Lecture 9
Abstract interpretation

Motivation

We reason about programs statically, but we are really trying to make predictions about their dynamic behaviour.

Why not examine this behaviour directly?

It isn’t generally feasible (e.g. termination, inputs) to run an *entire* computation at compile-time, but we can find things out about it by running a *simplified* version.

This is the basic idea of *abstract interpretation*.

Abstract interpretation

Warning: this will be a heavily simplified view of abstract interpretation; there is only time to give a brief introduction to the ideas, not explore them with depth or rigour.

Abstract interpretation

The key idea is to use an *abstraction*: a model of (otherwise unmanageable) reality, which

- discards enough detail that the model becomes manageable (e.g. small enough, computable enough), but
- retains enough detail to provide useful insight into the real world.

Abstract interpretation

For example, to plan a trip, you might use a map.

- A road map sacrifices a lot of detail —
  - trees, road conditions, individual buildings;
  - an entire dimension —
- but it retains most of the information which is important for planning a journey:
  - place names;
  - roads and how they are interconnected.

Abstract interpretation

Crucially, a road map is a useful abstraction because the route you plan is probably still valid back in reality.

- A cartographer creates an abstraction of reality (a map),
- you perform some computation on that abstraction (plan a route),
- and then you transfer the result of that computation back into the real world (drive to your destination).

Abstract interpretation

Trying to plan a journey by exploring the concrete world instead of the abstraction (i.e. driving around aimlessly) is either very expensive or virtually impossible.

A trustworthy map makes it possible — even easy.

This is a *real application* of abstract interpretation, but in this course we're more interested in computer programs.

Multiplying integers

A canonical example is the multiplication of integers.

If we want to know whether −1515 × 37 is positive or negative, there are two ways to find out:

- Compute in the concrete world (arithmetic), using the *standard interpretation* of multiplication. −1515 × 37 = −56055, which is negative.
- Compute in an abstract world, using an *abstract interpretation* of multiplication: call it ⊗.

Multiplying integers

In this example the magnitudes of the numbers are insignificant; we care only about their sign, so we can use this information to design our abstraction.

$$(-) = \{\, z \in \mathbb{Z} \mid z < 0 \,\}$$

$$(0) = \{\, 0 \,\}$$

$$(+) = \{\, z \in \mathbb{Z} \mid z > 0 \,\}$$

In the concrete world we have all the integers; in the abstract world we have only the values (−), (0) and (+).

Multiplying integers

We need to define the abstract operator ⊗.
Luckily, we have been to primary school.

⊗	(−)	(0)	(+)
(−)	(+)	(0)	(−)
(0)	(0)	(0)	(0)
(+)	(−)	(0)	(+)

Multiplying integers

Armed with our abstraction, we can now tackle the original problem.

$$abs(-1515) = (-)$$

$$abs(37) = (+)$$

$$(-) \otimes (+) = (-)$$

So, without doing any *concrete* computation, we have discovered that −1515 × 37 has a negative result.

Multiplying integers

This is just a toy example, but it demonstrates the methodology: state a problem, devise an abstraction that retains the characteristics of that problem, solve the problem in the abstract world, and then interpret the solution back in the concrete world.

This abstraction has avoided doing arithmetic; in compilers, we will mostly be interested in avoiding expensive computation, nontermination or undecidability.

Safety

As always, there are important safety issues.

Because an abstraction discards detail, a computation in the abstract world will necessarily produce less precise results than its concrete counterpart.

It is important to ensure that this imprecision is *safe*.

Safety

We consider a particular abstraction to be safe if, whenever a property is true in the abstract world, it must also be true in the concrete world.

Our multiplication example is actually quite precise, and therefore trivially safe: the magnitudes of the original integers are irrelevant, so when the abstraction says that the result of a multiplication will be negative, it definitely will be.

In general, however, abstractions will be more approximate than this.

Adding integers

A good example is the *addition* of integers.
How do we define the abstract operator ⊕?

⊕	(−)	(0)	(+)
(−)	(−)	(−)	(?)
(0)	(−)	(0)	(+)
(+)	(?)	(+)	(+)

Adding integers

When adding integers, their (relative) magnitudes *are* important in determining the sign of the result, but our abstraction has discarded this information.

