Contents

Chapter 1

Syntax Driven Abstract Interpretation

In this section, we transform a metafunction -a function on a parsetree- into a metafunction working on (possibly infinite) set of parsetrees. This is the main contribution of the dissertation, as it is a novel technique to automatically reason about programming languages. Furthermore, these analysises will be used to build gradualization.

- 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 a **efficient representation** to capture infinite sets of parsetrees.
- Afterwards, **operations on these setrepresentations are stated**.
- As last, we build usefull **algorithms** and checks with this algebra.

1.1 What is abstract interpretation?

Abstract interpretation is a collection of techniques to derive properties about programs, based on sound approximation. However, per Rice's theorem, it is generally impossable to make exact statements about every possible program.

Furhtermore, it is often impractical to calculate the value we might return. Therefore, we merely calculate for a given program what property might hold. Essential in our approach is that these properties can be combined in a monotone way. Often, the properties we work with, will have the form of a lattice.

Examples of abstract interpretation domains are:

- Working with the sign of functions
- Working with an upper or lower bound on integer functions
- Working with a set of possible returned values, called *collecting semantics*

In contrast to *abstract interpretation*, is *concrete* interpretation; this would be running the program.

The last important aspect of abstract interpretation is the translation of one property domain into another, often a costly -but more precise- domain into a

cheaper and less precise domain. This functions is called *abstraction* (α). The inverse function is *concretization* (γ). Abstraction and concretization form a Galois-connection.

First, we work out a simple example to get some feeling about the technique. Then we describe desired properties of abstraction α and concretization γ.

1.1.1 Successor example

Concrete interpretation

Consider the extremely simple example function `succ`:

```
1 succ    : Number -> Number
2 succ x  = x + 1
```

Working with the sign of functions

For this analysis, we are only interested in what sign our example might potentially return (thus `+`,`-`, `0`). We don't want to calculate each value of course.

Converting a concrete value into a signset (abstracting to the property domain) is straightforward:

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

Calculating the abstraction of a set boils down to calculating the abstract of each element and taking them all together. We will often use these definitions intermixed.

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

Calculating the set containing all values for a certain property value, is done with following *concretization function*:

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

But how to combine two signs? For example, it might turn out that a function might return values of both `0` or `+`. To cope with this, we will work with *sets*, which compose very naturally.

$$compose(N, M) = N \cup M$$

Again is taking the concretization of a set the union of concretizations of the elements:

$$\gamma(P) = \bigcup \{\gamma(p) | p \in P\}$$

Now, we'd want to know what sign we would yield for `succ +`, `succ 0` and `succ -`. For input property `P`, this is captured by

$$\alpha(\{\ \texttt{succ}\ (n) | n \in \gamma(\ \texttt{P}\ )\})$$

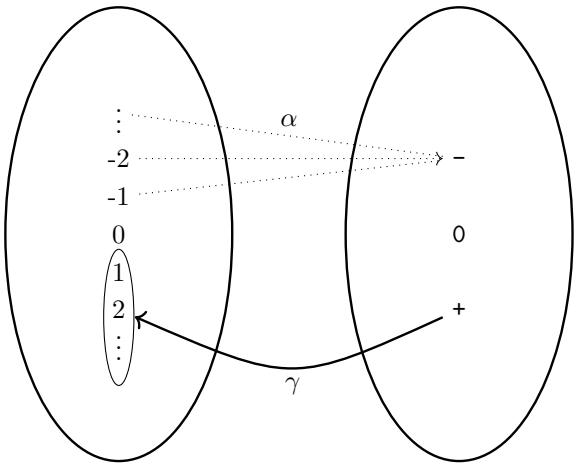

Figure 1.1: Concretization and abstraction relation for translation between numbers and signs

. In other word, we calculate the successor for each `n` in the category and then abstract the property.

