Contents

Chapter 1

Syntax Driven Abstract Interpretation

In this section, functions on parsetrees are converted into functions over sets of parsetrees. This is useful to algorithmically analyze these functions, which will help to gradualize them. The technique to convert these metafunctions, called **abstract interpretation** is dissected in this part, which is organized as following:

- First, we'll work out **what abstract interpretation is**, with simple examples followed by its desired properties.
- Then, we work out what **properties a syntax** has.
- With these, we develop an **efficient representation** to capture infinite sets of parsetrees.
- Afterwards, **operations on these setrepresentations** are stated.
- As last, the metafunctions are actually lifted to **metafunctions over sets**

1.1 Abstract interpretation

Per Rice's theorem, it is generally impossible to make precise statements about all programs. However, making useful statements about some programs is feasable. Cousot [1] introduces a suitable framework, named **abstract interpretation**: "A program denotes computations in some domain of objects. Abstract interpretation of programs consists in using that denotation to describe computations in *another domain of abstract objects*, so that the results of the abstract computations give some information on the actual computation".

This principle can best be illustrated, for which the he successor function (as defined in figure 1.1) is a prime example. The function normally operates in the domain of *integer numbers*, as `succ` applied on `1` yields `2`; `succ -1` yields `0`.

But `succ` might also be applied on *signs*: the symbols `+`, `-` or `0` are used instead of integers, where `+` represents all strictly positive numbers and `-` represents all strictly negative numbers. These symbols are used to perform the computation, giving rise to a computation in the abstract domain of signs.

```
succ n   = n + 1
```

Figure 1.1: Definition of the *successor* function, defined for natural numbers

1.1.1 The rule of signs

A positive number which is increased is always positive. Thus, per rule of signs `+` $+1$ is equal to `+`. The calculation of `succ +` thus yields `+`.

On the other hand, a negative number which is increased by one, might be negative, but might be zero too. `-` $+1$ thus results in both `0` and `-`. The result is that applying `succ` to a negative number gives less precise information.

The question now arises how to deal with less precise information. One option might be to fail and indicate that no information could be deduced at all. Another option is to introduce exra symbols representing the unions of `+`, `-` and `0`. The symbol representing the negative numbers (including zero) would be `0-`, analogously does `0+` represent the positive numbers (including zero). Both the strictly negative and strictly positive numbers are represented by `+-`. At last, the union of all numbers is represented with $\top$.

These symbols represent a set, where the set represented by `+` (the strictly positive numbers) are embedded in the set represented by `0+` (the positive numbers). This *embedding* relation forms a lattice, as each two symbols have a symbol embedding the union of both, as can be seen in 1.2.

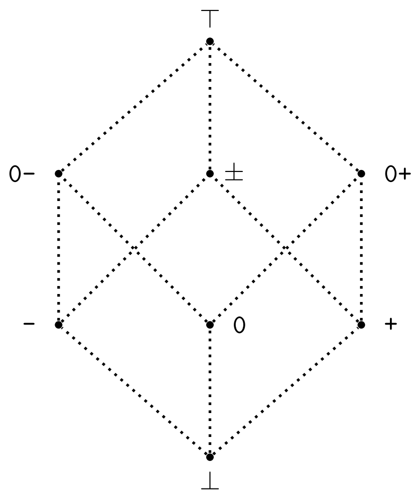

Figure 1.2: The symbols representing sets. Some sets, such as `+` are embedded in others, such as `0+` . These embeddings form a lattice.

Concretization and abstraction

The meaning of `succ -` *giving* `0-` is intuitionally clear: *the successor of a negative number is either negative or zero.* More formally, it can be stated that, *given a negative number,* `succ` *will give an element from* $\{n|n \leq 0\}$. The meaning of

`0-` is formalized by the **concretization** function γ, which translates from the abstract domain to the concrete domain:

$$\begin{array}{rcl}
\gamma(\texttt{ - }) & = & \{z | z \in \mathbb{Z} \wedge z < 0\} \\
\gamma(\texttt{ 0- }) & = & \{z | z \in \mathbb{Z} \wedge z \leq 0\} \\
\gamma(\texttt{ 0 }) & = & \{0\} \\
\gamma(\texttt{ 0+ }) & = & \{z | z \in \mathbb{Z} \wedge z \geq 0\} \\
\gamma(\texttt{ + }) & = & \{z | z \in \mathbb{Z} \wedge z > 0\} \\
\gamma(\texttt{ +- }) & = & \{z | z \in \mathbb{Z} \wedge z \neq 0\} \\
\gamma(\top) & = & \mathbb{Z}
\end{array}$$

On the other hand, an element from the concrete domain is mapped onto the abstract domain with the **abstraction** function. This functions *abstracts* a property of the concrete element:

$$\alpha(n) = \left\{ \begin{array}{cl} \texttt{-} & \text{if } n < 0 \\ \texttt{0} & \text{if } n = 0 \\ \texttt{+} & \text{if } n > 0 \end{array} \right\}$$