As a result, we need a new abstract value: (?).

$$(-) = \{\, z \in \mathbb{Z} \mid z < 0 \,\}$$

$$(0) = \{\, 0 \,\}$$

$$(+) = \{\, z \in \mathbb{Z} \mid z > 0 \,\}$$

$$(?) = \mathbb{Z}$$

Adding integers

(?) is less precise than (–), (0) and (+); it means “I don’t know”, or “it could be anything”.

Because we want the abstraction to be safe, we must put up with this weakness.

Adding integers

$$\mathit{abs}(19 + 23) = \mathit{abs}(42) = (+)$$

$$\mathit{abs}(19) \oplus \mathit{abs}(23) = (+) \oplus (+) = (+)$$

$$\begin{tikzcd} 19, 23 \arrow[r, "+"] \arrow[d, "\mathit{abs}"'] & 42 \arrow[d, "\mathit{abs}"] \\ (+), (+) \arrow[r, "\oplus"'] & (+) \end{tikzcd}$$

Adding integers

$$\mathit{abs}(-1515 + 37) = \mathit{abs}(-1478) = (-)$$

$$\mathit{abs}(-1515) \oplus \mathit{abs}(37) = (-) \oplus (+) = (?)$$

$$\begin{tikzcd}
-1515, 37 \arrow[r, "+"] \arrow[d, "\mathit{abs}"'] & -1478 \arrow[d, "\mathit{abs}"] \\
(-), (+) \arrow[r, "\oplus"'] & (?)
\end{tikzcd}$$

Safety

Here, safety is represented by the fact that (–) ⊆ (?):

$$\{ z \in \mathbb{Z} \mid z < 0 \} \subseteq \mathbb{Z}$$

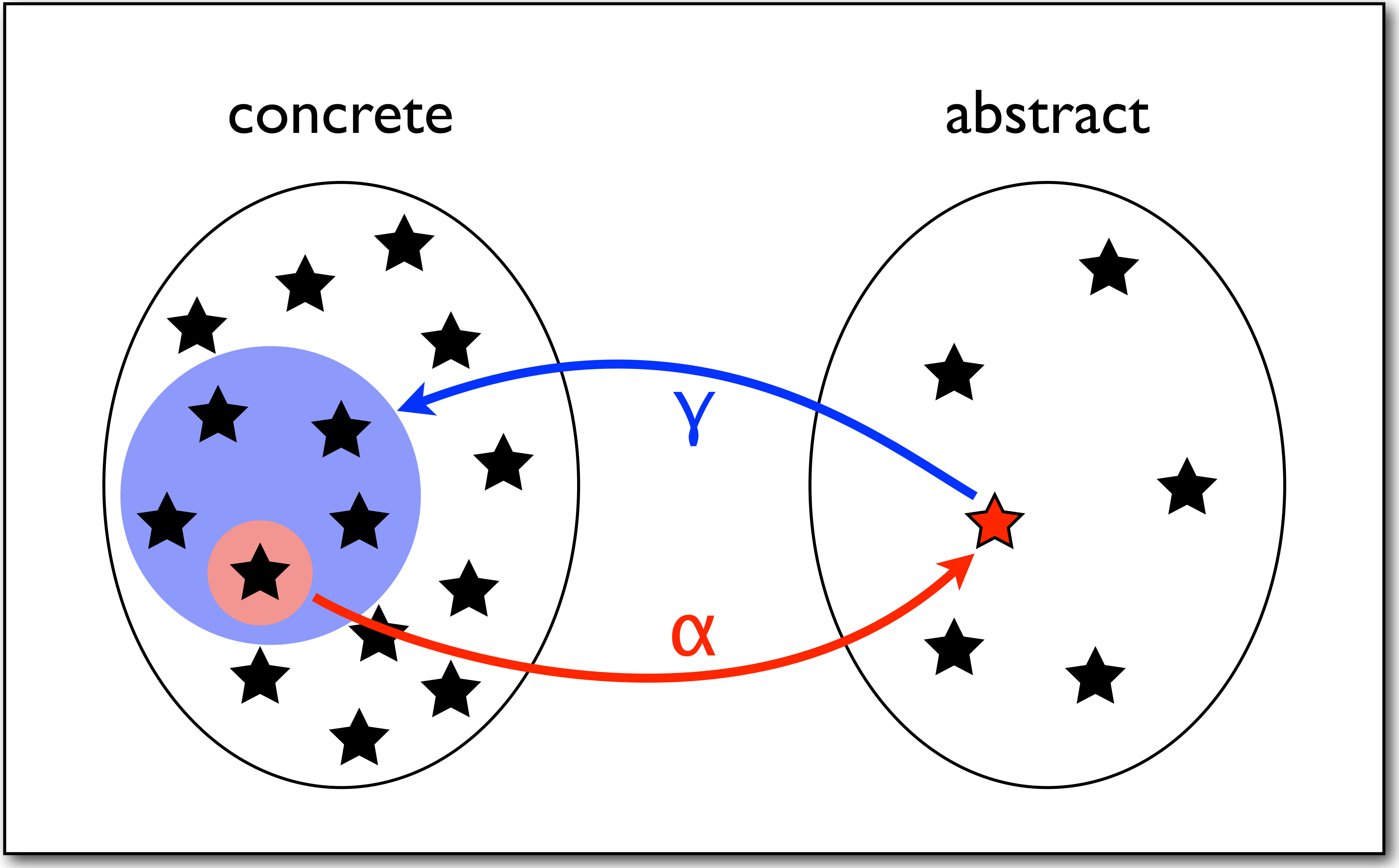
The result of doing the abstract computation is less precise, but crucially includes the result of doing the concrete computation (and then abstracting), so the abstraction is safe and hasn't missed anything.