For example, if we want to now what sign `succ 0` yields, we calculate:

$$\begin{aligned} & \alpha(\{\ \texttt{succ}\ (n) | n \in \gamma(\ \texttt{0}\ )\}) \\ = \ & \alpha(\{\ \texttt{succ}\ (n) | n \in \{0\}\}) \\ = \ & \alpha(\{\ \texttt{succ}\ (0)\}) \\ = \ & \alpha(\{1\}) \\ = \ & \{+\} \end{aligned}$$

If we would want to know what sign we would yield for a positive number, calculating

$$\alpha(\{\text{succ}(n) | n \in \gamma(+)\})$$

is intractable. Luckily can apply the properties of `+` to calculate the property abstractly. The property we'll uses here, is that the sum of two positive numbers, is positive.

Instead of working with concrete values, we use our sign property as *symbol*, and run our program with that:

$$\begin{aligned} & \textit{succ}\ \texttt{+} \\ = \ & \texttt{n}\ + \alpha(1) \\ = \ & \texttt{+}\ + \alpha(1) \\ = \ & \texttt{+}\ +\ \texttt{+} \\ = \ & \texttt{+} \end{aligned}$$

Analogously, we might repeat this with `-`. Sadly, we cannot conclude anything usefull out of this analysis; as the sum of zero and a positive number is positive, whereas the sum of a negative number and a positive number might be negative, zero or positive.

Our abstract evaluation would look as following:

$$
\begin{array}{rl}
 & succ\ \texttt{-} \\
= & \texttt{n} + \alpha(1) \\
= & \texttt{-} + \alpha(1) \\
= & \texttt{-} + \texttt{+} \\
= & \{\ \texttt{-,0,+}\ \}
\end{array}
$$

In summary, for the sign function, we can conclude that `succ` behaves as following:

$$
\begin{array}{lcl}
\texttt{succ (-\ )} & = & \{-, 0, +\} \\
\texttt{succ (0\ )} & = & \{+\} \\
\texttt{succ (+\ )} & = & \{+\}
\end{array}
$$

Working with ranges

Another possibility is to keep track of the range a value might be. Our abstract property is now denoted as `[n, m]` (where $n \leq m$). To go from a concrete value to a range, we use the following abstraction function:

$$\alpha(n) = \texttt{[n, n]}$$

Going the other way with the concretization function, is quite predictable:

$$\gamma(\ \texttt{[n, m]}\ ) = \{x | n \leq x \wedge x \leq m\}$$

The last question is how to compose two ranges. A function might return values between either range `[n1, m1]` or `[n2, m2]`.

$$compose(\ \texttt{[n1, m1]}\ ,\ \texttt{[n1, m2]}\ ) = \ \texttt{[}\ min(\ \texttt{n1}\ ,\ \texttt{n2}\ ), max(\ \texttt{m1}\ ,\ \texttt{m2}\ )\ \texttt{]}$$

If we have an abstract representation (e.g. `[2,5]`) for what range `succ` would return, we can calculate this:

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

Working with collecting semantics

At last, we can keep track of *all* possible values through the calculation. For example, if the input might be `{1,2,41}`, we might run our program *on all of these values*. At first glance, this is ridiculous. Why not run the program three times? However, this can be usefull, as this set representation might allow for efficient internal representation or to deduce other properties.

The abstraction, concretization and composition functions are trivial:

$$
\begin{array}{lcl}
\alpha(n) & = & \{n\} \\
\gamma(\{n1, n2, \ldots\} & = & \{n1, n2, ...\} \\
compose(N, M) & = & N \cup M
\end{array}
$$

Calculating the result for the example `{1,2,41}` gives:

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

Note that using the collecting semantics with a set, containing a single value, is exactly the concrete interpretation. This is guaranteed by the underlying deterministic semantics.

1.1.2 Desired properties

The functions α and γ should obey to some properties to make this approach work, namely *monotonicity* and *correctness*. It turns out that these properties form a **Galois connection** between the concrete values and the property domain of choice.

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:

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

Soundness

When we transform a concrete value into a property, we expect that property actually represents this value. A property represents a concrete value iff its concretization contains this value. This gives another important property:

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

On the other hand, if we calculate which concrete values correspond with a certain property p, we expect some of these values to exhibit property p:

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

This guarantees the *soundness* of our approach.

