Contents

Chapter 1

Syntax Driven Abstract Interpretation

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

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

1.1 Abstract interpretation

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

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

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

```
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.

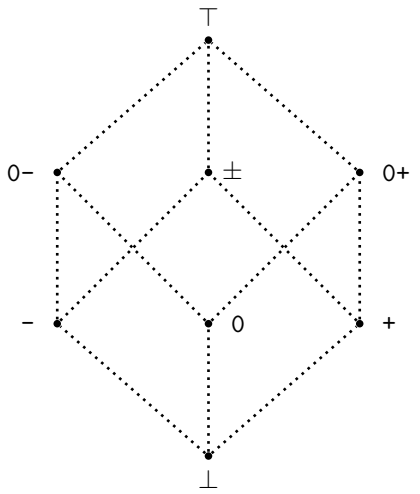

Figure 1.2: The symbols representing sets. Some sets, such as `+` are embedded in others, such as `0+`. These embeddings form a lattice.

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:

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

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

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

These functions, both visualized in figure 1.3, give us a general way to deduce the behaviour of a function in the abstract domain. The abstract output for a function can be calculated by converting the abstract input to the concrete domain, using γ, resulting in a set. For this set, the concrete function is applied on each element, resulting in a new concrete set. This new concrete set is abstracted, giving the output for the abstract function:

$$\alpha(map(f, \gamma(input)))$$

This formula is unusable in a practical implementation. Calculating the function for *each* element of the conrete set is costly and often impossible. However, it offers an excellent way to derive the theoritical properties, offering a hint on how the practical implementation can be made.

1.1.2 Ranges as abstract domain

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

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

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

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

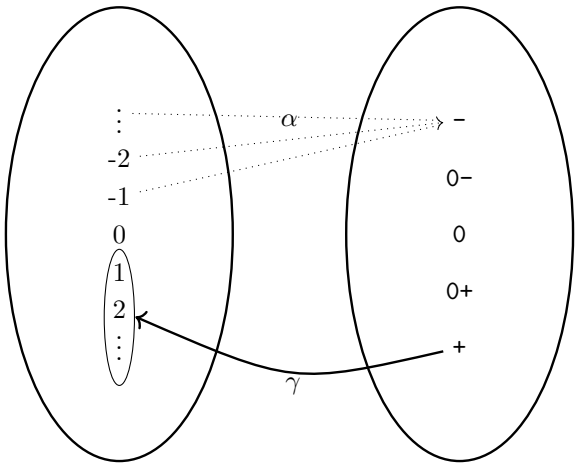

Figure 1.3: Concretization and abstraction between integers and signs

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

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

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

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

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

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

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

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

1.1.3 Collecting semantics

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

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

Translation from and to the abstract domain are trivially implemented. After all, the abstraction of a concrete value is the singleton containing 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:

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

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 inputs 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 constructed 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. The properties of the concretization and abstraction function also imply a **Galois connection** between the concrete and abstract domains.

Monotonicity of α and γ

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

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

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

This is 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, $\alpha(n)$ should 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:

$$m \in \alpha(\gamma(m))$$
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

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

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

Galois connection

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

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

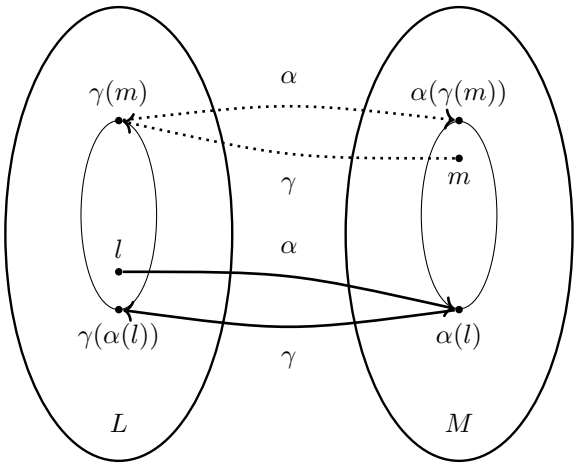

Figure 1.4: Galois-connection, visualized

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. In this section, we study the structure and properties of the syntax, which are used in the next section to construct a set representation for parsetrees.

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 for both the other properties studied and the efficient representation for sets introduced in section 1.2.

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 of the choice is also embedded into the declared syntactic form.

In the following example, both `bool` and `int` are embedded into `expr`, visualized by ALGT in figure 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 (see section 1.2.4) implies that no cycles can exist in this supertype relationship. This lattice can be visualized by ALGT, as in figure 1.6. Note that this lattice is cached in memory, allowing a lookup for the supertype relation.

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:

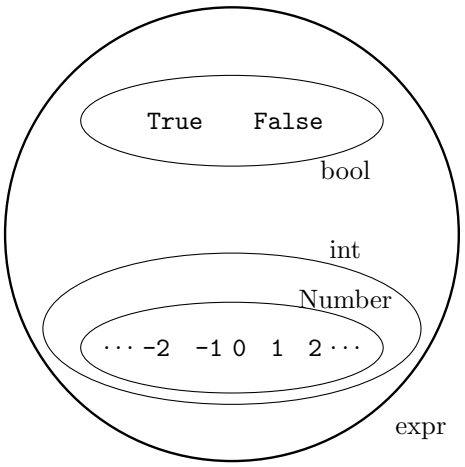

Figure 1.5: Nested syntactic forms

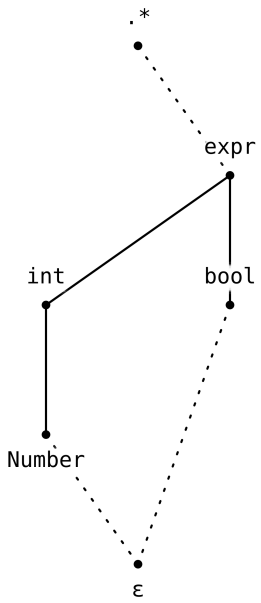

Figure 1.6: A simple subtyping relationship

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

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 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`.

Left recursion can be easily detected algorithmically. The algorithm itself can be found in figure 1.7. 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 syntactic forms 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 syntactic forms can be removed anymore. Only syntactic forms containing left recursion remain[1], 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] Technically, syntactic forms containing a left recursive form on a leftmost position will be included too.

1.2.5 Uniqueness of sequences

When a parsetree is given, pinpointing exactly which syntactic form parsed it is usefull, as this can be used to minimize the set representation later on. A syntax should thus not contain duplciate sequences.

