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 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".

In other words, a function designed to work on integers can be interpreted to suddenly work on other objects (such as the representation of the signs), so that the resulting signs tells something about the integers a function might return.

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.

```
1 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.

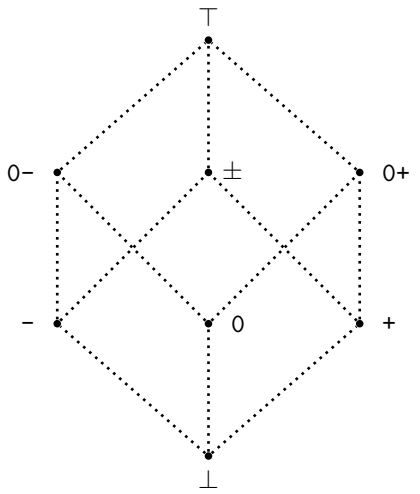

Figure 1.2: The symbols representing sets. Some sets, such as `+` are embedded in others, such as `0+`. These embeddings form a lattice.

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:

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

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:

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

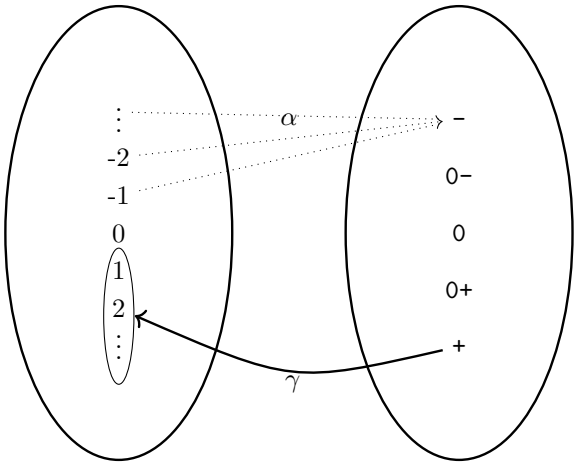

Figure 1.3: Concretization and abstraction between integers and signs

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. 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, giving the same result in an efficient way:

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

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:

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

Translation from and to the abstract domain are 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:

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

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). 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 1.3.

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.

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

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

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

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:

$$n \in \gamma(\alpha(n))$$
or equivalent
$$X \subseteq \alpha(Y) \Rightarrow Y \subseteq \gamma(X)$$

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

$$p \in \alpha(\gamma(p))$$
or equivalent
$$Y \subseteq \gamma(X) \Rightarrow X \subseteq \alpha(Y)$$

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:

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

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

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

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

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:

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

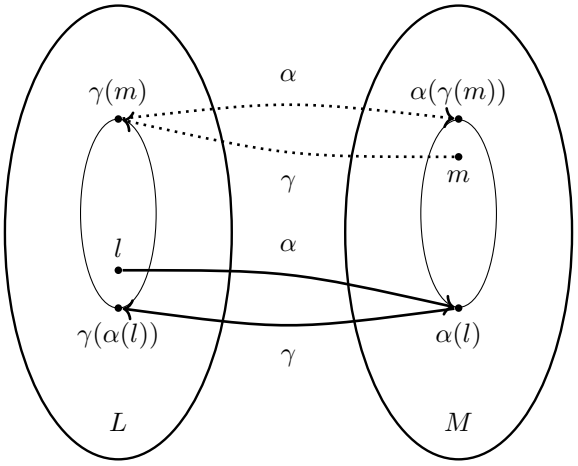

Figure 1.4: Galois-connection, visualized

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 latter case, each element the choice is also embedded into declared syntactic form.

In the following example, both `bool` and `int` are embedded into `expr`, visualized by ALGT in 1.5:

```
1 bool    ::= "True" | "False"
2 int     ::= Number       # Number is a builtin, parsing integers
3 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 1.6.

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 whether this should be included in the parsetree. Therefore, it is not allowed.

