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, 21st. 2015
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:

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

Compiled code:

```assembly
.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
    movl %ecx, %eax
    movl %eax, -4(%ebp)
    jmp .L3
    .L2:
    movl $0, %eax
    .L3:
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .LFE0:
    .size f, .-f
    .ident
    "GCC: (Gentoo 4.7.3-r1 p1.4, pie-0.5.5) 4.7.3"
    .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...
Introduction to program transformations

Slicing

Slice extraction

- A slice $S$ is a **syntactic subset of a program** $P$
- 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);
2:    input(y);
3:    a = 4 * x + 8;    → 3:    a = 4 * x + 8;
4:    b = 3 - 2 * y + a;
5:    c = a + b;
```

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
Introduction to program transformations

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(b){
        x = 1;
    }else{
        x = f(x);
    }
    b = false;
}

if(c){
    x = 1;
}
```

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 Miscompiled, 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:

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

Only ONE store 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
Compilation correctness

Outline
Compilation correctness

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** $S = (S, \rightarrow, S_I)$:

- $S = L \times M$ is the set of **states**, where $M = X \rightarrow V$
- $\rightarrow \subseteq S \times S$ is the **transition relation**
- $S_I \subseteq S$ is the set of **initial states**

We consider their **finite traces semantics**:

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

\[
F : \mathcal{P}(S^*) \longrightarrow \mathcal{P}(S^*)
\]

$X \longmapsto \{\langle s_0 \rangle \mid s \in S_I\}$

$\cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle \mid \langle s_0, \ldots, s_n \rangle \in X \land s_n \rightarrow s_{n+1}\}$

(exercise)
A very minimal imperative language

\[ l ::= \text{l-values} \]

\[ | \quad x \quad (x \in X) \]

\[ e ::= \text{expressions} \]

\[ | \quad c \quad (c \in \mathbb{V}) \]

\[ | \quad l \quad (lvalue) \]

\[ | \quad e \oplus e \quad (\text{arithoperation, comparison}) \]

\[ s ::= \text{statements} \]

\[ | \quad l = e \quad (\text{assignment}) \]

\[ | \quad s; \ldots s; \quad (\text{sequence}) \]

\[ | \quad \text{if}(e)\{s\} \quad (\text{condition}) \]

\[ | \quad \text{while}(e)\{s\} \quad (\text{loop}) \]

Other extensions, not considered at this stage:

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

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: $\mathbb{I}_{MIPS}$)
  and stored into the same space as data (i.e., $\mathbb{I}_{MIPS} \subseteq \mathbb{B}^{32}$)
- 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 $r_0, \ldots, r_{31}$
- main memory (RAM) $\text{Addr} \rightarrow \mathbb{B}^{32}$ where $\text{Addr} \subseteq \mathbb{B}^{32}$
- condition register $cr$

Then: $X^c = \{pc, cr, r_0, \ldots, r_{31}\} \uplus \text{Addr}$
A basic, PPC-like assembly language: instruction set

Instruction encoded into 32-bits words:

\[ \nu, \text{dst}, o \in \mathbb{B}^{32}, \quad \text{cr} \in \{\text{LT}, \text{EQ}, \text{GT}\} \]

\[ i ::= (\in I_{\text{MIPS}}) \]

- \( \text{li } r_d, \nu \): load \( \nu \in \mathbb{B}^{32} \)
- \( \text{add } r_d, r_{s0}, r_{s1} \): addition
- \( \text{addi } r_d, r_{s0}, \nu \): add. \( \nu \in \mathbb{B}^{32} \)
- \( \text{sub } r_d, r_{s0}, r_{s1} \): subtraction
- \( \text{cmp } r_{s0}, r_{s1} \): comparison
- \( b \text{ dst} \): branch
- \( \text{blt} \langle \text{cr} \rangle \text{ dst} \): cond. branch
- \( \text{ld } r_d, o \): absolute load
- \( \text{st } r_d, o \): absolute store
- \( \text{ldx } r_d, o, r_x \): relative load (aka indeXed load)
- \( \text{stx } r_d, o, r_x \): relative store (aka indeXed store)
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 $cr \in \{LT, EQ, GT\}$
- a function mapping each **memory cell** to its value $\mu : \text{Addrs} \rightarrow \mathbb{B}^{32}$

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

- the **control state** $\ell$ is the current $pc$ value
- the **memory state** $m$ is the triple $(\rho, cr, \mu)$

(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 $i = \text{li } r_d, v$, then:
  $$ s \rightarrow (pc + 4, \rho[d \mapsto v], cr, \mu) $$

- if $i = \text{add } r_d, r_{s0}, r_{s1}$, then:
  $$ s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) + \rho(s1))], cr, \mu) $$

- if $i = \text{addi } r_d, r_{s0}, v$, then:
  $$ s \rightarrow (pc + 4, \rho[d \mapsto (\rho(s0) + v)], cr, \mu) $$

- if $i = \text{sub } r_d, r_{s0}, r_{s1}$, 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 $i = \text{cmp} \ r_{s0}, r_{s1}$, then:
  
  $$ s \rightarrow \left\{ \begin{array}{l}
  (pc + 4, \rho, \text{LT}, \mu) \quad \text{if } \rho(s0) < \rho(s1) \\
  (pc + 4, \rho, \text{EQ}, \mu) \quad \text{if } \rho(s0) = \rho(s1) \\
  (pc + 4, \rho, \text{GT}, \mu) \quad \text{if } \rho(s0) > \rho(s1)
  \end{array} \right. $$

- if $i = \text{blt} \langle \text{cond} \rangle \ dst$, then:
  
  $$ s \rightarrow \left\{ \begin{array}{l}
  (dst, \rho, \text{cr}, \mu) \quad \text{if } \text{cr} = \text{cond} \\
  (pc + 4, \rho, \text{cr}, \mu) \quad \text{otherwise}
  \end{array} \right. $$

- if $i = \text{b} \ dst$, then:
  
  $$ s \rightarrow (dst, \rho, \text{cr}, \mu) $$
A basic, PPC-like assembly language: instruction set

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

Then:

- if \( i = \text{ldx } r_d, o, r_x \), then:
  \[
  s \rightarrow \begin{cases} 
  (pc + 4, \rho[d \mapsto \mu(\rho(x) + o)], cr, \mu) & \text{if } \mu(\rho(x) + o) \text{ is defined} \\
  \Omega & \text{otherwise}
  \end{cases}
  \]

- if \( i = \text{ld } r_d, o \), then:
  \[
  s \rightarrow \begin{cases} 
  (pc + 4, \rho[d \mapsto \mu(o)], cr, \mu) & \text{if } \mu(o) \text{ is defined} \\
  \Omega & \text{otherwise}
  \end{cases}
  \]

- if \( i = \text{stx } r_d, o, r_x \), then:
  \[
  s \rightarrow \begin{cases} 
  (pc + 4, \rho, cr, \mu[\rho(x) + o] \mapsto \rho(d)]) & \text{if } \mu(\rho(x) + o) \text{ is defined} \\
  \Omega & \text{otherwise}
  \end{cases}
  \]

- if \( i = \text{ld } r_d, o \), then effect can be deduced from the above cases
Compilation correctness

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, \( x \) denotes the address of \( x \)

Source code

\[ \begin{align*}
\ell_0^s & \quad i := i + 1; \\
\ell_1^s & \quad x := x + t[i]; \\
\ell_2^s & \quad \ldots
\end{align*} \]

Compiled code

\[ \begin{align*}
\ell_0^c & \quad \text{ld } r_0, i \\
\ell_1^c & \quad \text{addi } r_0, r_0, 1 \\
\ell_2^c & \quad \text{st } r_0, i \\
\ell_3^c & \quad \text{ld } r_0, x \\
\ell_4^c & \quad \text{ld } r_1, i \\
\ell_5^c & \quad \text{ldx } r_2, t, r_1 \\
\ell_6^c & \quad \text{add } r_0, r_0, r_2 \\
\ell_7^c & \quad \text{st } r_0, x \\
\ell_8^c & \quad \ldots
\end{align*} \]

Is it sound? What property does it preserve?
Compilation correctness

A source level execution

\[