```
1 a ::= ... | "a" | ...
2 b ::= ... | "a" | ...
3
4 x ::= "x"
5 c ::= ... | a x | ...
6 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:

```
1 aToken  ::= "a"
2 a ::= ... | aToken | ...
3 b ::= ... | aToken | ...
4
5 x ::= "x"
6 ax ::= a x
7 c ::= ... | ax | ...
8 d ::= ... | ax | ...
```

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

```
1 a ::= "a"
2 b ::= a | "b"
3
4 c ::= "c"
5 d ::= c | "d"
6
7 x ::= a d
8 y ::= b c
```

Here, the string `"a c"` might be parsed with both syntactic forms `x` and `y`. There is no straightforward way to detect or refactor such a construction, without making things overly complicated. In other words, the approach refactoring duplicate constructions can not be used. Instead, runtime annotations are used to keep track of which form originated a parsetree.

The uniqueness-constraint is still 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:

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

```
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" }
```

1.3.2 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.

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

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

Which would be a succint notation for:

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

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

is translated to

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

Figure 1.8: Translation of a syntax to set representation

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" ")", ...
     = ...
```

This notation allows abstract functions to run over infinite sets, another important feature of the abstract interpretation of metafunctions.

1.3.4 Set representation of a syntax

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, closesly resembling the original definition. This can be seen in figure 1.8.

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 be 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, constructing these algorithms requires basic operations 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 (see section 1.4.3) 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 to unfold each part.

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 implicitly.

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.

In the actual implementation, directed unfolds are sometimes used too. Depending on the context, the elements that should be unfolded are known. If not, refolding after an operation took place often has the same effect of a directed unfold.

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 embedded in another symbolic value (e.g. in `{"Bool", type}`, `"Bool"` can be omitted, as it is embedded in `"type"`)

This is repeated until no further changes are possible on the set. The entire algorithm can be found in figure 1.9, 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 folded recursively (line 24):

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

- This yields a new set; `{"Bool" "->" baseType, "Int" "->" baseType}`. As the folding algorithm tries to reach a fixpoint, grouping will be run again. 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 embedded values

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 is straightforward, by comparing each value in the set against each other value and checking whether this element is embedded in the other (line 41 and 42).