As example, consider following syntax:

```
1 a       ::= "=" | ""
2 b       ::= "x" a "y" | "x" "y"
3 c       ::= a as
```

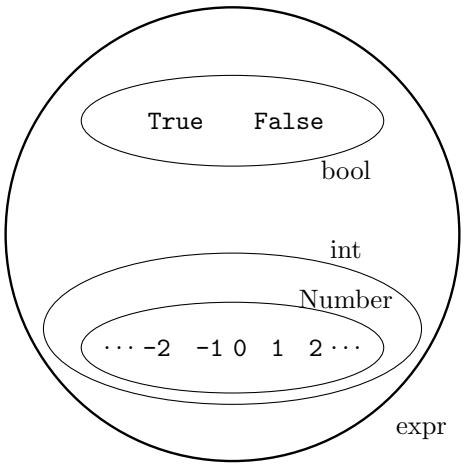

Figure 1.5: Nested syntactic forms

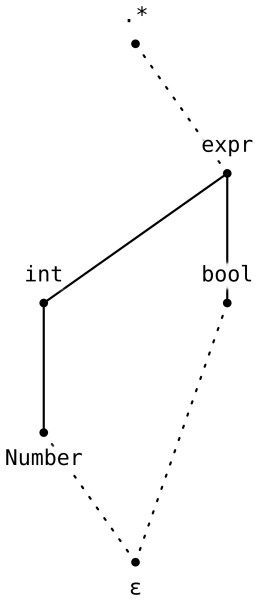

Figure 1.6: A simple subtyping relationship

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 the 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:

```
1 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:

```
1 a        ::= a
2
3 a        ::= b
4 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:

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

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

```
1 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. The algorithm itself can be found in figure **??**. To make this algorithm more tangible, consider following syntax:

```
1 a        ::= "a" | "b" | "c" "d"
2 b        ::= a
3 c        ::= b | c "d"
```

First, the tail from each sequence is removed, e.g. sequence `"c" "d"` becomes `"c"`. This is expressed in lines 3-5 and has following result:

```
1 a        ::= "a" | "b" | "c"
2 b        ::= a
3 c        ::= b | d
4 d        ::= c
```

Now, all tokens, everything that is not a call to another syntactic form, is erased (lines 8-10):

```
1 a        ::=              # empty
2 b        ::= a
3 c        ::= b | d
4 d        ::= c
```

At this point, the main loop is entered: all empty rules and their calls are deleted (lines 16 till 21 for actual deletion):

```
1 b        ::=              # empty
2 c        ::= b | d
3 d        ::= c
```

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

```
1 c       ::= d
2 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 (lines 24 till 26).

```
1  # For each sequence in each syntactic form,
2  # remove all but the first element
3  for each syntactic_form in syntax:
4          for each choice in syntactic_form:
5                  choice.remove(1..)
6
7  # Remove all concrete tokens (including builtins)
8  for each syntactic_form in syntax:
9          for each choice in syntactic_form:
10                 choice.removeTokens()
11
12
13 empty_rules = syntax.getEmptyRules()
14 while empty_rules.hasElements():
15         # remove each empty rule and occurences of it in other rules
16         for empty_rule in empty_rules:
17                 syntax.remove(empty_rule)
18                 for each syntactic_form in syntax:
19                         for each choice in syntactic_form:
20                                 choice.remove(empty_rule)
21         empty_rules = syntax.getEmptyRules
22
23
24 if syntax.isEmpty():
25         # all clear!
26 else:
27         error("Left recursion detected: "+syntax)
```

Figure 1.7: The algorithm to detect left recursion in a syntax

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. Furthermore, it helps the typechecker to work more efficient and with less errors.

1.3 Representing sets of values

In this chapter, a general **representation** for a (possibly infinite) set of parsetrees is constructed. This representation is used as abstract domain for metafunctions, lifting any metafunction from a metafunction on parsetrees to a metafunction of sets of parsetrees. This set representation is constructed using the properties outlined in the previous chapter, using a small syntax as example:

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

1.3.1 Sets with concrete values

A set with only concrete values is simply represented by giving its inhabitants; the set `baseType` is thus represented as following:

```
{ "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" }
```

1.3.2 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.