\langle l_0^s, \begin{cases} 
i \mapsto 1; \\
x \mapsto 1; \\
t[0] \mapsto 0; \\
t[1] \mapsto 1; \\
t[2] \mapsto 4; \\
t[3] \mapsto -1; \end{cases} \rangle, \quad \langle l_1^s, \begin{cases} 
i \mapsto 2; \\
x \mapsto 1; \\
t[0] \mapsto 0; \\
t[1] \mapsto 1; \\
t[2] \mapsto 4; \\
t[3] \mapsto -1; \end{cases} \rangle, \quad \langle l_2^s, \begin{cases} 
i \mapsto 2; \\
x \mapsto 5; \\
t[0] \mapsto 0; \\
t[1] \mapsto 1; \\
t[2] \mapsto 4; \\
t[3] \mapsto -1; \end{cases} \rangle
\]

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

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

<table>
<thead>
<tr>
<th>state $s_i^c$</th>
<th>$s_0^c$</th>
<th>$s_1^c$</th>
<th>$s_2^c$</th>
<th>$s_3^c$</th>
<th>$s_4^c$</th>
<th>$s_5^c$</th>
<th>$s_6^c$</th>
<th>$s_7^c$</th>
<th>$s_8^c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>control state $pc_i$</td>
<td>$l_0^c$</td>
<td>$l_1^c$</td>
<td>$l_2^c$</td>
<td>$l_3^c$</td>
<td>$l_4^c$</td>
<td>$l_5^c$</td>
<td>$l_6^c$</td>
<td>$l_7^c$</td>
<td>$l_8^c$</td>
</tr>
<tr>
<td>register state $\rho_i(0)$</td>
<td>45</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>register state $\rho_i(1)$</td>
<td>$-5$</td>
<td>$-5$</td>
<td>$-5$</td>
<td>$-5$</td>
<td>$-5$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>register state $\rho_i(2)$</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(\overline{i})$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>memory state $\mu_i(x)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 0)$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 1)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 2)$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 3)$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
</tbody>
</table>
## Source and assembly executions compared

<table>
<thead>
<tr>
<th>state $s_i^s$</th>
<th>$s_0^s$</th>
<th>$s_1^s$</th>
<th>$s_2^s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>control state $l_i^s$</td>
<td>$l_0^s$</td>
<td>$l_1^s$</td>
<td>$l_2^s$</td>
</tr>
<tr>
<td>memory state $m_i^s(i)$</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>memory state $m_i^s(x)$</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>memory state $m_i^s(t[0])$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory state $m_i^s(t[1])$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $m_i^s(t[2])$</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $m_i^s(t[3])$</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state $s_i^c$</th>
<th>$s_0^c$</th>
<th>$s_1^c$</th>
<th>$s_2^c$</th>
<th>$s_3^c$</th>
<th>$s_4^c$</th>
<th>$s_5^c$</th>
<th>$s_6^c$</th>
<th>$s_7^c$</th>
<th>$s_8^c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>control state $pc_i$</td>
<td>$l_0^c$</td>
<td>$l_1^c$</td>
<td>$l_2^c$</td>
<td>$l_3^c$</td>
<td>$l_4^c$</td>
<td>$l_5^c$</td>
<td>$l_6^c$</td>
<td>$l_7^c$</td>
<td>$l_8^c$</td>
</tr>
<tr>
<td>register state $\rho_i(0)$</td>
<td>45</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>register state $\rho_i(1)$</td>
<td>−5</td>
<td>−5</td>
<td>−5</td>
<td>−5</td>
<td>−5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>register state $\rho_i(2)$</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(i)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(x)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 0)$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 1)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 2)$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(t + 3)$</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
</tr>
</tbody>
</table>

Much more information in the assembly trace:

- registers values
- more control states
We can abstract away intermediate control states
Compilation correctness

An abstraction approach

- Intermediate control states abstracted; we can forget registers
An abstraction approach

<table>
<thead>
<tr>
<th>state $s^s_i$</th>
<th>$s^s_0$</th>
<th>$s^s_1$</th>
<th>$s^s_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>control state $l^s_i$</td>
<td>$l^s_0$</td>
<td>$l^s_1$</td>
<td>$l^s_2$</td>
</tr>
<tr>
<td>memory state $m^s_i(i)$</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>memory state $m^s_i(x)$</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>memory state $m^s_i(t[0])$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory state $m^s_i(t[1])$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $m^s_i(t[2])$</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $m^s_i(t[3])$</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state $s^c_i$</th>
<th>$s^c_0$</th>
<th>$s^c_1$</th>
<th>$s^c_2$</th>
<th>$s^c_3$</th>
<th>$s^c_4$</th>
<th>$s^c_5$</th>
<th>$s^c_6$</th>
<th>$s^c_7$</th>
<th>$s^c_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>control state $pc_i$</td>
<td>$l^c_0$</td>
<td>$l^c_1$</td>
<td>$l^c_2$</td>
<td>$l^c_3$</td>
<td>$l^c_4$</td>
<td>$l^c_5$</td>
<td>$l^c_6$</td>
<td>$l^c_7$</td>
<td>$l^c_8$</td>
</tr>
<tr>
<td>register state $\rho_i(0)$</td>
<td>45</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>register state $\rho_i(1)$</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>register state $\rho_i(2)$</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(i)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>memory state $\mu_i(x)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>memory state $\mu_i(t[0])$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory state $\mu_i(t[1])$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(t[2])$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(t[3])$</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

- Registers and intermediate control states removed

We get very similar traces!
Compilation correctness

Syntactic relations

What we did remove:
- intermediate control states
- memory locations associated to registers

What we did preserve:
- control states in correspondence:
  \[ l_0^s \leftrightarrow l_0^c \quad l_1^s \leftrightarrow l_3^c \quad l_2^s \leftrightarrow l_8^c \]
- memory location in correspondence:
  \[
  i \leftrightarrow i \\
  t[0] \leftrightarrow t + 0 \\
  t[1] \leftrightarrow t + 1 \\
  t[2] \leftrightarrow t + 2 \\
  t[3] \leftrightarrow t + 3
  \]

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

Syntactic relations

Definition

We define two syntactic mappings:

- **Between control points:** $\pi_l : \mathbb{L}'_s \to \mathbb{L}'_c$ (where $\mathbb{L}'_i \subseteq \mathbb{L}_i$)
- **Between memory locations:** $\pi_x : \mathbb{X}'_s \to \mathbb{X}'_c$ (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, $\pi_l, \pi_x$ 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:**
  - $\mathbb{X}_s' = \{i, x, t[0], t[1], t[2], t[3]\}$ and $\mathbb{X}_c' = \{i, x, t, t + 1, t + 2, t + 3\}$
  - $\pi_x : \begin{cases} i \mapsto i \\ x \mapsto x \\ t[n] \mapsto t + n \end{cases}$
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 $\pi^m_c$ be defined by:

$$\pi^m_c : \mathcal{X}_c \to \mathbb{V} \quad \mapsto \quad (\mathcal{X}_c' \to \mathbb{V})$$

$$m \quad \mapsto \quad \lambda (x \in \mathcal{X}_c') \cdot m(x)$$

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

