Section 1. Concrete Semantics

We define a toy programming language, called WHILE, which has the following syntax:

$$
\begin{aligned}
S &\rightarrow x := E \mid S \,;\, S \mid \texttt{skip} \\
&\mid \texttt{if } E \texttt{ then } S \texttt{ else } S \\
&\mid \texttt{while } E \texttt{ do } S \\
E &\rightarrow x \mid \# \mid E \diamond E
\end{aligned}
$$

A basic statement in WHILE is either an assignment, a sequential composition of two statements (separated by semicolon), or a “skip” (no-op).
Control structures are branching conditional (if-then-else) and loops (while).
An expression is either a single variable name (x), a number literal (#), or it can be composed from two sub-expressions via a binary operator $\diamond$. We allow the following operators:

$$\diamond \in \{+, -, *, /, =, \neq, <, >, \leq, \geq\}$$

For simplicity of the presentation, assume that all expressions have type $\mathbb{Z}$, that is, they are signed integers. Comparison operators are supposed to return either false=0 or true=1.

A *concrete state* of a program is defined as a mapping from variables to their concrete integer values, written:

$$\sigma : \text{Var} \rightarrow \mathbb{Z}$$

Where Var is the set of all program variables. The set containing all stores of this form is denoted by Σ.

To define the semantics of statements in WHILE, we introduce the notation

$$[\![s]\!] : \Sigma \rightarrow \Sigma$$

That is, it maps a concrete state σ to a new concrete state $[\![s]\!]\sigma$, that is reached after executing the statement s on the state σ.

We now list the definition of $[\![\ ]\!]$ by enumerating all the possible cases for a statement s.

$[\![x := e]\!]\sigma = \sigma[x \mapsto [\![e]\!]\sigma]$	assignment to a single variable
$[\![s_1 \,;\, s_2]\!]\sigma = [\![s_2]\!]([\![s_1]\!]\sigma)$	sequential composition
$[\![\texttt{skip}]\!]\sigma = \sigma$	no-op
$[\![\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2]\!]\sigma =$ $[\![s_1]\!]\sigma$ if $[\![e]\!]\sigma = \text{true}$, $[\![s_2]\!]\sigma$ if $[\![e]\!]\sigma = \text{false}$.	conditional statements

Where $[\![e]\!] : \Sigma \rightarrow \mathbb{Z}$ is the semantic value of expressions, and is also defined by case analysis:

$[\![x]\!]\sigma = \sigma(x)$	for a single variable x
$[\![c]\!]\sigma = c$	for a numerical constant c
$[\![e_1 \diamond e_2]\!]\sigma = [\![e_1]\!]\sigma \diamond [\![e_1]\!]\sigma$	for a binary operation $\diamond$

Notice that we have forgone loop statements. The definition for loops is a bit more convoluted:

When $[\![e]\!]\sigma = \text{false}$, $[\![\texttt{while}\ e\ \texttt{do}\ s]\!]\sigma = \sigma$ (a statement that has no effect).

When $[\![e]\!]\sigma = \text{true}$, running `while e do s` has the same effect as `s ; while e do s`.

Therefore,

$$[\![\texttt{while}\ e\ \texttt{do}\ s]\!]\sigma = [\![s\ ;\ \texttt{while}\ e\ \texttt{do}\ s]\!]\sigma = [\![\texttt{while}\ e\ \texttt{do}\ s]\!]([\![s]\!]\sigma)$$

It can be seen that the semantic definition of "while" is given in terms of itself.
Let $w = [\![\texttt{while}\ e\ \texttt{do}\ s]\!]$, we can express it as the equality:

$$w(\sigma) = \begin{cases} \sigma & \text{if } [\![e]\!]\sigma = \text{false}, \\ w([\![s]\!]\sigma) & \text{if } [\![e]\!]\sigma = \text{true}. \end{cases} \tag{1}$$

For loops that always terminate, there can be only one possible solution for w. (This can be shown by induction on the number of iterations the loop takes.) However, for nonterminating executions, may be multiple or even infinitely many solutions.