This symbolic representation can be used in a sequence too, with any number of concrete or symbolic values intermixed:

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

Which would be a succint notation for:

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

In other words, this notation gives a compact representation of bigger sets, which could be exploited; such as applying a function on the entire set at once.

1.3.3 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.

1.3.4 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.

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

becomes

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

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

1.3.5 Conclusion

This representation of sets offer a compact representation of larger, arbitrary sets of parsetrees. As seen, the sets represented might even contain infinite elements. This representation could form the basis of many operations and algorithms, such as abstract interpretation. These algorithms are presented in the next section.

1.4 Operations on representations

In the previous chapter, an efficient and compact representation was introduced for sets of parsetrees - a big step towards multiple usefull algorithms and abstract interpreation of metafunctions.

However, in order to construct these algorithms, basic operations are needed to transform these sets. These operations are presented here.

1.4.1 Addition of sets

Addition is the merging of two set representations. This is implemented by simply taking all elements of both set representations and removing all the duplicate elements.

Per example, `{"Bool"} + {baseType}` yields `{"Bool", baseType}`. Note that `"Bool"` is embedded within basetype, the refolding operation (defined below) will remove this.

1.4.2 Unfolding

The first important operation is unfolding a single level of the symbolic representation. This operation is usefull when introspecting the set, e.g. for applying pattern matches, calculating differences, ...

The unfolding of a set representation is done by unfolding each of the parts, parts which could be either a concrete value, a symbolic value or a sequence of those. Following paragraphs detail on how the unfold of such a part is calculated.

Unfolding concrete values

The unfold of a concrete value is just the concrete value itself:

```
1 unfold("Bool") = {"Bool"}
```

Unfolding symbolic values

Unfolding a symbolic value boils down to replacing it by its definition. This implies that the definition of each syntactic form should be known; making unfold a context-dependant operation. For simplicity, it is assumed that the syntax definition is passed to the unfold operation.

Some examples of unfolding a symbolic value would be:

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

Note the usage of parentheses around `(typeTerm "->" type")`. This groups the sequence together and is needed to prevent ambiguities later on. Such ambiguities might arise in specific syntaxes, such as a syntax containing following definition:

```
1 subtraction == {number "-" subtraction, number}
```

Unfolding subtraction two times would yield:

```
1 subtraction == { ..., number "-" number "-" number, ... }
```

This is ambiguous. The expression instance `3 - 2 - 1` could be parsed both as `(3 - 2) - 1` (which equals 0) and as `3 - (2 - 1)` (which equals 2). The syntax definition suggests the latter, as `subtraction` is defined in a right-associative way. To mirror this in the sequence representation, parentheses are needed:

```
subtraction == { ..., number "-" (number "-" number), ... }
```

Unfolding sequences

To unfold a sequence, each of the parts is unfolded. Combining the new sets is done by calculating the cartesian product:

```
unfold(baseType "->" baseType)
== unfold(baseType) × unfold("->") × unfold(baseType)
== {"Bool", "Int"} × {"->"} × {"Bool", "Int"}
== { "Bool" "->" "Bool"
   , "Bool" "->" "Int"
   , "Int" "->" "Bool"
   , "Int" "->" "Int" }
```

Unfolding set representions

Finally, a set representation is unfolded by unfolding each of the parts and collecting them in a new set:

```
unfold({baseType, baseType "->" baseType})
= unfold(baseType) + unfold(baseType "->" baseType)
= {"Bool", "Int"}
  + {"Bool" "->" "Bool"
    , "Bool" "->" "Int"
    , "Int" "->" "Bool"
    , "Int" "->" "Int" }
= { "Bool"
  , "Int"
  , "Bool" "->" "Bool"
  , "Bool" "->" "Int"
  , "Int" "->" "Bool"
  , "Int" "->" "Int" }
```

Directed unfolds

Throughout the text, unfolding of a set is often needed. However, an unfolded set can be big and unwieldy for this printed medium, especially when only a few elements of the set matter. In the examples we will thus only unfold the elements needed at hand and leave the others unchanged, thus using a *directed unfold*. It should be clear from context which elements are unfolded.