\[ X'_s = \{ i, x, t[0], t[1], t[2], t[3] \} \]
\[ X'_c = \{ i, x, t, t + 1, t + 2, t + 3 \} \]

Then \( \pi^m_c : (pc, \rho, cr, \mu) \mapsto \mu \)

So, in particular:

\[
\pi^m_c : \left( \begin{array}{c}
pc \\
\rho : & 0 & \mapsto & 45 \\
& 1 & \mapsto & -5 \\
& 2 & \mapsto & 4 \\
\mu : & i & \mapsto & 1 \\
x & \mapsto & 1 \\
\overline{t} + 0 & \mapsto & 0 \\
\overline{t} + 1 & \mapsto & 1 \\
\overline{t} + 2 & \mapsto & 4 \\
\overline{t} + 3 & \mapsto & -1 \\
\end{array} \right)
\mapsto \left( \begin{array}{c}
\mu : & i & \mapsto & 1 \\
x & \mapsto & 1 \\
\overline{t} + 0 & \mapsto & 0 \\
\overline{t} + 1 & \mapsto & 1 \\
\overline{t} + 2 & \mapsto & 4 \\
\overline{t} + 3 & \mapsto & -1 \\
\end{array} \right)
\]
Trace observational abstraction

We can now lift the same abstraction principle to traces:

**Definition: trace abstraction**

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

$$
\pi_c^{tr} : (\mathbb{L}_c \times (X_c \rightarrow \mathbb{V}))^* \rightarrow (\mathbb{L}'_c \times (X'_c \rightarrow \mathbb{V}))^*
$$

$$
\langle (l_0, m_0), \ldots, (l_n, m_n) \rangle \mapsto \langle (l_{k_0}, \pi_c^m(m_{k_0})), \ldots, (l_{k_m}, \pi_c^m(m_{k_m})) \rangle
$$

where:

$$
\left\{ k_0, \ldots, k_m \right\} = \left\{ k \mid 0 \leq k \leq n \land l_k \in \mathbb{L}'_c \right\}
$$

$$
k_0 < \ldots < k_m
$$

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

<table>
<thead>
<tr>
<th>control state $pc_i$</th>
<th>$l^c_0$</th>
<th>$l^c_1$</th>
<th>$l^c_2$</th>
<th>$l^c_3$</th>
<th>$l^c_4$</th>
<th>$l^c_5$</th>
<th>$l^c_6$</th>
<th>$l^c_7$</th>
<th>$l^c_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>register state $\rho_i(0)$</td>
<td>45</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>register state $\rho_i(1)$</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>-5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>register state $\rho_i(2)$</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(\overline{i})$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>memory state $\mu_i(\overline{x})$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(t+0)$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory state $\mu_i(t+1)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(t+2)$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(t+3)$</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

$\pi^{\text{tr}}$: $\overrightarrow{\text{control state } pc_i}$

<table>
<thead>
<tr>
<th>control state $pc_i$</th>
<th>$l^c_0$</th>
<th>$l^c_3$</th>
<th>$l^c_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory state $\mu_i(\overline{i})$</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>memory state $\mu_i(\overline{x})$</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>memory state $\mu_i(t+0)$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory state $\mu_i(t+1)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>memory state $\mu_i(t+2)$</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>memory state $\mu_i(t+3)$</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Compilation correctness

Observable behaviors inclusions

Applying this systematically to all traces results into an abstraction:

Result: compiled code observational abstraction

We let $\alpha^r_c$ be the compiled code observational abstraction:

$$\alpha^r_c : \mathcal{P}(\mathbb{L}_c \times (X_c \rightarrow V))^*_{\mathcal{E}} \rightarrow \mathcal{P}(\mathbb{L}'_c \times (X'_c \rightarrow V))^*_{\mathcal{E}}$$

It defines a Galois connection with an adjoint concretization $\gamma^r_c$:

$$(\mathcal{P}(\mathbb{L}_c \times (X_c \rightarrow V))^*, \subseteq) \xleftarrow{\alpha^r_c} (\mathcal{P}(\mathbb{L}'_c \times (X'_c \rightarrow V))^*, \subseteq) \xrightarrow{\gamma^r_c}$$

- $\alpha^r_c$ is monotone and the concrete domain is a complete lattice; the concretization function follows and is defined by
  $$\gamma^r_c(\mathcal{E}') = \bigcup_{\mathcal{E}} \{ \mathcal{E} | \alpha^r_c(\mathcal{E}) \subseteq \mathcal{E}' \} = \{ \sigma | \pi^\text{tr}(\sigma) \in \mathcal{E}' \}$$
- The observational semantics is defined by: $\llbracket P_c \rrbracket_{\text{obs}} = \alpha^r_c(\llbracket P_c \rrbracket)$
Compilation correctness

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 $\pi_I, \pi_X$ are bijective

Memory state renaming

We let the memory state renaming function be defined by:

$$
\pi_m : (X'_s \to \mathbb{V}) \rightarrow (X'_c \to \mathbb{V}) \\
\quad m \quad \mapsto m \circ \pi_X^{-1}
$$

Trace renaming

We let the trace renaming function be defined by:

$$
\pi_t : \mathcal{L}'_s \times (X'_s \to \mathbb{V}) \rightarrow \mathcal{L}'_c \times (X'_c \to \mathbb{V}) \\
\langle (l_0, m_0), \ldots, (l_n, m_n) \rangle \mapsto \langle (\pi_I(l_0), \pi_m(m_0)), \ldots, (\pi_I(l_n), \pi_m(m_n)) \rangle
$$
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 $\pi_1, \pi_x$ if and only if $\pi_t$ establishes a bijection between $\alpha^r_s([P_s])$ and $\alpha^r_c([P_c])$.

This definition can be illustrated by the diagram:

$$
\alpha^r_s([P_s]) \xrightarrow{\pi_t} \alpha^r_c([P_c])
$$
Compilation correctness

Correctness by semantic equivalence

This approach generalizes to other program transformations

This definition can be illustrated by the diagram:

\[
\begin{align*}
\alpha_s([P_i]) & \quad \xrightarrow{\text{semantic transformation}} \quad \alpha_c([P_o]) \\
observation & \quad \uparrow \quad \text{observation} \\
[P_i] & \quad \uparrow \quad [P_o] \\
\text{semantics} & \quad \uparrow \quad \text{semantics} \\
P_i & \quad \text{transformation} \quad \rightarrow \quad P_o
\end{align*}
\]
Choice of another concrete semantics: consequences

New compilation correctness definition

\[ \forall \rho \in \mathbb{M}, \ [P_c]_{\text{rel}} \equiv [P_s]_{\text{rel}} \mod \pi_I, \pi_X \]

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 \( \infty \ldots \)
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: \( L_+ \) (resp. \( L_- \))
- Notation for non-termination written \( \infty \);
- Observational semantics: relations between \( M \) and \( M \cup \{ \infty \} \)
- Observational abstraction defined by collecting for all traces:

\[
\langle (L_+, \rho), \ldots, (L_+, \rho') \rangle \rightarrow (\rho, \rho')
\]

\[
\sigma = \langle (L_-, \rho), \ldots \rangle \rightarrow (\rho, \infty) \text{ if } \sigma \text{ infinite}
\]

- Denotational semantics defined by:

\[
[P]_{rel} = \{(\rho, \rho') | \ldots\} \cup \{(\rho, \infty) | \ldots\}
\]
Outline
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**: $\pi_l$, $\pi_x$, $L_i'$, $X_i'$ 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:**

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Compiled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_0^s$ x := 4;</td>
<td>$l_0^c$ li r, 4</td>
</tr>
<tr>
<td>$l_1^s$ if(x &lt; 0){</td>
<td>$l_1^c$ st r, x</td>
</tr>
<tr>
<td>$l_2^s$ x = -x;</td>
<td></td>
</tr>
<tr>
<td>$l_3^s$</td>
<td></td>
</tr>
<tr>
<td>$l_4^s$ x = x + 1</td>
<td>$l_2^c$ ld r, x</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Correctness of optimizing compilation

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