The abstraction function α is often used over sets as well. Abstraction of a set is equivalent to abstracting each element in the set, after which the union of all is taken:

$$\alpha N = \bigcup \{\alpha(n) | n \in N\}$$

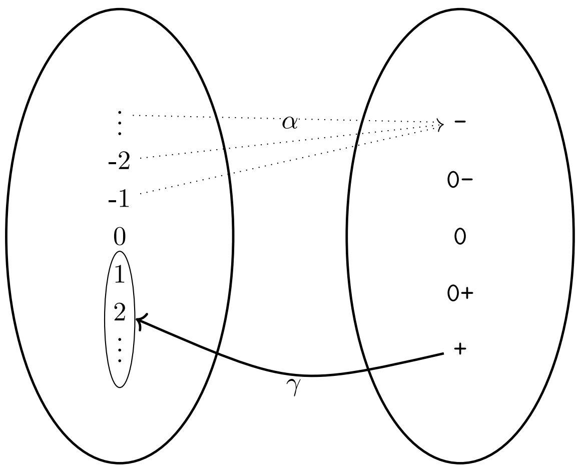

Figure 1.3: Concretization and abstraction between integers and signs

1.1.2 Ranges as abstract domain

Another possibility is to apply functions on an entire range at once (such as `[2,41]`), using abstract interpretation. Here, the abstract domain used are ranges of the form `[n, m]`.

This can be calculated by taking each element the range represents, applying the function on each element and abstracting the newly obtained set, thus

$$\alpha(map(f, \gamma(\texttt{ [n, m] })))$$

Where *map* applies f on each element of the set (like most functional programming languages), α is the abstraction function and γ is the concretization function, with following definition:

$$\begin{aligned} \alpha(n) &= \texttt{[n, n]} \\ \alpha(N) &= \texttt{[}\ min(N), max(N)\ \texttt{]} \\ \gamma(\ \texttt{[n, m]}\ ) &= \{x | n \leq x \leq m\} \end{aligned}$$

For example, the outputrange for `succ [2,41]` can be calculated in the following way:

$$\begin{aligned} & \alpha(map(\ \texttt{succ}\ , \gamma(\ \texttt{[2,41]}\ ))) \\ =\ & \alpha(map(\ \texttt{succ}\ , \{2, 3, ..., 40, 41\})) \\ =\ & \alpha(\{\ \texttt{succ}\ 2,\ \texttt{succ}\ 3, ...,\ \texttt{succ}\ 40,\ \texttt{succ}\ 41\}) \\ =\ & \alpha(\{3, 4, ..., 41, 42\}) \\ =\ & [3, 42] \end{aligned}$$

However, this is computationally expensive: aside from translating the set from and to the abstract domain, the function f has to be calculated $m - n$ times. By exploiting the underlying structure of ranges, calculating f has only to be done *two* times, aside from never having to leave the abstract domain.

The key insight is that addition of two ranges is equivalent to addition of the borders:

$$[n, m] + [x, x] = [n + x, m + x]$$

Thus, applying `succ` to a range can be calculated as following, giving the same result in an efficient way:

$$\begin{aligned} & \texttt{succ [2,5]} \\ =\ & \texttt{[2,5]}\ + \alpha(1) \\ =\ & \texttt{[2,5]}\ +\ \texttt{[1,1]} \\ =\ & \texttt{[3,6]} \end{aligned}$$

1.1.3 Collecting semantics

As last example, the abstract domain might be the *set* of possible values, such as `{1,2,41}`. Applying `succ` to this set will yield a new set:

$$\begin{aligned} & \texttt{succ}\ \{\texttt{1,2,41}\} \\ =\ & \{\texttt{1,2,41}\}\ + \alpha(1) \\ =\ & \{\texttt{1,2,41}\}\ +\ \{\texttt{1}\} \\ =\ & \{\texttt{2,3,42}\} \end{aligned}$$

Translation from and to the abstract domain are trivially implemented. After all, the abstraction of a concrete value is the set containing only the value itself, where the concretization of a set of values is exactly the set of these values. This results in the following straightforward definitions:

$$\begin{aligned} \alpha(n) &= \{n\} \\ \gamma(\{n1, n2, \ldots\} &= \{n1, n2, ...\} \end{aligned}$$

Performing the computation in the abstract domain of sets can be more efficient than the equivalent concrete computations, as the structure of the concrete domain can be exploited to use a more efficient representation in memory (such as ranges). Furthermore, different input might turn out to have the same result halfway in the calculation, such as `f x = abs(x) + 1` with input `{+1,-1}` which becomes `{1} + 1`. This state merging might result in additional speed increases.