1.4.3 Refolding

Refolding attempts to undo the folding process. While not strictly necessary, it allows for more compact representations throughout the algorithms - increasing speed - and a more compact output - increasing readability.

For example, refolding would change `{"Bool", "Int", "Bool" "->" "Int"}` into `{baseType, "Bool" "->" "Int"}`, but also `{type, typeTerm, "Bool"}` into `{type}`.

Refolding is done in two steps, repeated until no further folds can be made:

- Grouping subsets (e.g. `"Bool"` and `"Int"`) into their symbolic value (`baseType`)
- Filter away values that are included in another symbolic value (e.g. in `{"Bool", type}`, `"Bool"` can be omitted, as it is included in `"type"`)

This is repeated until no further changes are possible on the set. The entire algorithm can be found in **??**, following paragraphs detail on each step in the main loop.

Grouping sets

Grouping tries to replace a part of the set representation by its symbolic representation, resulting in an equivalent set with a smaller representation (line 11 and 12 in the algorithm).

If the set defining a syntactic form is present in a set representation, the definition set can be folded into its symbolic value.

E.g. given the definition `baseType == {"Bool", "Int}`, we can make the following refold as each element of `baseType` is present:

```
1 refold({"Bool", "Int", "Bool" "->" "Int"})
2         = {baseType, "Bool" "->" "Int"}
```

Grouping sequences

Refolding elements within a sequence is not as straightforward. Consider following set:

```
1 { "Bool" "->" "Bool"
2 , "Bool" "->" "Int"
3 , "Int" "->" "Bool"
4 , "Int" "->" "Int" }
```

This set representation can be refolded, using the following steps:

- First, the sequences are sorted in buckets, where each sequence in the bucket only has a single different element (line 17):

```
1   ["Bool" "->" "Bool", "Bool" "->" "Int"]
2   ["Int" "->" "Bool", "Int" "->" "Int"]
```

- Then the different element in the sequences are grouped in a smaller set (line 20):

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

- This smaller *difference set* is unfold recursively (line 24):

```
1   "Bool" "->" unfold({"Bool", "Int"})
2   "Int" "->" unfold({"Bool", "Int"})
```

- This yields a new set; `{"Bool" "->" baseType, "Int" "->" baseType}`. As the unfolding algorithm tries to reach a fixpoint, it will rerun. This yields a new bucket, on which the steps could be repeated:

```
1   ["Bool" "->" baseType, "Int" "->" baseType]
2   -> {"Bool", "Int"} "->" baseType
3   -> unfold({"Bool", "Int"}) "->" baseType
4   -> baseType "->" baseType
```

This yields the expression we unfolded earlier: `baseType "->" baseType`.

Filtering out subvalues

The second step in the algorithm is the removal of already represented values. Consider `{ baseType, "Bool"}`. As the definition of `baseType` includes `"Bool"`, it is unneeded in this representation.

This can be done straightforward, by comparing each value in the set against each other value and checking wether this is contained in it (line 41 and 42).

```
1  refold(syntax, repr):
2      do
3          # Save the value to check if we got into a fixpoint
4          old_repr = repr;
5
6          # Simple, direct refolds
7          for (name, definition) in syntax:
8              # The definition set is a subset of the current set
9              # Remove the definition set
10             # replace it by the symbolic value
11             if definition ⊆ repr:
12                 repr = repr - definition + {name}
13
14         for sequence in repr:
15             # a bucket is a set of sequences
16             # with exactly one different element
17             buckets = sortOnDifferences(repr)
18             for (bucket, different_element_index) in buckets:
19                 # extract the differences of the sequence
20                 different_set = bucket.each(
21                         get(different_element_index))
22
23                 # actually a (possibly smaller) set
24                 folded = refold(syntax, different_set)
25
26                 # Extract the identical parts of the bucket
27                 (prefix, postfix) = bucket.get(0)
28                         .split(different_element_index)
29                 # Create a new set of sequences,
30                 # based on the smaller folded set
31                 sequences = prefix × folded × postfix
32
33                 # remove the old sequences,
34                 # add the new elements to the set
35                 repr = repr -  bucket + sequences
36
37         for element in repr:
38             for other_element in repr:
39                 if element == other_element:
40                     continue
41                 if other_element.embeds(element):
42                     repr = repr - element
43
44     while(old_repr != repr)
45     return repr
```

Figure 1.8: The algorithm to refold a set representation

1.4.4 Resolving to a syntactic form

Given a set, it can be usefull to calculate what syntactic form contains all of the elements of the set.
E.g., each element from `{"Bool", "(" "Int" ")", "Int"}` is embedded in `typeTerm`.