```plaintext
...  
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*
Correctness of optimizing compilation

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 $X_s$ altogether: in the example, this would not be correct, as $y$ would be completely lost
- thus, $\pi_X$ should be relational

Compilation correctness in presence of variable-code elimination

Similar definition as before, but with $\pi_X : \mathbb{I}_s \times X_s \to X_c$ instead.

Exercise: formalize the new definition, inspired from the previous one, and with $\pi_X : \mathbb{I}_s \times X_s \to X_c$ 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
\[
\begin{align*}
\ell_0^s &: i := 0; \\
\ell_1^s &: \textbf{while}(i < 1000) \\
\ell_2^s &: x := x \ast y; \\
\ell_3^s &: y := y - 1; \\
\ell_4^s &: i := i + 1; \\
\ell_5^s &: \}\end{align*}
\]

optimized code
\[
\begin{align*}
\ell_0^o &: i := 0; \\
\ell_1^o &: \textbf{while}(i < 500) \\
\ell_2^o &: x := x \ast y; \\
\ell_3^o &: y := y - 1; \\
\ell_4^o &: x := x \ast y; \\
\ell_5^o &: y := y - 1; \\
\ell_6^o &: i := i + 1; \\
\ell_7^o &: \}\end{align*}
\]

Control state correspondance $\pi_I$ is clearly broken:

\[\pi_I : \begin{cases} 
\ell_2^s & \leftrightarrow \ell_2^o \\
\ell_2^s & \leftrightarrow \ell_4^o 
\end{cases}\]
Correctness of optimizing compilation

Loop unrolling source and assembly traces

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

- \textbf{At the source code level:}

<table>
<thead>
<tr>
<th>control state</th>
<th>$l_2^s$</th>
<th>$l_2^s$</th>
<th>$l_2^s$</th>
<th>$l_2^s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of $i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>value of $y$</td>
<td>1200</td>
<td>1199</td>
<td>1198</td>
<td>1197</td>
</tr>
</tbody>
</table>

- \textbf{At the compiled code level:}

<table>
<thead>
<tr>
<th>control state</th>
<th>$l_2^o$</th>
<th>$l_4^o$</th>
<th>$l_2^o$</th>
<th>$l_4^o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of $i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>value of $y$</td>
<td>1200</td>
<td>1199</td>
<td>1198</td>
<td>1197</td>
</tr>
</tbody>
</table>

As expected:

- \textbf{the correlation} between the values of $i$ and the other variables is \textbf{lost}
- \textbf{the real correspondence} is between \textbf{values of other variables} and \textbf{iterations even-ness}
Correctness of optimizing compilation

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 $X_s, X_o$
- each control state in the source loop should be divided into a pair of labels, that carry an even-ness tab:

$$
\begin{align*}
&\ell_2^s \mapsto \ell_2^{s,e}, \ell_2^{s,o} \\
&\ell_3^s \mapsto \ell_3^{s,e}, \ell_3^{s,o} \\
&\ldots \mapsto \ldots
\end{align*}
$$

- the trace abstraction function $\pi^\text{tr}_s$ 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:

<table>
<thead>
<tr>
<th>source code</th>
<th>control state</th>
<th>( l_2^s )</th>
<th>( l_2^s )</th>
<th>( l_2^s )</th>
<th>( l_2^s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of ( i )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>value of ( y )</td>
<td>1200</td>
<td>1199</td>
<td>1198</td>
<td>1197</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>source code, abstract</th>
<th>control state</th>
<th>( l_2^{s,e} )</th>
<th>( l_2^{s,o} )</th>
<th>( l_2^{s,e} )</th>
<th>( l_2^{s,o} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of ( i )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>value of ( y )</td>
<td>1200</td>
<td>1199</td>
<td>1198</td>
<td>1197</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>optimized code</th>
<th>control state</th>
<th>( l_2^o )</th>
<th>( l_4^o )</th>
<th>( l_2^o )</th>
<th>( l_4^o )</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of ( i )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>value of ( y )</td>
<td>1200</td>
<td>1199</td>
<td>1198</td>
<td>1197</td>
<td></td>
</tr>
</tbody>
</table>

We observe the following control state correspondance:

\[
\pi_1 : \quad l_2^{s,e} \quad \longmapsto \quad l_2^o \\
\quad l_4^{s,o} \quad \longmapsto \quad l_4^o
\]
Then, the definition follows a **very similar form** as before:

**Compilation correctness in presence of loop unrolling**

- trace partitioning $\alpha^r_s$ abstraction
- a mapping $\pi_l$ that preserves even-ness
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:

```
ld r0, i
addi r0, r0, 0
st r0, i
```