Using this abstract domain effectively lifts a function over integers into a function over sets of integers. Exactly this abstract domain is used to lift the functions over parsetrees into functions over sets of parsetrees. To perform these calculations, an efficient representations of possible parsetrees will be deduced later in this section, in chapter **??**.

1.1.4 Properties of α and γ

For abstract interpretation framework to work, the functions α and γ should obey to the properties *monotonicity* and *correctness.* These properties guarantee the soundness of the approach. On top of that is a **Galois connection** between the concrete and abstract domains implied by these properties.

Monotonicity of α and γ

The first requirement is that both *abstraction* and *concretization* are monotone. This states that, if the set to concretize grows, the set of possible properties *might* grow, but never shrink.

Analogously, if the set of properties grows, the set of concrete values represented by these properties might grow too.

$$\begin{gathered} X \subseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y) \\ X \subseteq Y \Rightarrow \alpha(X) \subseteq \alpha(Y) \end{gathered}$$

This can be illustrated with the abstract domain of signs. Consider $X = 1, 2$ and $Y = 0, 1, 2$. This gives:

$$\begin{array}{rrcl} & X \subseteq Y & \Rightarrow & \gamma(X) \subseteq \gamma(Y) \\ = & \{1,2\} \subseteq \{0,1,2\} & \Rightarrow & \alpha(\{1,2\}) \subseteq \alpha(\{0,1,2\}) \\ = & \{1,2\} \subseteq \{0,1,2\} & \Rightarrow & \texttt{+} \subseteq \texttt{0+} \end{array}$$

Per definition is `+` a subset of `0+`, so this example holds.

Soundness

When a concrete value n is translated into the abstract domain, we expect that $\alpha(n)$ represents this value. An abstract object m represents a concrete value n iff its concretization contains this value:

$$\begin{gathered} n \in \gamma(\alpha(n)) \\ \text{or equivalent} \\ X \subseteq \alpha(Y) \Rightarrow Y \subseteq \gamma(X) \end{gathered}$$

Inversly, some of the concrete objects in $\gamma(m)$ should exhibit the abstract property m:

$$\begin{gathered} m \in \alpha(\gamma(m)) \\ \text{or equivalent} \\ Y \subseteq \gamma(X) \Rightarrow X \subseteq \alpha(Y) \end{gathered}$$

This guarantees the *soundness* of the approach. This propery guarantees that the abstract object obtained by an abstract computation, indicates what a concrete computation might yield.

Without these properties tying α and γ together, abstract interpretation would be meaningless: the abstract computation would not be able to make statements about the concrete computations. For example, working with the abstract domain of signs where α maps `0` onto `+` yields following results:

$$\alpha(n) = \left\{ \begin{array}{cl} \texttt{-} & \text{if } n < 0 \\ \texttt{+} & \text{if } n = 0 \\ \texttt{+} & \text{if } n > 0 \end{array} \right\}$$

$$\gamma(\ \texttt{+}\ ) = \{n | n > 0\}$$
$$\gamma(\ \texttt{-}\ ) = \{n | n < 0\}$$

This breaks soundness, as $\gamma(\alpha(0)) = \gamma(\ \texttt{+}\ ) = \{1, 2, 3, ...\}$, clearly not containing the original concrete element 0. With these definitions, the approach becomes faulty. For example, `x - 1` with $x =   `+` ‘ would become

$$\begin{array}{rl} & \alpha(\gamma(\ \texttt{+}\ ) - 1) \\ = & \alpha(\{n - 1 | n > 0\}) \\ = & \texttt{+} \end{array}$$

A blatant lie, of course; `0 - 1` is all but a positive number.

Galois connection

Together, α and γ form a *Galois connection*, as it obeys its central property:

$$\alpha(a) \subseteq b \Leftrightarrow a \subseteq \gamma(b)$$

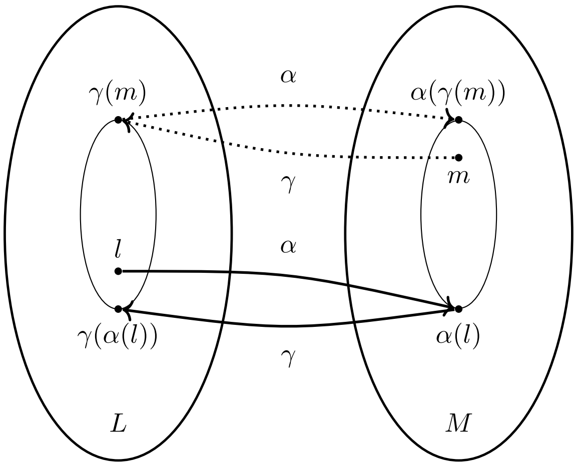

Figure 1.4: Galois-connection, visualized

Bibliography

[1] *Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approcimation of fixpoints*, ACM Symposium on Principles of Programming Languages (POPL), 1977.