This can be calculated quite easily:

- First, every sequence is substituded by the syntactic form which created it, its generator. In the earlier example, this would give:

```
1 | {baseType , typeTerm , baseType}
```

- The least common supertype of these syntactic forms gives the syntactic form embedding all. As noted in chapter 1.2.2, the supertype relationship of a syntax forms an order. Getting least common supertype thus becomes calculating the meet of these types (`typeTerm`).

1.4.5 Subtraction of sets

Subtraction of sets enabes a lot of usefull algorithms, but is quite complicated.

Subtraction is performed on each sequence in the set. This subtraction of a single element might result in no, one or multiple new elements. There are a few cases to consider, depending on what is subtracted.

We will split them up as following:

- A concrete value is subtracted from a sequence
- A symbolic value is subtracted from a sequence
- A sequence is subtracted from a symbolic value
- A sequence is subtracted from a sequence

Subtraction of concrete values

Subtraction of a concrete value from a concrete sequence is straightforward: we check wether the sequence contains one single element and that this element is the same as the one we subtract from:

- `"Bool" - "Bool"` is empty
- `"Int" - "Bool"` is `"Int"`

If the sequence is a single symbolic value, we check if the symbolic value embeds the concrete value. If that is the case, we unfold and subtract that set recursively:

- `baseType - "Bool"` equals `{"Bool", "Int"} - "Bool"`, resulting in `{"Int"}`
- `type - "("` equals `type`

Subtraction of a symbolic value from a sequence

When a symbolic value is subtracted from a sequence, we first check wether this the sequence is embedded in this symbolic value:

- `"Bool" - baseType` is empty, as `"Bool"` is embedded in `baseType`
- `baseType - baseType` is empty, as baseType equals itself

- `"Bool" - type` is empty as `"Bool"` is embedded in `type` (via `typeTerm` and `baseType`)
- `baseType - type` is empty too, as it is embedded as well
- `("(" type ")") - type` is empty, as this is a sequence inside `typeTerm`

If the symbolic value does not embed the sequence, it might still subract a part of the sequence.

This is the case if the sequence embeds the symbolic value we wish to subtract. If that is the case, we unfold this sequence, and subtract each element in the set with the subtrahendum:

- `typeTerm - baseType` becomes `{baseType, "(" type ")"} - baseType`, resulting in `{"(" type ")"}`

Subtraction of a sequence from a symbolic value

This is straightforward too. Given the symbolic value, we check wether it shadows the sequence we want to subtract. If this is the case, we unfold this symbolic value and subtract the sequence from each of the elements. If no shadowing occurs, we just return the symbolic value.

```
1 ‘type - ("(" type ")")
2 = {typeTerm "->" type, typeTerm} - ("(" type ")")
3 = {typeTerm "->" type, baseType, "(" type ")"} - ("(" type ")")
4 = {typeTerm "->" type, baseType}
```

Subtraction of a sequence

The last case is subtracting a sequence from another sequence. As this is rather complicated, we start with an example., we start with an example before giving the algorithmic approach.

Example As all good things in programming, this subtraction relies on recursion; we will follow the example through the call stack.