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**
Correctness of optimizing compilation

Instruction scheduling example

**source code**

\[ l^s_0 \quad i := i + 1; \]

\[ l^s_1 \quad x := x + t[i]; \]

\[ l^s_2 \quad \ldots \]

**non optimized code**

\[ l^a_0 \quad ld \ r_0, \ i \]

\[ l^a_1 \quad addi \ r_0, r_0, 1 \]

\[ l^a_2 \quad st \ r_0, \ i \]

\[ l^a_3 \quad ld \ r_1, x \]

\[ l^a_4 \quad ldx \ r_2, t, r_0 \]

\[ l^a_5 \quad add \ r_1, r_1, r_2 \]

\[ l^a_6 \quad st \ r_1, x \]

\[ l^a_7 \quad \ldots \]

**optimized code**

\[ l^o_0 \quad ld \ r_0, \ i \]

\[ l^o_1 \quad ld \ r_1, x \]

\[ l^o_2 \quad addi \ r_0, r_0, 1 \]

\[ l^o_3 \quad ldx \ r_2, t, r_0 \]

\[ l^o_4 \quad st \ r_0, \ i \]

\[ l^o_5 \quad add \ r_1, r_1, r_2 \]

\[ l^o_6 \quad st \ r_1, x \]

\[ l^o_7 \quad \ldots \]

Without optimization:

4 stall cycles, 14 cycles total

Without optimization:

2 stall cycles, 12 cycles total

\[ l^s_0 \leftrightarrow l^a_0 \]

\[ l^s_1 \leftrightarrow l^a_3 \]

\[ l^s_2 \leftrightarrow l^a_7 \]
Correctness of optimizing compilation

Instruction scheduling observational abstractions

Issues to fix our definition:

- **Instructions execution order modified:**
  \[ l_1^a \rightarrow l_2^a \text{ and } l_2^a \rightarrow l_3^a \] are postponed

- **Mapping \( \pi_l \) 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** \((\pi_l, \pi_x)\),
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

\[
\pi_{X \times X} : (L'_s \times X'_s) \rightarrow (L_c \times X_c)
\]
**Intuition**

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

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Optimized Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell^s_0$ $i := i + 1;$</td>
<td>$\ell^o_0$ ld $r_0, i$</td>
</tr>
<tr>
<td>$\ell^s_1$ $x := x + t[i]$;</td>
<td>$\ell^o_1$ ld $r_1, x$</td>
</tr>
<tr>
<td>$\ell^s_2$ ...</td>
<td>$\ell^o_2$ addi $r_0, r_0, 1$</td>
</tr>
<tr>
<td>$\ell^s_3$ $x := x + t[i]$;</td>
<td>$\ell^o_3$ ld $r_2, t, r_0$</td>
</tr>
<tr>
<td>$\ell^s_4$ $st r_0, i$</td>
<td>$\ell^o_4$ st $r_0, i$</td>
</tr>
<tr>
<td>$\ell^s_5$ $add r_1, r_2$</td>
<td>$\ell^o_5$ add $r_1, r_1, r_2$</td>
</tr>
<tr>
<td>$\ell^s_6$ $st r_1, x$</td>
<td>$\ell^o_6$ st $r_1, x$</td>
</tr>
<tr>
<td>$\ell^s_7$ ...</td>
<td>$\ell^o_7$ ...</td>
</tr>
</tbody>
</table>

We then have:

$$\pi_{X \times X} : (\ell^s_0, i) \mapsto (\ell^o_0, i)$$

$$\pi_{X \times X} : (\ell^s_0, x) \mapsto (\ell^o_0, x)$$

$$\pi_{X \times X} : (\ell^s_1, i) \mapsto (\ell^o_5, i)$$

$$\pi_{X \times X} : (\ell^s_1, x) \mapsto (\ell^o_1, x)$$

$$\pi_{X \times X} : (\ell^s_2, i) \mapsto (\ell^o_7, i)$$

$$\pi_{X \times X} : (\ell^s_2, x) \mapsto (\ell^o_7, x)$$
Instruction scheduling correctness

The source level observational abstraction is unchanged.

**Optimized level observational abstraction**

Optimized code observational abstraction $\alpha^r_s$ 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** $\alpha^r_s$ derived from $\pi_{X \times X}$
- **semantic mapping** $\pi_t$ derived from $\pi_{X \times X}$
Definition: compilation correctness

Compilation of $P_s$ into $P_c$ is correct with respect to $\pi_I, \pi_X$ (resp., $\pi_X \times X$) if and only if $\pi_t$ establishes a bijection between $\alpha^r_s([P_s])$ and $\alpha^r_c([P_c])$.

\[
\alpha^r_s([P_s]) \xrightarrow[\pi_t]{} \alpha^r_c([P_c])
\]

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 $I_s$
2. **Translate** $I_s$ into assembly level candidate invariant $I_t$
3. Perform an **assembly level check** of $I_t$

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)
Examples: 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 $\Rightarrow$ 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
Example

Start with **invariants on the source code**

- \(0 \leq x + y \leq 9\)
- \(x := x + 6;\)
- \(6 \leq x + y \leq 15\)

- \(l_0^c\)
  - \(ld r_0, x\)
- \(l_1^c\)
  - \(addi r_0, r_0, 6\)
- \(l_2^c\)
  - \(st r_0, x\)
- \(l_3^c\)
Example

Translates those invariants
but not all control states are decorated
Example

- Propagates the invariants and computes refined local invariants
Example

- Propagates the invariants and computes refined local invariants
Example

- Propagates the invariants and computes refined local invariants
Example

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

0 ≤ x + y ≤ 9

\[ x := x + 6; \]

6 ≤ x + y ≤ 15

0 ≤ x + y ≤ 9

ld r₀, x

addi r₀, r₀, 6

st r₀, x

6 ≤ x + y ≤ 15

0 ≤ x + y ≤ 9 ∧ r₀ = x

6 ≤ x + y ≤ 15 ∧ r₀ = x + 4

0 ≤ x + y ≤ 9 ∧ r₀ = x

6 ≤ x + y ≤ 15 ∧ r₀ = x
Source static analysis: assumptions

- We assume an abstraction of sets of stores defined by an abstraction function for sets of stores
  \[ \alpha_{\text{num}} : (\mathcal{P}(M_s), \subseteq) \to (\mathbb{D}_{\text{num}}^\#, \sqsubseteq) \]

- We derive an abstraction for sets of executions:
  \[ \alpha_{i,s} : \mathcal{P}(\mathcal{S}^\ast_P) \to \mathbb{L}_s \to \mathbb{D}_{\text{num}}^\# \]
  \[ X \to (l \in \mathbb{L}_s) \mapsto \alpha_{\text{num}}(\{m \mid \langle \ldots, (l, m), \ldots \rangle \in X\}) \]

- We assume also a source code static analysis, that computes a sound over-approximation of the behaviors of the program:
  \[ \alpha_{i,s}(\llbracket P_s \rrbracket) \sqsubseteq \llbracket P_s \rrbracket_i^\# \]
Abstract invariant translation

