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. This section dissects the technique to do so -abstract interpretation- and is organized as following:

- First, we'll work out **what abstract interpretation is**, with a simple example followed by its desired properties.
- Then, we work out what **properties a syntax** should have.
- With these, we develop an **efficient representation** to capture infinite sets of parsetrees.
- Afterwards, **operations on these setrepresentations** are stated.
- As last, we build useful **algorithms** and checks with this algebra.

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 (1977) introduces a framework to do so, named **abstract interpretation**: "A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of the abstract computations give some information on the actual computation".

We illustrate this with the successor function, as defined below:

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

The successor function operates in the domain of *integer numbers*: `succ` applied on `1` yields `2`; `succ -1` yields `0`.

But `succ` might also be applied on *signs*, instead of integers: we use `+`, `-` or `0` to perform the computation, giving rise to a computation in the abstract domain of signs.

1.1.1 The rule of signs

Per rule of signs `+` $+1$ is equal to `+`, thus `succ +` yields `+`. Applying `succ` to a negative number gives less precise information, as `-` $+1$ could yield both zero or a strictly negative number, giving `0-` in the abstract domain.

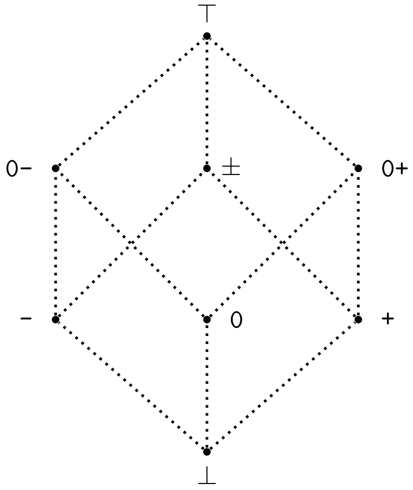

Figure 1.1: Composition of possibilities about signs

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:

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

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:

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

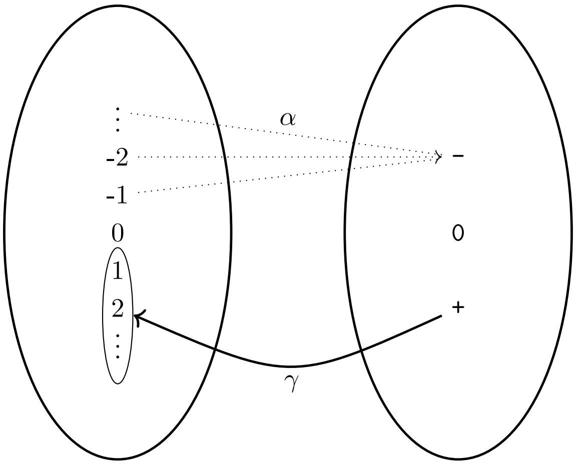

Figure 1.2: Concretization and abstraction between integers and signs

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. In order to do so, 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:

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

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

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

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:

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

1.1.3 Working with collecting semantics

At last, 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:

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

Translation from and to the abstract domain are straightforward:

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

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). 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.

1.1.4 Properties of α and γ

For abstract interpretation to work, the functions α and γ should obey to some properties to make this approach work, namely *monotonicity* and *correctness.* These properties imply a **Galois connection** between the concrete and abstract domains.

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.

$$X \subseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y)$$
$$X \subseteq Y \Rightarrow \alpha(X) \subseteq \alpha(Y)$$

When working with signs as properties, this property can be illustrated with:

$$\{1,2\} \subseteq \{0,1,2\} \Rightarrow \texttt{+} \subseteq \texttt{0+}$$

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:

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

This guarantees the *soundness* of our approach.