1. Consider `("(" type ")") - ("(" ("Bool" "->" type) ")")`. It is intuitively clear that the parentheses `"("` and `")"` should remain, and that we want to subtract `type - ("Bool" "->" type)`.
2. Here we subtract a sequence again. We unfold `type`, to yield: [1]

   {typeTerm, typeTerm “->” type} - (“Bool” “->” type)
3. In the set, we now have two values we should subtract:

   - `typeTerm - ("Bool" "->" type)` yields `typeTerm`, as neither shadows the other.
   - `(typeTerm "->" type) - ("Bool" "->" type)` can be aligned. It is clear we'll want to have `typeTerm - "Bool"` and `type - type` and let `"->"` unchanged.
4. `typeTerm - "Bool"` gives us `{baseType, "(" type ")"} - "Bool"`, resulting in `{"Int", "(" type ")"}`. It is important to note that `(Bool)` is *still* an element of this set, despite “Bool” without parentheses is not. Despite having the same semantics, these are two syntacticly different forms!

[1] for practical reasons, I swapped the order of both elements in the text. As this is a set, that doesn't matter.

5. `type - type` is empty.

At this point, we have all the ingredients necessary, and we can put our subtraction back together.

5. The biggest puzzle is how to put `(typeTerm "->" type) - ("Bool" "->" type)` together. Intuitivelly, we'd expect it to be `{"Int" "->" type, "(" type ")" "->" type}`.
   More formally, the directed unfold of
   `typeTerm "->" type` is `{ "Bool" "->" type, "Int" "->" type, ("(" type ")") "->" type}`.
   Subtracting `"Bool" "->" type` yields our former result.

6. We put together the results of the recursive calls, being:

   - `{typeTerm}`
   - `{"Int" "->" type, "(" type ")" "->" type}`
     This results in `{"Int" "->" type, "(" type ")" "->" type, typeTerm}`

7. We unfolded `type` to subtract the sequence, yielding
   `{"Int" "->" type, "(" type ")" "->" type, typeTerm}`

8. We put the parentheses back around each expression:
   `{"(" ("Int" "->" type) ")", "(" "(" type ")" "->" type ")", "(" typeTerm ")"}`,
   giving the desired result.

Algorithm So, how do we algorithmically subtract two sequences?

Consider sequence `a b`, where `a` unfolds to `a1, a2, a3, ...` and `b` unfolds to `b1, b2, b3, ....` This means that `a b` unfolds to `a1 b1, a1 b2, a1 b3, ... , a2 b1, a2 b2, ...`

Now, if we would subtract sequence `a1 b1` from this set, only a single value would be gone, the result would nearly be the original cartesian product.

To do this, we first calculate the pointwise differences between the sequences. We align both sequences[2] and calculate the difference:

```
a - a1= a - a1 = {a2, ...}
b - b1 = {b2, ...}
```

Now, the resulting sequences are `{(a - a1) b, a (b - b1)}`.

Generalized to sequences from aribitrary length, we replace each element of the sequence once with the pointwise difference:

```
a b c d ... - a1 b1 c1 d1  ...
= { (a - a1) b c d ...
  , a (b - b1) c d ...
  , a b (c - c1) d ...
  , a b c (d - d1) ...
  }
```

We apply this on the example `(typeTerm "->" type) - ("Bool" "->" type)`, yielding following pointwise differences:

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

Resulting in:

[2] This implies both sequences have the same length. If these don't have the same length, just return the original sequence as the subtraction does not make sense anyway.

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

1.5 Algorithms using abstract interpretation

Now that we have our basic building blocks and operations, quite some usefull algorithms can be built using those.

1.5.1 Calculating a possible pattern match

We can compare a representation against a pattern match:

```
{type} ~ (T1 "->" T2)
```

Ultimatly, we want to calculate what variables might be what parsetrees. We want to create a store σ, mapping each variable on a possible set:

$$\sigma = \{T1 \rightarrow \ \{\texttt{typeTerm}\}\ , T2 \rightarrow \ \{\texttt{type}\}\ \}$$

Note that *no* syntactic form might match the pattern, as a match might fail. When the input is the entire syntactic form, this is a sign the clause is dead.

There are four kinds of patterns:

- A pattern assigning to a variable
- A pattern assigned to a variable which is already encountered
- A pattern comparing to a concrete value
- A pattern deconstructing the parse tree

Variable assignment

When a set representation S is compared against a variable assignment pattern `T`, we update the store σ with $T \rightarrow S$.