Two abstractions have been defined:
- Abstraction for static analysis of $P_s$
- Abstraction for defining compilation correctness

$$\left[ P_s \right]_i^\# \quad \alpha_s(\left[ P_s \right]) \equiv_{\pi_t} \alpha_c(\left[ P_c \right])$$

Those abstractions are in general not comparable
Abstract invariant translation

We can derive another abstraction, more abstract than both $\alpha_s^r$ 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 $L'_s, X'_s$**:
  e.g., all dead variables, all unreachable control states...
  e.g., in case of loop unrolling, it should perform the same trace partitioning

Moreover, $\pi_I, \pi_X$ induce a **safe abstract invariant translation function**
$$\pi^\#: (L'_s \rightarrow D^{\#}_{\text{num}}) \rightarrow (L'_c \rightarrow D^{\#}_{\text{num}})$$

- for each pair of control points in correspondance in $\pi_I$
- 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** $\pi^{\#}$ to an abstract invariant $\llbracket P_s \rrbracket^i_j$ computed for $P_s$

2. **Result:** a candidate invariant $\pi^{\#}(\llbracket P_s \rrbracket^i_j)$ for $P_c$
Invariant translation: soundness

**Soundness lemma**

If:
- the compilation $P_s \rightarrow P_c$ is sound with respect to $\pi_l, \pi_x$
- the analysis of $P_s$ computes a sound $\llbracket P_s \rrbracket_i^{\#} \alpha_{i,s}(\llbracket P_s \rrbracket_i) \sqsubseteq \llbracket P_s \rrbracket_i^{\#}$

Then, $\pi^{\#}((\alpha'_s)^{(\llbracket P_s \rrbracket_i^{\#}))}$ is a sound approximation of $\llbracket P_c \rrbracket$:

$$\alpha_{i,r,c}(\llbracket P_c \rrbracket) \sqsubseteq \pi^{\#}((\alpha'_s)^{(\llbracket P_s \rrbracket_i^{\#}))}$$

**Consequence of the choice of another observational semantics** for compilation correctness:
If $\alpha'_s(\llbracket P_s \rrbracket)$, $\alpha'_c(\llbracket P_c \rrbracket)$ 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 $\gamma$,

- The *concrete* function $F$ is montone,
- $F \circ \gamma \subseteq \gamma \circ F^\#$,
- $F^\#(x) \sqsubseteq x$.

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

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 $\gamma$,

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

Then, $\text{Ifp } F \subseteq \gamma(x)$

**Invariant checking refines abstract predicates:**

this phase also produces more precise abstract properties about:

- *memory locations* in $X_c \setminus X_c'$
- *program points* in $L_c \setminus L_c'$

**In practice, every cycle of the compiled code control flow graph should contain an element of $X_s$**
Invariant checking and difficulties

We consider the verification of invariants around a condition test

**Assumptions:**
- \( x \in [0, 12] \) 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 \leq 5) {
    assert(x \leq 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: \( x \in [0, 12] \)
   \textbf{li} \( r_0, x \)

4:
   \textbf{li} \( r_1, 5 \)

8:
   \textbf{cmp} \( r_0, r_1 \)

12:
   \textbf{blt}(GT) \( \ell \) \# (jump point)

16:
Invariant checking and difficulties

0: \( x \in [0, 12] \)
  \text{ld} r_0, x

4: \( x \in [0, 12] \land r_0 \in [0, 12] \)
  \text{li} r_1, 5

8:
  \text{cmp} r_0, r_1

12:
  \text{blt}(GT) \quad \# \text{(jump point)}

16:
Invariant checking and difficulties

0: \( x \in [0, 12] \)
   \textbf{ld} r_0, x

4: \( x \in [0, 12] \land r_0 \in [0, 12] \)
   \textbf{li} r_1, 5

8: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \)
   \textbf{cmp} r_0, r_1

12: \textbf{blt}(\text{GT}) / \quad \# \text{ (jump point)}
Invariant checking and difficulties

0: \( x \in [0, 12] \)

\( \text{ld } r_0, x \)

4: \( x \in [0, 12] \land r_0 \in [0, 12] \)

\( \text{li } r_1, 5 \)

8: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \)

\( \text{cmp } r_0, r_1 \)

12: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land cr \in \{\text{LT, EQ, GT}\} \)

\( \text{blt}(\text{GT}) \) \# (jump point)

16:
Invariant checking and difficulties

0: \( x \in [0, 12] \)

\texttt{ld} \( r_0, x \)

4: \( x \in [0, 12] \land r_0 \in [0, 12] \)

\texttt{li} \( r_1, 5 \)

8: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \)

\texttt{cmp} \( r_0, r_1 \)

12: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land cr \in \{\text{LT}, \text{EQ}, \text{GT}\} \)

\texttt{blt\langle GT\rangle l} \quad \# \text{(jump point)}

16: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land cr \in \{\text{LT}, \text{EQ}\} \)
Invariant checking and difficulties

0: \( x \in [0, 12] \)

\textbf{ld} \( r_0, x \)

4: \( x \in [0, 12] \land r_0 \in [0, 12] \)

\textbf{li} \( r_1, 5 \)

8: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \)

\textbf{cmp} \( r_0, r_1 \)

12: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land cr \in \{\text{LT, EQ, GT}\} \)

\textbf{blt}<\text{GT}> / \# \text{ (jump point)}

16: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land cr \in \{\text{LT, EQ}\} \)

The condition at the branch point is not precise

The range of \( x \) was not refined by the test:

- the test and branching are \textbf{independent}
  relations between test results and values need be tracked

- the test is made on \textbf{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 \(\{\text{LT}, \text{EQ}, \text{GT}\}\) should be bound to the ranges of the other location
- this is obtained by a value partitioning, based on the value of \(\text{cr}\):

\[
\gamma : (\{\text{LT}, \text{EQ}, \text{GT}\} \rightarrow \mathbb{D}^\#_{\text{num}}) \rightarrow \mathcal{P}(\mathbb{M})
\]

\[
\phi^\# \quad \mapsto \quad \{ m \mid m \in \gamma_{\text{num}} \circ \phi^\# \circ m(\text{cr}) \}
\]

Equalities between copies, e.g., of \(x\) and \(r_0\):

- an equality abstraction abstracts partitions of \(X_c\)
- replacement of \(\mathbb{D}^\#_{\text{num}}\) with a reduced product of \(\mathbb{D}^\#_{\text{num}}\) and an equality abstraction
Invariant checking: fixed

0: \[ x \in [0, 12] \]
\ld r_0, x

4:
\li r_1, 5

8:
\cmp r_0, r_1

12:
\blt\langle GT\rangle \ell \quad \# \text{ (jump point)}

16:

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

0: \( x \in [0, 12] \)
\( \text{ld} \ r_0, x \)

4: \( x \in [0, 12] \land r_0 \in [0, 12] \land x = r_0 \)
\( \text{li} \ r_1, 5 \)