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:

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

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

What would `x - 1` give, where `x = +`? This is equivalent to

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

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

Galois connection

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

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

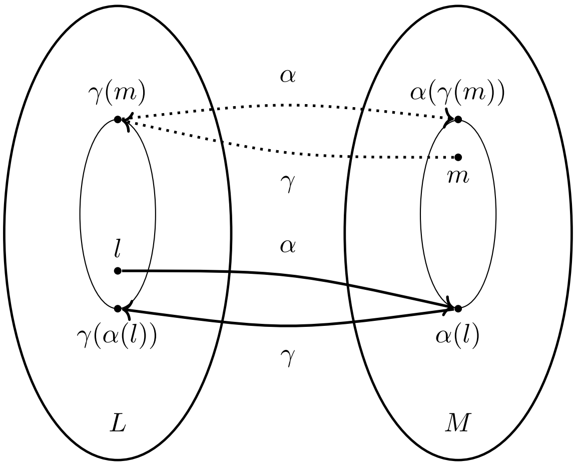

Figure 1.3: Galois-connection, visualized

1.2 Properties of the syntax

Abstract interpretation gives us a way to lift metafunctions over parsetrees to metafunctions over sets of parsetrees. To efficiently perform computations in this abstract domain, a representation for these sets should be constructed, exploiting the underlying structure of syntaxes. Here, we study the necessary properties which are exploited to construct a set representation in the next part.

Some of these properties are inherent to each syntax, others should be enforced. For these, we present the necessary algorithms to detect these inconsistencies, both to help the programmer and to allow abstract interpretation.

1.2.1 Syntactic forms as sets

When a syntactic form is declared, this is equivalent to defining a set.

The declaration of `bool ::= "True" | "False"` is equivalent to declaring *bool* = { `True` , `False` }.

$$\texttt{bool} \longleftrightarrow \{\text{True, False}\}$$

This equivalence between syntactic forms and sets is the main driver, both for the other properties and the efficient representation for sets we will use later on.

1.2.2 Embedded syntactic forms

Quite often, one syntactic form is defined in term of another syntactic form. This might be in a sequence (e.g. `int "+" int`) or as a bare choice (e.g. ... `| int |` ...). If the case of a bare choice, each element the choice is embedded into declared syntactic form.

In the following example, both `bool` and `int` are embedded into `expr`, visualized by ALGT in 1.4:

```
bool     ::= "True" | "False"
int      ::= Number        # Number is a builtin, parsing integers
expr     ::= bool | int
```

This effectively establishes a *supertype* relationship between the different syntactic forms. We can say that *every `bool` is a `expr`*, or `bool <: expr`.

This supertype relationship is a lattice - the absence of left recursion implies that no cycles can exist in this supertype relationship. This lattice can be visualized, as in figure 1.5.

1.2.3 Empty sets

The use of empty strings might lead to ambiguities of the syntax. When an emtpy string can be parsed, it is unclear wether this should be included in the parsetree. Therefore, it is not allowed.

As example, consider following syntax:

```
a        ::= "=" | ""
b        ::= "x" a "y" | "x" "y"
c        ::= a as
```

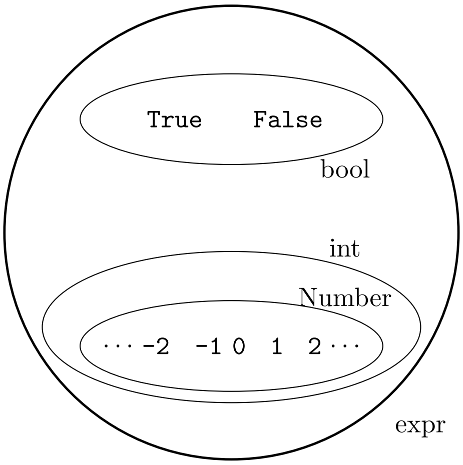

Figure 1.4: Nested syntactic forms

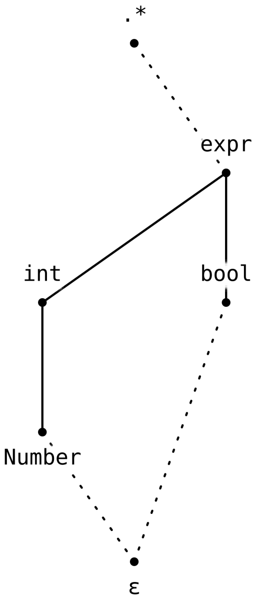

Figure 1.5: A simple subtyping relationship

Parsing `b` over string `x y` is ambiguous: the parser might return a parstree with or without an empty token representing `a`. Parsing `c` is even more troublesome, here hte parser might return an infinite list containing only empty `a`-elements.