As an example, suppose $e = (n \neq 0)$ and $s = (n := n - 1)$, that is, the statement is

$$\texttt{while}\ n \neq 0\ \texttt{do}\ n := n - 1$$

Clearly, for any nonnegative initial value of n, the loop terminates with $n = 0$. However, for negative values, the loop diverges, *i.e.* never terminates. We denote nontermination with $\perp$, meaning that the semantic function is undefined for these inputs.

$$w(\sigma) = \begin{cases} \sigma[n \mapsto 0] & \text{if } \sigma(n) \geq 0, \\ \perp & \text{if } \sigma(n) < 0. \end{cases}$$

Notice that this w satisfies equation (1). However, here is another definition that also satisfies it:

$$w^{o}(\sigma) = \begin{cases} \sigma[n \mapsto 0] & \text{if } \sigma(n) \geq 0, \\ \sigma[n \mapsto 42] & \text{if } \sigma(n) < 0. \end{cases}$$

The ambiguity stems from the fact that for states satisfying $\sigma(n) < 0$, it is also true that $([\![s]\!]\sigma)(n) < 0$. and in both cases, $[\![e]\!]\sigma = [\![e]\!]([\![s]\!]\sigma) = \text{true}$. Equation (1) alone is not sufficient to enforce the value to be $\perp$; all it requires is of the values of w to be the same on the two states — which clearly holds for both $\perp$ and $\sigma[n \mapsto 42]$.

The solution is to define the "true" semantics of while-loops as the *least* solution that satisfies (1). In this respect, $\perp \sqsubseteq \sigma$ for any state σ, and we say therefore that $w \sqsubseteq w^{o}$. This kind of semantics is called *least fixed point semantics*.

Section 2. Galois Connection

As a prelude to defining semantics for abstract interpretation, we define the notion of a relationship between lattices known as *Galois connection*.

Definition. Let C, A be two lattices and let $\alpha : C \to A$ and $\gamma : A \to C$ be monotone functions between them. Then, α, γ form a Galois connection when the following holds:

$$\forall c \in C, a \in A. \quad \alpha(c) \sqsubseteq a \Leftrightarrow c \sqsubseteq \gamma(a) \tag{2}$$

Or, equivalently (since (2) is somewhat hard to grasp intuitively),

$$\forall c \in C, a \in A. \quad \alpha(\gamma(a)) \sqsubseteq a \ \wedge \ c \sqsubseteq \gamma(\alpha(c)) \tag{3}$$

Which states, that applying γ and then α, can only push the argument lower (in A), where as applying α followed by γ can only lift it higher (in C).

It can be shown easily that (2) implies (3):

$\gamma(a) \sqsubseteq \gamma(a) \Rightarrow \alpha(\gamma(a)) \sqsubseteq a$ (from reflexivity of $\sqsubseteq$ and from (2), going right to left, with $c = \gamma(a)$)

$\alpha(c) \sqsubseteq \alpha(c) \Rightarrow c \sqsubseteq \gamma(\alpha(c))$ (from reflexivity of $\sqsubseteq$ and from (2), going left to right, with $a = \alpha(c)$)

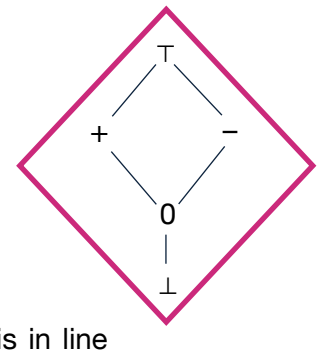
The converse, that (3) implies (2), can also be derived based on the monotonicity of α and γ:

$\alpha(c) \sqsubseteq a \Rightarrow c \sqsubseteq \gamma(\alpha(c)) \sqsubseteq \gamma(a) \Rightarrow c \sqsubseteq \gamma(a)$ (recall that $\sqsubseteq$ is transitive)

$c \sqsubseteq \gamma(a) \Rightarrow \alpha(c) \sqsubseteq \alpha(\gamma(a)) \sqsubseteq a \Rightarrow \alpha(c) \sqsubseteq a$

In our context, we assume from now on that C is the power-set lattice $\langle \mathscr{P}(\Sigma), \subseteq \rangle$, that is, elements of C are **sets** of concrete states, and they are ordered by set inclusion. We refer to C as the *concrete domain*, and to A as the *abstract domain*. Furthermore, α is our *abstraction function*, and γ is our *concretization function*.