8:
\( \text{cmp} \ r_0, r_1 \)

12:

16:

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

0: \( x \in [0, 12] \)
\text{ld} r_0, x

4: \( x \in [0, 12] \land r_0 \in [0, 12] \land x = r_0 \)
\text{li} r_1, 5

8: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land x = r_0 \)
\text{cmp} r_0, r_1

12:

\text{blt}<\text{GT}> L \quad \# \text{(jump point)}

16:

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

0: \( x \in [0, 12] \)
\( \text{ld } r_0, x \)

4: \( x \in [0, 12] \land r_0 \in [0, 12] \land x = r_0 \)
\( \text{li } r_1, 5 \)

8: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land x = r_0 \)
\( \text{cmp } r_0, r_1 \)
\[
\begin{align*}
\text{cr} &= \text{LT} \quad \Rightarrow \quad x \in [0, 4] \land r_0 \in [0, 4] \land x = r_0 \land r_1 \in [5, 5] \\
\text{cr} &= \text{EQ} \quad \Rightarrow \quad x \in [5, 5] \land r_0 \in [5, 5] \land x = r_0 \land r_1 \in [5, 5] \\
\text{cr} &= \text{GT} \quad \Rightarrow \quad x \in [6, 12] \land r_0 \in [6, 12] \land x = r_0 \land r_1 \in [5, 5]
\end{align*}
\]

12: \( \text{blt} \langle \text{GT} \rangle l \quad \# \text{ (jump point)} \)

16: 

**In general, invariant checking is incomplete...**

**It may require some refinement in the verifier**
Invariant checking: fixed

0: \( x \in [0, 12] \)
    ld \( r_0, x \)

4: \( x \in [0, 12] \land r_0 \in [0, 12] \land x = r_0 \)
    li \( r_1, 5 \)

8: \( x \in [0, 12] \land r_0 \in [0, 12] \land r_1 \in [5, 5] \land x = r_0 \)
    cmp \( r_0, r_1 \)
        \[
        \begin{cases}
            cr = \text{LT} \implies x \in [0, 4] \land r_0 \in [0, 4] \land x = r_0 \land r_1 \in [5, 5] \\
            cr = \text{EQ} \implies x \in [5, 5] \land r_0 \in [5, 5] \land x = r_0 \land r_1 \in [5, 5] \\
            cr = \text{GT} \implies x \in [6, 12] \land r_0 \in [6, 12] \land x = r_0 \land r_1 \in [5, 5] 
        \end{cases}
        \]

12: \[
\begin{cases}
    \text{cr} = \text{LT} \implies x \in [0, 4] \land r_0 \in [0, 4] \land x = r_0 \land r_1 \in [5, 5] \\
    \text{cr} = \text{EQ} \implies x \in [5, 5] \land r_0 \in [5, 5] \land x = r_0 \land r_1 \in [5, 5] \\
    \text{cr} = \text{GT} \implies x \in [6, 12] \land r_0 \in [6, 12] \land x = r_0 \land r_1 \in [5, 5] 
\end{cases}
\]
    blt\( \langle \text{GT} \rangle \ l \) \# (jump point)

16: \[
\begin{cases}
    \text{cr} = \text{LT} \implies x \in [0, 4] \land r_0 \in [0, 4] \land x = r_0 \land r_1 \in [5, 5] \\
    \text{cr} = \text{EQ} \implies x \in [5, 5] \land r_0 \in [5, 5] \land x = r_0 \land r_1 \in [5, 5] \\
    \text{cr} = \text{EQ} \implies \bot 
\end{cases}
\]

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

**Principle:** verify the semantic equivalence between source and compiled programs

**Verification process:** translation validation

1. Establish mappings $\pi_l, \pi_x$ between source and compiled programs
2. Prove (with a specialized prover) the semantic equivalence of each basic block

**Process:**

$$\alpha^s_s([P_s]) \xrightarrow{\pi_t} \alpha^c_c([P_c])$$

observation

$\alpha^s_s([P_s])$

$\alpha^c_c([P_c])$

observation

semantics

$[P_s]$

$[P_c]$

compilation

$P_s$

$P_c$
A technique based on fixpoint transfer

Foundation: fixpoint transfer

Theorem

Let $F_s : \mathcal{P}(S_s^*) \rightarrow \mathcal{P}(S_s^*)$ and $F_c : \mathcal{P}(S_c^*) \rightarrow \mathcal{P}(S_c^*)$ and $\pi_t : S_s^* \rightarrow S_c^*$ (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:

$$\text{lfp } F_c = \pi_t(\text{lfp } F_s)$$

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:**

- **Source code trace** and **compiled code trace**.

- **Standard semantics** $\llbracket P_s \rrbracket$ and $\llbracket P_c \rrbracket$ are expressed as least fixpoints, but not directly correlated by $\pi_x, \pi_I$.

- **Observational semantics** $\alpha_s^r(\llbracket P_s \rrbracket)$ and $\alpha_c^r(\llbracket P_c \rrbracket)$ are directly correlated by not expressed as least fixpoint.

We need fixpoint definitions for $\alpha_s^r(\llbracket P_s \rrbracket), \alpha_c^r(\llbracket P_c \rrbracket)$ (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:

\[ \delta(\in T) ::= \Box \quad \text{no transition (dead branch, error)} \]
\[ \quad \mid [x \leftarrow e] \quad \text{parallel assignment} \]
\[ \quad \mid [c \ ? \ \delta_0 \ | \ \delta_1] \quad \text{conditionnal} \]

Intuitively, a symbolic transfer function is a **store transformer**
Symbolic transfer functions: semantics

Semantic domain:
- \( \perp \) corresponds to the absence of behavior (error, blocking)
- \([\delta]\) \( \in \mathbb{M} \rightarrow \mathbb{M} \cup \{\perp\}\)

Denotational Semantics:
- \([\square](\rho) = \perp\)
- \([x \leftarrow e](\rho) = \rho[\forall i, [x_i](\rho) \leftarrow [e_i](\rho)]\)
  if \(\forall i, [x_i](\rho) \neq \text{error}\) and \(\forall i, [e_i](\rho) \neq \text{error}\)
- \([x \leftarrow e](\rho) = \perp\) otherwise

\([e ? \delta_0 | \delta_1](\rho) = \begin{cases} 
[\delta_0](\rho) & \text{if } [e](\rho) = \text{true} \\
[\delta_1](\rho) & \text{if } [e](\rho) = \text{false} \\
\perp & \text{if } [e](\rho) = \text{error}
\end{cases}\)

Note: observe the identity is described by \(\iota = [\cdot \leftarrow \cdot]\) (parallel assignment, with empty support)
Symbolic transfer functions: example

Encoding of a few instructions:

- **“Addition”** \( l_0 : \text{addi } r_1, r_1, v; \ l_1 : \ldots : \)

  \[
  \delta_{l_0,l_1} = [r_0 \leftarrow r_1 + v]
  \]

