf{r}_d \leftarrow \mu(o + \mathbf{r}_x) \rfloor$$

- **"Load"** ℓ_0 : **ld** $\mathbf{r}_d, o$; ℓ_1 : ...:

$$\delta_{\ell_0,\ell_1} = \lfloor \mathbf{r}_d \leftarrow \mu(o) \rfloor$$

- **"Store"** ℓ_0 : **stx** $\mathbf{r}_d, o, \mathbf{r}_x$; ℓ_1 : ...:

$$\delta_{\ell_0,\ell_1} = \lfloor \mu(o + \mathbf{r}_x) \leftarrow \mathbf{r}_d \rfloor$$

The encoding of **the source semantics** is **straightforward**

Symbolic transfer functions: composition operation

Theorem

We can define a **fully syntactic composition operation** $\otimes : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$ such that:

$$[\![\delta_0 \otimes \delta_1]\!] \simeq [\![\delta_0]\!] \circ [\![\delta_1]\!]$$

Full proof left as exercise; we consider a few cases:

- $\square \otimes \delta = \square$
- $\delta \otimes \square = \square$
- $\delta \otimes \lfloor c \ ? \ \delta_0 \mid \delta_1 \rfloor = \lfloor c \ ? \ \delta \otimes \delta_0 \mid \delta \otimes \delta_1 \rfloor$
- $\lfloor x_0 \leftarrow e_0 \rfloor \otimes \lfloor x_1 \leftarrow e_1 \rfloor = \begin{cases} \lfloor x_0 \leftarrow e_0[x_1 \leftarrow e_1] \rfloor & \text{if } x_0 = x_1 \\ \left\lfloor \begin{array}{lll} x_0 & \leftarrow & e_0[x_1 \leftarrow e_1] \\ x_1 & \leftarrow & e_1 \end{array} \right\rfloor & \text{otherwise} \end{cases}$

  (note aliases must be treated with care)

Symbolic transfer functions: composition operation

Example:

- no aliasing between x, y, z
  (*i.e.*, locations x, y, z are disjoint pairwise)
- $\delta_0 = \left\lfloor \begin{array}{lll} x & \leftarrow & y + 4 \\ y & \leftarrow & 3 \end{array} \right\rfloor$
- $\delta_1 = \lfloor y \leftarrow z + 1 \rfloor$
-

Then:

$$\delta_0 \otimes \delta_1 = \left\lfloor \begin{array}{lll} x & \leftarrow & z + 5 \\ y & \leftarrow & 3 \end{array} \right\rfloor$$

Note that y is overwritten, and the expression written into x takes into account that assignment

Translation validation with symbolic transfer functions

Application of symbolic transfer functions:

Definition of **a new program (labeled transition system)** P'_c

Program Reduction

- **States:** L'_c
- **$\rightarrow$ is defined by a table of symbolic transfer functions:**
  $(l, \rho) \rightarrow (l', \rho') \iff$
  $$\begin{cases} \exists l_0, \ldots, l_n \in \mathbb{L}_c \setminus \mathbb{L}'_c, \\ \rho' = [\![\delta_{l_n, l'} \otimes \ldots \otimes \delta_{l_i, l_{i+1}} \otimes \delta_{l_{i-1}, l_i} \otimes \ldots \otimes \delta_{l, l_0}]\!](\rho) \end{cases}$$

Symbolic semantic abstraction

- **Semantics:** $[\![P'_c]\!] = \mathbf{lfp}\, F'_c$ where F^r_c is **derived from** P'_c
- **Soundness property:** $\alpha^r_c([\![P_c]\!]) = [\![P'_c]\!] = \mathbf{lfp}\, F'_c$
  Proof: by induction on the length of the traces of P'_c

Translation validation: example (condition test)

Compiled code:

```
0    ld r0, x
4    li r1, 5
8    cmp r0, r1
12   blt⟨GT⟩ ℓ     # (jump point)
16     ...# true branch contents
ℓ :  # false branch contents
```

Source code:

```
if(x ≤ 5){
    assert(x ≤ 5);
    ...
}else{
    ...
}
```

STF to the true branch:

$\delta^s = \lfloor \mathrm{x} \leq 5 \; ? \; \iota \mid \Box \rfloor$

STF to ℓ:

$$\delta^c_\ell = \left\lfloor \underline{\mathrm{x}} < 5 \; ? \left[\begin{array}{lcl} \mathbf{r}_0 & \leftarrow & \mu(\underline{\mathrm{x}}) \\ \mathbf{r}_1 & \leftarrow & 5 \\ \mathbf{cr} & \leftarrow & \mathrm{LT} \end{array} \right] \mid \ldots \right\rfloor$$

STF in P'_c:

$\delta^c_\ell = \lfloor \underline{\mathrm{x}} < 5 \; ? \; \iota \mid \lfloor \underline{\mathrm{x}} = 5 \; ? \; \iota \mid \Box \rfloor \rfloor$