If the variable `T` were already present in the store σ, we narrow down its respective set to the intersection of both the old and new match. This happens when the variable has already been encountered, e.g. in another argument or another part of the pattern.

This is illustrated by matching `[{type}, {baseType}]` against `(T, T)`, which would yield $\sigma = \{T \rightarrow \ \{\texttt{baseType}\}\ \}$‘.

Concrete parsetree

When a set representation is compared against a concrete parsetree, we check wether this concrete value is embedded in the syntactic form. If this is not the case, the abstract pattern match fails.

Pattern sequence

When a set representation S is compared against a pattern sequence, we compare each sequence in the set with the pattern. It might be needed to unfold nonterminals, namely the nonterminal embedding the sequence.

Note that parsetree-sequence and pattern sequence need to have an equal length. If not sequence of the right length can be found in S, the match fails.

As example, we match `{type} ~ (T1 "->" T2)`.

- First, we unfold `type` as it embeds the pattern: `{typeTerm "->" type, typeTerm} ~ (T1 "->" T2)`
- Then, we throw out `typeTerm`, it can't be matched as it does not embed the pattern: `{typeTerm "->" type} ~ (T1 "->" T2)`
- We match the sequences of the right length: `{typeTerm} ~ T1 ; {type} ~ T2`
- This yields the store:

$$\sigma = \{T1 \rightarrow \ \{\texttt{typeTerm}\}\ , T2 \rightarrow \ \{\texttt{type}\}\ \}$$

Multiple arguments

There are two ways to approach functions with multiple arguments:

- using currying or
- considering all arguments at once.

When using **currying**, the type signature would be read as `type -> (type -> type)`, thus `equals` is actually a function that, given a single input value, produces a new function. This is extremely usefull in languages supporting higher-order functions - which ALGT is not.

We rather consider the domain as another syntactic sequence: `{type} ×{type}`. The multiple patterns are fused together, in exactly the same way.

This gives also rise to a logical way to subtract arguments from each other - excatly the same as we did with set representations.

1.5.2 Calculating possible return values of an expression

Given what syntactic form a variable might be, we can deduce a representation of what a function expression might return. As function expressions are sequences with either concrete values or variables, the translation to a representation is quickly made:

- A concrete value, e.g. `"Bool"` is represented by itself: `{"Bool"}`
- A variable is represented by the types it might assume. E.g. if `T1` can be `{"(" type ")", "Bool"}`, we use this set to represent it
- A function call is represented by the syntactic form it returns; this is provided to the algorithm externally.
- A sequence is represented by the cartesian product of its parts: `"Bool" "->" T1` is represented with `{"Bool" "->" "(" type ")", "Bool" "->" "Bool"}`

Calculating which patterns matched

We can apply this to patterns too. As patterns and expressions are exactly the same, we can fill out the variables in patterns to gain the original, matched parsetree. This is used to calculate what patterns are *not* matched by a pattern.

1.5.3 Calculating the collecting function

This allows us already to execute functions over set representations, instead of concrete parsetrees. This comes with a single caveat: for recursive calls, we just return the codomain, as set.

Per example, consider function `dom`:

```
dom              : type -> type
dom("(" t ")")   = dom(t)
dom(T1 -> T2)    = T1
```

We might for example calculate what set we would get when we input `{type}` into `dom`:

The first clause yields:

```
{type} ~ ("(" t ")") → {type}
```

With fallthrough `{typeTerm "->" type, baseType}`, which is used as input for the second clause. This yields:

```
{typeTerm "->" type, baseType} ~ (T1 "->" T2) → T1 = typeTerm
```

The fallthrough is now `baseType`, our final result is `refold{typeTerm, type}`, thus `type`.

1.5.4 Calculating the function domain

Single argument functions

One way to calculate the domain, is by translating the patterns in sets, just like we did with the syntax. This can be done easily, as all patterns are typed. To get the domain, we just sum these sets together:

```
{"(" type ")", type "->" type}
```

Another approach is by taking the syntactic form of the signature, `{type}`, and subtract the patterns from it. This way, we derive which syntactic forms will *not* match:

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

Using this *fallthrough set*, we calculate the domain by subtracting it from the input type, yielding the same as earlier calculated:

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

Multiple argument functions

Consider the function `equals`

```
equals  : basetype -> basetype -> basetype
equals("Bool", "Bool")          = "Bool"
equals("Int", "Int")            = "Int"
```

For simplicity, we use `baseType`, to restrict input values solely to `{"Bool", "Int"}`. Also note that `"Bool" ×"Int")` is *not* part of it's domain.