- **“Comparison”** \( l_0 : \text{cmp } r_0, r_1; \ l_1 : \ldots : \)

  \[
  \delta_{l_0,l_1} = [r_0 < r_1 ? \ [cr \leftarrow \text{LT}] \\
  \ | \ [r_0 = r_1 ? \ [cr \leftarrow \text{EQ}] | [cr \leftarrow \text{GT}]])]
  \]

- **“Conditional branching”** \( l_0 : \text{blt} \langle \text{LT} \rangle \ l_1; \ l_2 : \ldots : \)

  \[
  \delta_{l_0,l_1} = [cr = \text{LT} ? \ i | \ ] \\
  \delta_{l_0,l_2} = [cr = \text{LT} ? \ ] \\
  \delta_{l_0,l_2} = [cr = \text{LT} ? \ ]
  \]
Symbolic transfer functions: example

Encoding of a few instructions:

- **“Load”** \( \ell_0 : \text{ldx } r_d, o, r_x; \ \ell_1 : \ldots : \)
  \[
  \delta_{\ell_0, \ell_1} = \lfloor r_d \leftarrow \mu(o + r_x) \rfloor
  \]

- **“Load”** \( \ell_0 : \text{ld } r_d, o; \ \ell_1 : \ldots : \)
  \[
  \delta_{\ell_0, \ell_1} = \lfloor r_d \leftarrow \mu(o) \rfloor
  \]

- **“Store”** \( \ell_0 : \text{stx } r_d, o, r_x; \ \ell_1 : \ldots : \)
  \[
  \delta_{\ell_0, \ell_1} = \lfloor \mu(o + r_x) \leftarrow r_d \rfloor
  \]

The encoding of the source semantics is straightforward
Theorem

We can define a fully syntactic composition operation \( \otimes : T \times T \to T \) such that:

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

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

- \( \Box \otimes \delta = \Box \)
- \( \delta \otimes \Box = \Box \)
- \( \delta \otimes [c ? \delta_0 | \delta_1] = [c ? \delta \otimes \delta_0 | \delta \otimes \delta_1] \)
- \( [x_0 \leftarrow e_0] \otimes [x_1 \leftarrow e_1] = \begin{cases} [x_0 \leftarrow e_0[x_1 \leftarrow e_1]] & \text{if } x_0 = x_1 \\ [x_0 \leftarrow e_0[x_1 \leftarrow e_1] \\ x_1 \leftarrow e_1] & \text{otherwise} \end{cases} \)

(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)
- \(\delta_0 = \begin{bmatrix} x & \leftarrow & y + 4 \\ y & \leftarrow & 3 \end{bmatrix}\)
- \(\delta_1 = [y \leftarrow z + 1]\)

Then:

\[
\delta_0 \otimes \delta_1 = \begin{bmatrix} x & \leftarrow & z + 5 \\ y & \leftarrow & 3 \end{bmatrix}
\]

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:
  $$(l, \rho) \rightarrow (l', \rho') \iff$$
  $$\exists l_0, \ldots, l_n \in L_c \setminus L'_c,$$
  $$\rho' = [\delta_{l_n, l'} \otimes \cdots \otimes \delta_{l_i, l_{i+1}} \otimes \delta_{l_{i-1}, l_i} \otimes \cdots \otimes \delta_{l_0}](\rho)$$

Symbolic semantic abstraction

- **Semantics:** $[P'_c] = \text{Ifp } F'_c$ where $F'_c$ is derived from $P'_c$
- **Soundness property:** $\alpha'_c([P_c]) = [P'_c] = \text{Ifp } F'_c$
  
  **Proof:** by induction on the length of the traces of $P'_c$
Translation validation: example (condition test)

Compiled code:

```plaintext
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
```

STF to l:

\[ \delta^c_l = [x < 5 ? \mu(x) \left| r_0 \rightarrow \mu(x), r_1 \rightarrow 5, cr \rightarrow LT, \ldots \right| \]  

STF in \( P' \):

\[ \delta^c_l = [x < 5 ? \mu(x) \left| [x = 5 ? \mu(x) \left| \right| \right. \]
Translation validation and optimization: instruction scheduling

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Optimized Code</th>
<th>Syntactic Mappings:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l^s_0 \ i := i + 1;$</td>
<td>$l^o_0 \ ld r_0, i$</td>
<td>$\pi_{X \times X}: (l^s_0, i) \mapsto (l^o_0, i)$</td>
</tr>
<tr>
<td>$l^s_1 \ x := x + t[i];$</td>
<td>$l^o_1 \ ld r_1, x$</td>
<td>$(l^s_0, x) \mapsto (l^o_0, x)$</td>
</tr>
<tr>
<td>$l^o_2 \ add r_0, r_0, 1$</td>
<td>$l^o_2 \ add r_0, r_1, r_2$</td>
<td>$(l^s_1, i) \mapsto (l^o_5, i)$</td>
</tr>
<tr>
<td>$\cdots$</td>
<td>$l^o_5 \ st r_0, i$</td>
<td>$(l^s_1, x) \mapsto (l^o_1, x)$</td>
</tr>
<tr>
<td>$l^o_6 \ st r_1, x$</td>
<td>$l^o_7 \ st r_1, x$</td>
<td>$(l^s_2, i) \mapsto (l^o_7, i)$</td>
</tr>
<tr>
<td>$\cdots$</td>
<td>$l^o_i \ ld x r_2, t, r_0$</td>
<td>$(l^s_2, x) \mapsto (l^o_7, x)$</td>
</tr>
<tr>
<td>$\cdots$</td>
<td>$l^o_i \ ld x r_2, t, r_0$</td>
<td>$\delta_{l^o_i, l^o_i} = [\mu(x) \leftarrow \mu(x) + \mu(t + \mu(i))]$</td>
</tr>
</tbody>
</table>

- **Source level transfer functions:**
  \[ \delta_{l^s_0, l^s_1} = [i \leftarrow i + 1] \quad \delta_{l^s_1, l^s_2} = [x \leftarrow x + t[i]] \]

- **Optimized level transfer functions (registered not displayed):**
  \[ \delta_{l^o_0, l^o_i} = [\mu(i) \leftarrow \mu(i) + 1] \quad \delta_{l^o_i, l^o_i} = [\mu(x) \leftarrow \mu(x) + \mu(t + \mu(i))] \]
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
Conclusion

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.  
Semantics

- **Program transformations:** P. Cousot and R. Cousot.
  Systematic design of program transformation frameworks by abstract interpretation.

- **Relation between types and static analysis:**
  P. Cousot,
  *Types as Abstract Interpretations.*

- **Symbolic transfer functions:**
  C. Colby and P. Lee.
  *Trace-based program analysis.*
  In *23rd POPL*, pages 195–207, St. Petersburg Beach, (Florida USA), 1996.
Conclusion

Bibliography: Certified Compilation

- **Proof Carrying Codes**: G. C. Necula.
  Proof-Carrying Code.

- **Typed Assembly languages**:
  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.
Bibliography: Certified Compilation