Translation validation and optimization: instruction scheduling

source code

l_0^s $\mathtt{i} := \mathtt{i} + 1;$

l_1^s $\mathtt{x} := \mathtt{x} + \mathtt{t}[\mathtt{i}];$

l_2^s $\ldots$

optimized code

l_0^o **ld** $\mathbf{r}_0, \underline{\mathtt{i}}$
l_1^o **ld** $\mathbf{r}_1, \underline{\mathtt{x}}$
l_2^o **addi** $\mathbf{r}_0, \mathbf{r}_0, 1$
l_3^o **ldx** $\mathbf{r}_2, \underline{\mathtt{t}}, \mathbf{r}_0$
l_4^o **st** $\mathbf{r}_0, \underline{\mathtt{i}}$
l_5^o **add** $\mathbf{r}_1, \mathbf{r}_1, \mathbf{r}_2$
l_6^o **st** $\mathbf{r}_1, \underline{\mathtt{x}}$
l_7^o $\ldots$

Syntactic mappings:

$$\pi_{\mathbb{X}\times\mathbb{X}} : \begin{array}{rcl} (l_0^s, \mathtt{i}) & \mapsto & (l_0^o, \underline{\mathtt{i}}) \\ (l_0^s, \mathtt{x}) & \mapsto & (l_0^o, \underline{\mathtt{x}}) \\ (l_1^s, \mathtt{i}) & \mapsto & (l_5^o, \underline{\mathtt{i}}) \\ (l_1^s, \mathtt{x}) & \mapsto & (l_1^o, \underline{\mathtt{x}}) \\ (l_2^s, \mathtt{i}) & \mapsto & (l_7^o, \underline{\mathtt{i}}) \\ (l_2^s, \mathtt{x}) & \mapsto & (l_7^o, \underline{\mathtt{x}}) \end{array}$$

Thus, $l_f^o = \mathtt{i}@l_5^o; \mathtt{x}@l_1^o$

- **Source level transfer functions:**

$$\delta_{l_0^s, l_1^s} = \lfloor \mathtt{i} \leftarrow \mathtt{i} + 1 \rfloor \qquad \delta_{l_1^s, l_2^s} = \lfloor \mathtt{x} \leftarrow \mathtt{x} + \mathtt{t}[\mathtt{i}] \rfloor$$

- **Optimized level transfer functions** (registered not displayed):

$$\delta_{l_0^o, l_f^o} = \lfloor \mu(\mathtt{i}) \leftarrow \mu(\mathtt{i}) + 1 \rfloor \qquad \delta_{l_f^o, l_7^o} = \lfloor \mu(\underline{\mathtt{x}}) \leftarrow \mu(\underline{\mathtt{x}}) + \mu(\underline{\mathtt{t}} + \mu(\underline{\mathtt{i}})) \rfloor$$

Translation validation and optimizations

Program reduction:

- produces a **set of symbolic transfer functions** that encode the transition relation of the program **up-to observational abstraction**
- **abstracts the effect of optimizations**
  as in the instruction scheduling example
  loop unrolling would result into unrolling at the source level (partitioning)

Translation validation:

- based on a **specialized prover**, to establish **equivalence of transfer functions**

Outline

Conclusion

Formalization of Compilation:

- At the **concrete level**: independent from analysis
- Very **broad**; works as well for
  - other architectures
  - optimizations (use of other abstractions)

Algorithms for certified compilation described in **the abstract interpretation frameworks**:

- **Invariant translation**
- **Invariant checking**
- **Translation validation**
- **Compiler formal certification**

Symbolic transfer functions and use in **static analysis** and **program transformations**.

This approach applies to other program transformations

Homework

1. Formalize the dead variable elimination correctness (P. 46)
2. **Read:**

P. Cousot and R. Cousot.
Systematic design of program transformation frameworks by abstract interpretation.
In *Conference Record of the 29th Symposium on Principles of Programming Languages (POPL'02)*, pages 178–190, Portland, Oregon, January 2002.

Semantics

- **Program transformations: P. Cousot and R. Cousot.**

  Systematic design of program transformation frameworks by abstract interpretation.

  In *Conference Record of the 29th Symposium on Principles of Programming Languages (POPL'02)*, pages 178–190, Portland, Oregon, January 2002.

- **Relation between types and static analysis:**

  P. Cousot,

  Types as Abstract Interpretations.

  In *POPL'97*, pages 316–331, Paris, January 1997.

- **Symbolic transfer functions:**

  C. Colby and P. Lee.

  Trace-based program analysis.

  In *23rd POPL*, pages 195–207, St. Petersburg Beach, (Florida USA), 1996.

Bibliography: Certified Compilation

- **Proof Carrying Codes:** **G. C. Necula.**
  Proof-Carrying Code.
  In *24th POPL*, pages 106–119, 1997.
- **Typed Assembly languages:**
  G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee.
  The TIL/ML Compiler: Performance and Safety Through Types.
  In *WCSSS*, 1996.
- **Abstract invariant translation (after compilation):**
  X. Rival.
  Abstract Interpretation-based Certification of Assembly Code.
  In *4th VMCAI*, New York (USA), 2003.

Bibliography: Certified Compilation

- **Translation validation: A. Pnueli, O. Strichman, and M. Siegel.**
  Translation Validation for Synchronous Languages.
  In *ICALP'98*, pages 235–246. Springer-Verlag, 1998.
- **Formal proof:**
  X. Leroy.
  Formal certification of a compiler back-end, or: programming a compiler with a proof assistant.
  In *POPL'06*, Charleston, january 2006.
- **A generic frameork:**
  X. Rival.
  Symbolic-Transfer Function-Based Approaches to Compilation Certification
  In *POPL'04*, Venice, january 2004.