Empty syntactic forms can cause the same ambiguities, and are not allowed as well:

```
a         ::=                # empty, syntax error
```

While a syntactic form with no choices is a syntax error within ALGT, it is possible to define an empty form through recursion:

```
a       ::= a

a       ::= b
b       ::= a
```

Note that such an empty set is, by necessity, defined by using *left recursion*.

1.2.4 Left recursive grammers

A syntactic form is recursive if it is defined in terms of itself, allowing concise definitions of arbitrary depth. All practical programming languages do have grammers where syntactic forms are recursive. An example would be types:

```
type    ::= baseType "->" type | ...
```

Left recursion is when this recursion occurs on a leftmost position in a sequence:

```
a ::= ... | a "b" | ...
```

While advanced parser-algorithms, such as *LALR-parsers* can handle this fine, it is not allowed in ALGT:

- First, this makes it easy to port a syntax created for ALGT to another parser toolchain - which possibly can't handle left recursion too.
- Second, this allows for a extremely simple parser implementation.
- Thirdly, this prevents having empty sets such as `a ::= a`.

We can easily detect this left recursion algorithmically, with following algorithm:

```

# For each sequence in each syntactic form,
# remove all but the first element
for each syntactic_form in syntax:
        for each choice in syntactic_form:
                choice.remove(1..)

# Remove all concrete tokens (including builtins)
for each syntactic_form in syntax:
        for each choice in syntactic_form:
                choice.removeTokens()


empty_rules = syntax.getEmptyRules()
while empty_rules.hasElements():
        # remove each empty rule and occurences of it in other rules
        for empty_rule in empty_rules:
                syntax.remove(empty_rule)
                for each syntactic_form in syntax:
                        for each choice in syntactic_form:
                                choice.remove(empty_rule)
        empty_rules = syntax.getEmptyRules


if syntax.isEmpty():
        # all clear!
else:
        error("Left recursion detected: "+syntax)
```

To make this algorithm more tangible, consider following syntax:

```
a       ::= "a" | "b" | "c" "d"
b       ::= a
c       ::= b | c "d"
```

First, the tail from each sequence is removed, e.g. sequence `"c" "d"` becomes `"c"`:

```
a       ::= "a" | "b" | "c"
b       ::= a
c       ::= b | d
d       ::= c
```

Now, all tokens, everything that is not a call to another syntactic form, is erased:

```
a       ::=             # empty
b       ::= a
c       ::= b | d
d       ::= c
```

At this point, the main loop is entered: all empty rules and their calls are deleted:

```
b       ::=             # empty
c       ::= b | d
d       ::= c
```

In the next iteration, `b` is removed as well:

```
c       ::= d
d       ::= c
```

At this point, no rules can be removed anymore. Only rules containing left recursion remain, for which an error message can be generated.

1.2.5 Uniqueness of sequences

When a parsetree is given, we want to be able to pinpoint exactly which syntactic form parsed it, as this can be used to minimize the set representation later on. In order to do so, we expect the syntax not to contain duplicate sequences.

```
a ::= ... | "a" | ...
b ::= ... | "a" | ...

x ::= "x"
c ::= ... | a x | ...
d ::= ... | a x | ...
```

A parsetree containg `"a"` could be parsed with both `a` and `b`, which is undesired; the sequence `"a" "x"` could be parsed with both `c` and `d`. To detect this, we compare each sequences with each every other sequence for equality. When such duplicate sequences exist, the language designer is demanded to refactor this sequence into a new rule:

```
aToken  ::= "a"
a ::= ... | aToken | ...
b ::= ... | aToken | ...

x ::= "x"
ax ::= a x
c ::= ... | ax | ...
d ::= ... | ax | ...
```

This is not foolproof though. Some sequences might embed each other, as in following syntax:

```
a ::= "a"
b ::= a | "b"

c ::= "c"
d ::= c | "d"

x ::= a d
y ::= b c
```

Here, the string `a c` might be parsed with both syntactic forms `x` and `y`. There is no straightforward way to refactor this, without making things overly complicated. Instead, runtime annotations are used to keep track of which form originated a parsetree.