As a running example, consider a lattice A comprising of the elements $\{\bot, 0, +, -, \top\}$ ordered according to the following Hasse diagram:



We can define a Galois connection as follows:

<u>Concretization</u>

$$\begin{aligned} &\gamma(\bot) = \varnothing, \gamma(\top) = \mathbb{Z}, \gamma(0) = \{0\} \\ &\gamma(+) = \{0, 1, 2, \ldots\} = \{v \in \mathbb{Z} \mid v \geq 0\} \\ &\gamma(-) = \{0, -1, -2, \ldots\} = \{v \in \mathbb{Z} \mid v \leq 0\} \end{aligned} \tag{4}$$

Notice that in our concretization, 0 is included in both $\gamma(+)$ and $\gamma(-)$. This is in line with the ordering of the abstract domain: $0 \sqsubseteq (+)$ and also $0 \sqsubseteq (-)$.

<u>Abstraction</u>

We define an auxiliary function $\beta : \mathbb{Z} \to A$ that "abstract" single values.

$$\beta(v) = \begin{cases} 0 & v = 0 \\ + & v > 0 \\ - & v < 0 \end{cases}$$

Then the abstraction function is defined via: $\alpha(S) = \sqcup\{\beta(\sigma) \mid \sigma \in S\}$

Note: the abstraction function can be defined in this manner for any abstract domain. Moreover, once β has been defined, the concretization function is determined by

$$\gamma(a) = \{\sigma \in \Sigma \mid \beta(\sigma) \sqsubseteq a\}$$

In the above case, where $\Sigma = \mathbb{Z}$, this definition coincides with γ as defined by (4).

We can show that α, γ that we just defined for $C = \mathscr{P}(\mathbb{Z})$ and $A = \{\bot, 0, +, -, \top\}$ indeed form a Galois connection:

E.g. for $a = (+)$,

$$\alpha(c) \sqsubseteq (+) \;\Leftrightarrow\; \sqcup\{\beta(\sigma) \mid \sigma \in c\} \sqsubseteq (+) \;\Leftrightarrow\; \forall\sigma \in c, \underbrace{\beta(\sigma)}_{\in \{0,+\}} \sqsubseteq (+) \;\Leftrightarrow\; \forall\sigma \in c, \sigma \geq 0 \;\Leftrightarrow\; c \sqsubseteq \underbrace{\{0,1,2,\ldots\}}_{= \gamma(+)}$$

The steps are then repeated for $a = (-)$, $a = (0)$.

For $a = (\bot)$, and $a = (\top)$, the proof is always trivial: $\alpha(c) \sqsubseteq \top$ and $c \sqsubseteq \gamma(\top)$ are both true statements unconditionally, and $\alpha(c) \sqsubseteq \bot$ and $c \sqsubseteq \gamma(\bot)$ are both true iff $c = \varnothing$.

Section 3. Abstract Semantics

Now that we have concrete semantics $[\![s]\!]$ and a connection between concrete sets of stores C and abstract states A, it is time to define the abstract interpreter.

We go back to our statements in the language WHILE, and introduce the notation:

$$[\![s]\!]^{\#} : A \longrightarrow A$$

The symbol $\#$ (pronounced: "sharp") signifies that this new semantics operates on abstract states. The expression $[\![s]\!]^{\#}\sigma^{\#}$ signifies the **abstract** state obtained from executing the statement s on an initial, also **abstract**, state $\sigma^{\#}$.

Intuitively, an abstract state $\sigma^{\#}$ will serve as a compact representation of a set of concrete states given by $\gamma(\sigma^{\#})$. For example, in a WHILE program with k variables $v_1, v_2, \ldots, v_k$, concrete stores can be represented as $\sigma \in \mathbb{Z}^k$, and abstract states can be represented as e.g.,

$$\sigma^{\#} \in A = \{\bot, 0, +, -, \top\}^k$$

Of course, the choice of abstract domain depends greatly on the properties that we wish to prove of our programs. Using the domain above, where each variable is associated with an element of $\{\bot, 0, +, -, \top\}$, would be suitable for proving properties about the sign (positive, negative) of the values stored in program variables.