Abstraction

Formally, an abstraction of some concrete domain D (e.g. $\wp(\mathbb{Z})$) consists of

- an abstract domain $D^{\#}$ (e.g. { (–), (0), (+), (?) }),
- an abstraction function $\alpha : D \to D^{\#}$ (e.g. *abs*), and
- a concretisation function $\gamma : D^{\#} \to D$, e.g.:
  - (–) $\mapsto$ $\{ z \in \mathbb{Z} \mid z < 0 \}$,
  - (0) $\mapsto$ $\{ 0 \}$, etc.

Abstraction



Abstraction

Given a function f from one concrete domain to another (e.g. $\hat{+} : \wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \rightarrow \wp(\mathbb{Z})$), we require an abstract function $f^{\#}$ (e.g. $\oplus$) between the corresponding abstract domains.

$$\hat{+}(\{\ 2, 5\ \}, \{\ -3, -7\ \}) = \{\ -1, -5, 2, -2\ \}$$

$$\oplus((+), (-)) = (?)$$

Abstraction

So, for $D = \wp(\mathbb{Z})$ and $D^\# = \{ (-), (0), (+), (?) \}$, we have:

$$\begin{tikzcd}
\mathbb{Z} \times \mathbb{Z} \arrow[r, "+"] \arrow[d] & \mathbb{Z} \arrow[d] \\
D \times D \arrow[r, "\hat{+}"] \arrow[d, shift right, "\alpha_1"'] & D \arrow[d, shift right, "\alpha_2"'] \\
D^\# \times D^\# \arrow[r, "\oplus"] \arrow[u, shift right, "\gamma_1"'] & D^\# \arrow[u, shift right, "\gamma_2"']
\end{tikzcd}$$

where $\alpha_{1,2}$ and $\gamma_{1,2}$ are the appropriate abstraction and concretisation functions.

Abstraction

These mathematical details are formally important, but are not examinable on this course.

Abstract interpretation can get very theoretical, but what's significant is the idea of using an abstraction to safely model reality.

Recognise that this is what we were doing in data-flow analysis: interpreting 3-address instructions as operations on *abstract values* — e.g. live variable sets — and then "executing" this abstract program.

Summary

- Abstractions are manageably simple models of unmanageably complex reality
- Abstract interpretation is a general technique for executing simplified versions of computations
- For example, the sign of an arithmetic result can be sometimes determined without doing any arithmetic
- Abstractions are approximate, but must be safe
- Data-flow analysis is a form of abstract interpretation