The uniqueness-constraint is merely added to keep things simpler and force the language designer to write a language with as little duplication as possible.

With all these properties in place, an efficient set representation can be designed.

1.3 Representing sets of values

The goal of this section is to make collecting metafunctions. Where a metafunction takes a parsetree and transforms it to another parsetree, a collecting metafunction takes a *set* of parsetrees and converts them into another *set* of parsetrees. To make matters worse, these sets might be infinite.

In this chapter, we construct a general **representation** for such a set of parsetrees, exploiting the properties of any syntax, as outlined in the previous chapter.

1.3.1 Representing sets

The first step to abstract interpretation is to represent arbitrary syntactic sets. We will show how to do this, using following example syntax:

```
baseType        ::= "Bool" | "Int"
typeTerm        ::= baseType | "(" type ")"
type            ::= typeTerm "->" type | typeTerm
```

1.3.2 Sets with concrete values

A set with only concrete values can be simply represented by giving its inhabitants; the set `baseType` can be represented the following way:

```
{ "Bool", "Int" }
```

We might also represent sequences of concrete values, in a similar way:

```
{"Bool" "->" "Bool"}
```

We could also create a set with, for example, all function types with one argument:

```
{ "Bool" "->" "Bool"
, "Bool" "->" "Int"
, "Int" "->" "Bool"
, "Int" "->" "Int" }
```

1.3.3 Symbolic sets

A set can also be represented *symbolically.* For example, we might represent `baseType` also as:

```
{ baseType } = { "Bool", "Int" }
```

While concrete values are written with double quotes around them, symbolic representations are not.

We can also use this symbolic representation in a sequence, with any number of concrete or symbolic values:

```
  { baseType "->" baseType }
```

Which would be a succint notation for:

```
= { "Int" "->" baseType, "Bool" "->" baseType}
= { "Bool" "->" "Bool"
  , "Bool" "->" "Int"
  , "Int" "->" "Bool"
  , "Int" "->" "Int" }
```

1.3.4 Infinite sets

This symbolic representation gives rise to a natural way to represent infinite sets through inductive definitions, such as `typeTerm`:

```
type ::= { baseType, "(" type ")" }
     = { "Bool", "Int", "(" typeTerm "->" type ")" , "(" typeTerm ")"}
     = { "Bool", "Int", "(" "Bool" ")", "(" "Int" ")", ...
     = ...
```

A symbolic representation is thus a set containing sequences of either a concrete value or a symbolic value.

1.3.5 Set representation of a syntax

This means that the BNF-notation of a syntax can be easily translated to this symbolic representation. Each choice in the BNF is translated into a sequence, rulecalls are translated into their symbolic value.

This is equivalent to the BNF-notation.

```
baseType        ::= "Bool" | "Int"
typeTerm        ::= baseType | "(" type ")"
type            ::= typeTerm "->" type | typeTerm
```

becomes

```
baseType == {"Bool", "Int"}
typeTerm == {baseType, "(" type ")"}
type     == {typeTerm "->" type, typeTerm}
```

Note that, per inclusion, `baseType` is a subset of `typeTerm`, and `typeTerm` is a subset of `type`.

1.3.6 Defining α and γ

Now that we have acquired this representation, we might define the *abstraction* and *concretization* functions for our actual abstract interpretation:

$$\begin{aligned} \alpha(v) &= \{v\} \\ \gamma(R) &= R \\ compose(R,S) &= R \cup S \end{aligned}$$

These definitions satisfy earlier mentioned properties trivially, *monotonicity* and *soundness.*

Lemma 1. *α (over sets) is monotone:*

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

Lemma 2. *γ is monotone:*

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

Lemma 3. *α and γ are sound:*

$$\begin{aligned} & n \in \gamma(\alpha(n)) \\ = \;& n \in \gamma(\{n\}) \\ = \;& n \in \{n\} \end{aligned}$$

and

$$\begin{aligned} & R \in \alpha(\gamma(R)) \\ = \;& R \in \alpha(R) \\ = \;& R \in \{R\} \end{aligned}$$

With these, we can convert the concrete parsetree into the domain of sets. Furthermore, we know that using this interpretation makes sense. However, we're still lacking the operations to actually interpret functions with them.