As in the case of concrete semantics, the abstract semantics is defined by case analysis on the structure of the statements.

$[\![x := e]\!]^{\#}\sigma = \sigma[x \mapsto [\![e]\!]^{\#}\sigma]$	assignment to a single variable
$[\![s_1 ; s_2]\!]^{\#}\sigma^{\#} = [\![s_2]\!]^{\#}([\![s_1]\!]^{\#}\sigma^{\#})$	sequential composition
$[\![\texttt{skip}]\!]\sigma^{\#} = \sigma^{\#}$	no-op

Where we also introduced a new notation $[\![e]\!]^{\#} : A \rightarrow \{\bot, 0, +, -, \top\}$ for the abstract semantics of expressions:

$[\![x]\!]^{\#}\sigma^{\#} = \sigma^{\#}(x)$	for a single variable x
$[\![c]\!]^{\#}\sigma = \beta(c)$	for a numerical constant c
$[\![e_1 \diamond e_2]\!]^{\#}\sigma = [\![e_1]\!]^{\#}\sigma \,\bar{\diamond}\, [\![e_1]\!]^{\#}\sigma$	for a binary operation $\diamond$

Notice the "hat" on top of the operator $\bar{\diamond}$, telling us that this is not the "usual" arithmetic operation operating on numbers, but rather an abstract operation whose operands are from the abstract value domain, $\{\bot, 0, +, -, \top\}$.

As an example, consider integer multiplication, '$\cdot$'. The abstract variant $\overline{\cdot}$ is given by the following "multiplication" table:

$\overline{\cdot}$	$\bot$	0	+	−	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	0	0	0
+	$\bot$	0	+	−	$\top$
−	$\bot$	0	−	+	$\top$
$\top$	$\bot$	0	$\top$	$\top$	$\top$

The row and column pertaining to $\bot$ are somewhat immaterial, as they will always contain $\bot$ as a result (if one of the operands of an expression is undefined, than the result is undefined). Of the other entries, 0 multiplied by whatever value (even $\top$) gives 0, and the sign of the product can be deduced by the signs of the factors if they are both known (equal sign = positive, unequal signs = negative).

Similar tables can be constructed for other operators as well. In some cases, the result will have to be (conservatively) coerced to $\top$; for example, $(+) \overline{+} (+) = (+)$, but $(+) \overline{+} (-) = \top$.

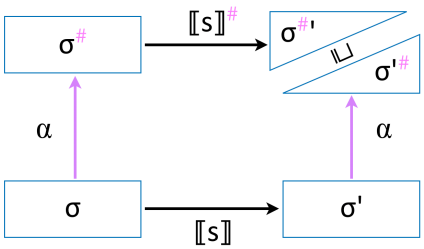
Abstract transformers: Soundness and Precision

How do we know that the abstract semantics defined via $[\![\ ]\!]^\#$ is "correct", in that it respects the abstraction defined by the Galois connection (α, γ)?

For any possible statement s, we require that the following inequality hold:

$$\alpha([\![s]\!](\sigma)) \sqsubseteq [\![s]\!]^\#(\alpha(\sigma))$$

Meaning that the abstract state obtained from the abstract transformer $[\![s]\!]^\#$ (when applied to an abstract state $\alpha(\sigma)$), must *over-approximate* — intuitively, "cover" more possible states — than what would be obtained from applying the concrete transformer $[\![s]\!]$ (to a concrete state σ), and abstracting the result.



An example of a transformer that **violates** this property is:

$$[\![y := y + 1]\!]^\#\sigma^\# = \sigma^\#[y \mapsto (+)]$$

Because, while certainly for states where $y \geq -1$ it would hold that y is non-negative ($\in \gamma(+)$), in all other situations, y would be negative. Formally, e.g. for $\sigma = \langle y \mapsto -2 \rangle$:

$$\alpha([\![y := y + 1]\!](\sigma)) = \langle y \mapsto (-) \rangle \not\sqsubseteq \langle y \mapsto (+) \rangle = [\![y := y + 1]\!]^\#(\alpha(\sigma))$$