The difference between two arguments is calculated just as the difference between two sequences (as it is the same). This is, we take n copies of the arguments, where n is the number of arguments, and subtract each argument once pointwise.

For clause 1, this gives:

```
[baseType, baseType] - ["Bool", "Bool"]
= {[baseType - "Bool", baseType], [baseType, baseType - "Bool"}
= {["Int", baseType], [baseType, "Int"]}
```

We repeat this process for each clause, thus for clause 2, this gives:

```
{["Int", baseType], [baseType, "Int"]} - ["Int", "Int"]
= { ["Int", baseType] - ["Int", "Int"]
  , [baseType, "Int"] - ["Int", "Int"] }
= { ["Int" - "Int", baseType]
  , ["Int", baseType - "Int"]
  , [baseType - "Int", "Int"]
  , [baseType, "Int" - "Int"] }
= { [ {} , baseType]
  , ["Int", "Bool"]
  , ["Bool", "Int"]
  , [baseType, {} ] }
= { ["Int", "Bool"]
  , ["Bool", "Int"]}
```

This gives us the arguments for which this function is not defined. The domain of the function are all arguments *not* captured by these sequences. The domain would thus be defined by subtracting the result from the input form.

This is algorithm is practical for small inputs, but can become slow in the presence of large types; the sequence set might contain up O(#Args*#TypeSize) elements. However, we can effectively pickup dead clauses on the way: clauses which can't match any input that is still available.

1.5.5 Calculating the codomain

We can also calculate what set a function might return. In the previous algorithm, we calculated what values a clause might receive at the beginning.

We use this information to calculate the set that a clause -and thus a function- might potentially return. For this, we use the earlier introduced abstract pattern matching and expression calculation. Afterwards, we sum all the sets.

For the first clause of the domain function, we would yield:

```
dom("(" t ")")  = dom(t)        <: {type}
Used patterns: {"(" type ")"};
Patterns falling through: {typeTerm "->" type, baseType}
```

For the second clause, we yield:

```
dom(T1 "->" T2) = T1    <: {baseType}
Used patterns: {baseType "->" type};
Patterns falling through: {baseType}
```

Summing all returned values and resolving them to the smallest common supertype, gives:

```
1 {baseType , type} = {type}
```

This already gives us some usefull checks for functions, namely a **pattern match totality checker** and a **clause livebility checker** (as we might detect a clause *not* consuming patterns.

But with a slight modification to this check, we can do better.

We can calculate a dictionary of what syntactic forms a function does return. Instead of initializing this set with the returned syntactic form (thus `dom` →`{type}`), we initialize it with empty sets (`dom` →`{}`). When we would use this to resolve function calls, we yield the following:

```
1 dom("(" t ")")  = dom(t)         <: {}
2 dom(T1 "->" T2) = T1             <: {baseType}
```

Summing into `{baseType}`

With this, we can update our dictionary to `dom --> {baseType}` and rerun.

```
1 dom("(" t ")")  = dom(t)         <: {baseType}
2 dom(T1 "->" T2) = T1             <: {baseType}
```

Summing into `{baseType}`. This does not add new information; in other words, there is no need for a new iteration.

This gives rise to another check, namely that the function signature is the **smallest possible syntactic form** and partial **infinite recursion check**.

Infinite recursion can -in some cases- be detected. If we were to release previous algorithm on following function:

```
1 f       : a -> a
2 f(a)    = f(a)
```

we would yield:

```
1 {f → {}}
```

Per rice theorem, we know it won't be possible to apply this algorithm to every program. The smallest possible syntactic form check would hang on:

```
1 f       : type -> type
2 f(t)    = "Bool" "->" f(t)
```