Consider we would not have this guarantee about α and γ, our approach would fail. As example, we change the functions which map numbers onto their sign, but we map `0` onto the negative range, *without changing concretization* :

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

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

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

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

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

Galois connection

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

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

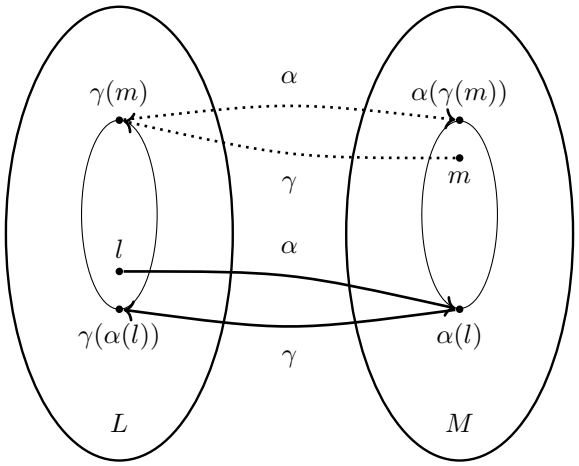

Figure 1.2: Galois-connection, visualized

1.2 Properties of the syntax

Before presenting the actual abstract interpretation on metafunctions, we first study necessary properties about the syntax. Some properties are inherent to each syntax, others should be enforced. For this last category, we present the necessary algorithms to detect satisfaction to these conditions.

1.2.1 Syntactic forms as sets

When we create a syntactic form, we actually declare a set of possible values for this syntactic form: `bool ::= "True" | "False"` is equivalent to declaring 'bool ::= {"True", False"}'. We will exploit this notion to create an efficient, collecting representation to describe values.

1.2.2 Subtyping

When a syntactic form a uses another bare syntactic form b in its definition, we say a embeds b it. In the following example, the set `expr` consists of `{"True", "False", "0", "1", "2", "3", ...}`, containing both `bool` and `int`.

```
1 bool    ::= "True" | "False"
2 int     ::= Number
3 expr    ::= bool | number
```

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 1.3.

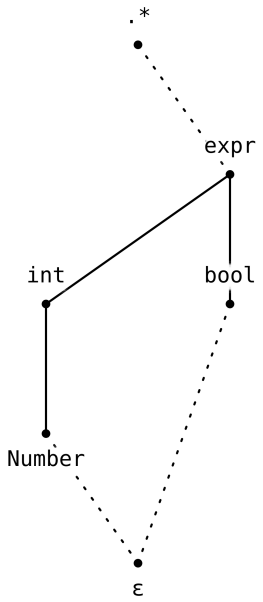

Figure 1.3: A simple subtyping relationship

1.2.3 Empty sets

Empty strings are not allowed. Consider syntax:

```
1 a       ::= "=" | ""
2 b       ::= "x" a "y"
```

```
3 c        ::= a as
```

Parsing `b` over string `x y` is ambiguous. Should the parsetree contain an element representing an empty `a` or not? Parsing `c` is even more troublesome: the parser might return an infinite list, containing only empty `a`-elements.

Empty rules are just as troublesome and are not allowed as well:

```
1 a        ::=              # empty
```

This also includes all kind of degenerate recursive calls:

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

Note that an empty set, by necessity, can only be defined by using *left recursion.*

1.2.4 Left recursive grammers

We allow recursive definitions, this is, we allow syntactic forms to be defined in terms of itself:

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

Left recursion is when this recursion is used on a leftmost position of a sequence:

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

While algorithms, such as *LALR-parsers* can handle this fine, we don't allow these.

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 easy parser implementation. Thirdly, this prevents having empty sets such as `a ::= a`.

We can easily detect this left recursion algorithmically, with a fixpoint algorithm. Consider the following syntax:

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

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

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

Now, we remove all tokens, thus everything that is not a call to another syntactic form:

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

At this point, we enter the main loop of the fixpoint. We remove all empty rules and their calls, until no rule can be removed:

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

Next iteration:

```
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.