In such a case we say that the abstract transformer is not sound.
The same transformer can be made sound quite easily in the following way:

$$[\![y := y + 1]\!]^\#\sigma^\# = \sigma^\#[y \mapsto \top]$$

Since $a \sqsubseteq \top$ holds for any $a \in \{\bot, 0, -, +, \top\}$. However, for certain abstract values of y, namely 0 and +, we know that mapping $y \mapsto (+)$ would also be sound, and is *strictly smaller* ($+ \sqsubset \top$, meaning that

$+ \sqsubseteq \top$ but $+ \neq \top$). In this case the over-approximation made by the transformer was too coarse, and we say that the transformer is not precise.

A sound and precise transformer for y := y + 1 can be given directly by:

$$[\![y := y + 1]\!]^{\#}\sigma^{\#} = \begin{cases} \sigma^{\#}[y \mapsto (+)] & \text{if } \sigma^{\#}(y) \in \{0,+\} \\ \sigma^{\#}[y \mapsto \top] & \text{otherwise} \end{cases}$$

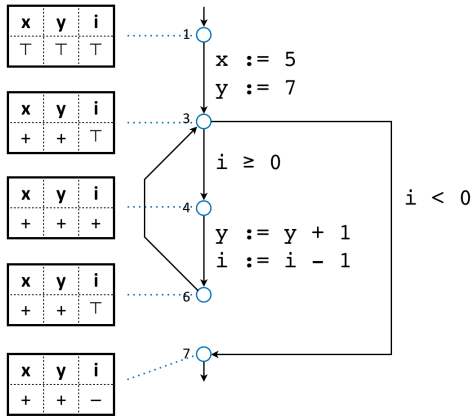
The best abstract transformer always exists and is given by the formula

$$[\![s]\!]^{\#}\sigma^{\#} = \alpha\big(\{[\![s]\!]\sigma \mid \sigma \in \gamma(\sigma^{\#})\}\big)$$

That is, by applying the concrete semantics of s to each concrete state in $\gamma(\sigma^{\#})$ and then abstracting the result. Sadly, the latter definition is of little practical use, since the set $\gamma(\sigma^{\#})$ is usually infinite. In general, while the best abstract transformer always exists, it is not always a *computable* function. (Note: obviously, it is always computable if the abstract domain is finite.)

Example. We have the following WHILE program:

```
1: x := 5;
2: y := 7;
3: while (i ≥ 0) do (
4:   y := y + 1;
5:   i := i - 1
6: )
7: assert 0 ≤ x + y
```



Annotated with an assertion at line 7.

(Notice that the assertion itself is not part of the WHILE language; it is used as a specification for the sake of static analysis).

A common problem in software verification is to make sure (at compile time) that all the assertions in the input program are satisfied in all possible executions of the program.

To verify a property, we construct the control-flow graph (CFG) of the given program. It is easy to think of the vertices of the CFG as representing *program locations*, and of the edges as representing *transitions*, labeled by a basic block whose semantic must be followed when the program is executed and moves from one state to the next. For example, in order for control to flow from location 1 (beginning of program) to location 3 (just before entering the **while** loop), the values of variables x and y must be updated to 5 and 7, respectively.

Similarly to data-flow analysis (lecture 8 of this course), we will define transfer functions for CFG edges. We will use the abstract transformers that were just defined above: for an edge $u \to v$ labeled by a statement s, we associate the transfer function $[\![s]\!]^{\#}$. Remember that these functions operate on *abstract* states (elements of A). We then proceed with the same fixed-point algorithm; if and when the analysis converges, the result is one abstract state $\sigma^{\#}(\ell)$ per program location ℓ. It is then **guaranteed** that in any execution, when the program reaches the location ℓ, its state must be in $\gamma(\sigma^{\#}(\ell))$.

In the diagram above, the abstract state at location 7 at the end of the analysis is $\sigma^{\#}(7) = \langle x \mapsto (+), y \mapsto (+), i \mapsto (-)\rangle$. Therefore, x and y must both be non-negative, in any execution of the program. As a consequence, the assertion at line 7 is always true; this can be verified by noting that:

$$[\![x + y]\!]^{\#}(\sigma^{\#}(7)) = (+)$$