```
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, we demand the programmer 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, we'll opt to work with runtime annotations on the parsetree which rule parsed it.

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.

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:

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

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

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

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

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

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

1.3.3 Symbolic sets

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

```
1 { 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:

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

Which would be a succint notation for:

```
1 = { "Int" "->" baseType, "Bool" "->" baseType}
2 = { "Bool" "->" "Bool"
3   , "Bool" "->" "Int"
4   , "Int" "->" "Bool"
5   , "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:

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

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:*

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

and

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

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.

1.4 Operations on representations

1.4.1 Unfolding

The first important operation is unfolding a single level of the symbolic representation. This is done by substituting the symbolic values by its definition. This operation would thus also need some context, namely these definitions.

Algorithmically unfolding a value is done as following:

Unfolding a concrete value, is just the concrete value itself:

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

Unfolding a symbolic value is the set, as it is defined:

```
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, as illustrated in following definition:

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

Unfolding this a few times towards subtraction, we might yield:

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

This is unclear. Should the syntactic expression `3 - 2 - 1` equal `(3 - 2) - 1 = 0` or `3 - (2 - 1) = 2`? Our syntax definition suggests the latter, as it is built in a right-associative way. To mirror this in the sequence, we add parentheses:

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

To unfold a sequence, we unfold each of the parts and take a cartesian product:

```
1 baseType "->" baseType
2 == {"Bool", "Int"} × {"->"} × {"Bool", "Int"}
3 == { "Bool" "->" "Bool"
4    , "Bool" "->" "Int"
5    , "Int" "->" "Bool"
6    , "Int" "->" "Int" }
```

At last, to unfold a symbolic set representation, we unfold each of the sequences in the set, and collect them:

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

1.4.2 Directed unfolds

Throughout the text, we will freely use *directed unfold*. This is an unfold of only certain elements in the set, often the ones we conveniently need 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}`

How does this work algorithmically? Actually, this is done in two steps:

- Group 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.

Grouping sets

Grouping simple sets is quite straightforward. To refold, we just look which known definitions are a subset.

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

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

Grouping complex sequences is not as straightforward as it seems. Consider

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

How will we proceed to refold this algorithmically? First, we sort these sequences in buckets where only a single element is different between the sequences.

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

We then single out the differences as a single set...

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

... and unfold these recursively:

```
"Bool" "->" baseType
"Int" "->" baseType
```

This yields us a new set; `{"Bool" "->" baseType, "Int" "->" baseType}`. As the unfolding algorithm tries to reach a fixpoint, it will rerun. This would yield a new bucket:

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

Where the different element would this time be the first element:

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

Refolding this yields our original expression `baseType "->" baseType`

Filtering out subvalues

The second step in the algorithm is the removal of already represented values. Consider `{ baseType, "Bool"}`. In the definition of `baseType` is "Bool" included, thus it is unneeded here. We say that `baseType` *shadows* `"Bool"`.

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

1.4.4 Resolving to a syntactic form

Given a set, it can be usefull to calculate what syntactic form does contain all of the elements of the set. E.g., given `{"Bool", "(" "Int" ")", "Int"}`, we want to know what syntactic form contains all of these values, namely `baseType`.

To do this, first we change every sequence by the syntactic form from which it was derived, its generator. For the example, this becomes `{baseType, typeTerm, baseType}`.

To get the least common supertype of those, we calculate the meet of all these types, being `typeTerm`.

1.4.5 Addition of sets

Another extremely usefull and straightforward operation is the addition of two sets. This is joining both sets, optionally refolding it.

1.4.6 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 ")"}`

1.4.7 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!

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.

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

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:

```
  { (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:

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

[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.

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`:

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

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

The first clause yields:

```
1 {type} ~ ("(" t ")") → {type}
```

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

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

The fallthrough is now `baseType`, our final result is `refold{typeTerm, type} = 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:

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

For the second clause, we yield:

```
1 dom(T1 "->" T2) = T1     <: {baseType}
2 Used patterns: {baseType "->" type};
3 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:

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

we would yield:

```
{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:

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