```
refold(syntax, repr):
    do
        # Save the value to check if we got into a fixpoint
        old_repr = repr;

        # Simple, direct refolds (aka. grouping)
        for (name, definition) in syntax:
            # The definition set is a subset of the current set
            # Remove the definition set
            # replace it by the symbolic value
            if definition ⊆ repr:
                repr = repr - definition + {name}

        for sequence in repr:
            # a bucket is a set of sequences
            # with exactly one different element
            buckets = sortOnDifferences(repr)
            for (bucket, different_element_index) in buckets:
                # extract the differences of the sequence
                different_set = bucket.each(
                        get(different_element_index))

                # actually a (possibly smaller) set
                folded = refold(syntax, different_set)

                # Extract the identical parts of the bucket
                (prefix, postfix) = bucket.get(0)
                        .split(different_element_index)
                # Create a new set of sequences,
                # based on the smaller folded set
                sequences = prefix × folded × postfix

                # remove the old sequences,
                # add the new elements to the set
                repr = repr -  bucket + sequences

        for element in repr:
            for other_element in repr:
                if element == other_element:
                    continue
                if other_element.embeds(element):
                    repr = repr - element

    while(old_repr != repr)
    return repr
```

Figure 1.9: 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 embeds all the elements in the set.
E.g., each element from `{"Bool", "(" "Int" ")", "Int"}` is embedded in `typeTerm`.
This can be calculated quite easily:

- First, every sequence is substituted 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 section 1.2.2, the supertype relationship of a syntax forms a lattice. 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 (such as liveability checks), but is quite complicated to implement.

Subtracting a set from another set is done by calculating the subtraction between all element of both sets. A single element, subtracted by another element, might result in no, one or multiple new elements.

There are a few different cases to consider, depending on what element is subtracted from what element. Following cases should be considered, each for which a paragraph will detail how subtraction is handled:

- 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

In these paragraphs, the term **subtrahend** refers to the element that is *subtracted*, whereas the **minuend** refers to the element that is *subtracted from*. As mnemonic, the minuend will di*min*ish, whereas the subtrahend *subtracts*.

Subtraction of concrete values from a sequence

Subtraction of a concrete value from a concrete sequence only has an effect if the subtrahend and minuend are the same. Subtracting a concrete value from a different concrete value or subtracting it from a sequence has no effect.

Some example subtractions are:

- `"Bool" - "Bool"` is `{}` (thus the empty set)
- `"Int" - "Bool"` is `{"Int"}`
- `("Int" "->" "Int") - "Int"` is `{("Int" "->" "Int")}`

If the minuend is a single symbolic value, the subtraction only has an effect if the subtrahend is embedded within this symbolic value. If that is the case, the symbolic value is unfolded to a new set, from which the subtrahend is subtracted recursively. This can be seen in following examples:

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

Subtraction of a symbolic value from a sequence

Subtracting a symbolic value from a sequence could result in the empty set, when the subtrahend (the symbolic set) embeds the minuend:

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

If the minuend is a single symbolic value which embeds the subtrahend, the resulting set is smaller than the minuend. Calculating the result is done by unfolding the minuend, from which the subtrahend is subtracted:

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

Subtraction of a sequence from a symbolic value

If the minuend (a symbolic value) embeds the subtrahend (a sequence), then the result of the subtraction can be calculated by unfolding the minuend and then subtracting the subtrahend from each element, as can be seen in the following example:

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

If the minuend (a symbolic value) does not embed the subtrahend, then the subtraction has no effect:

- `type - (expr "-" expr)` is `{type}`

Subtraction of a sequence from a sequence

The last case is subtracting a sequence from another sequence. Without losing generatlity, let the minuend to be the sequence `a b` and the subtrahend `a1 b1`, where `a` unfolds to `a1, a2, a3, ...` and `b` unfolds to `b1, b2, b3, ....` The sequence `a b` would thus unfold to the cartesian product, `a1 b1, a1 b2, a1 b3, ... , a2 b1, a2 b2, ...`

The subtraction would result in a set closely resembling the cartesian product, with only a single element gone, namely `a1 b1`. The resulting set, being `{ a2 b1, a3 b1, ...} + {a1 b2, a1 b2, ...} + {a1 b3, a2 b3, ...} + ...` can be factored as `{(a - a1) ×b} + {a ×(b - b1)}`, which is a compact and tractable representation for this set.

This factorization is calculated by taking the pointwise differences between the sequences, after which the product with the rest of the sequence is taken. The pointwise difference is calculated by subtracting every ith subtrahend sequence from the ith minuend sequence (which implies both sequences have the same length).

Generalized to sequences from aribitrary length, the subtraction can be calculated by pointwise replacement of each element in the minuend:

```
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) ...
  }
```

As example, the subtraction `(typeTerm "->" type) - ("Bool" "->" type)` yields following pointwise differences:

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

Resulting in the following set:

```
  { (typeTerm - "Bool") "->" type
  , typeTerm ("->" - "->") type
  , typeTerm "->" (type - type)}
= {{"Int", "(" type ")"} "->" type
  , typeTerm {} type    # an empty element implies an empty cartesian product
  , typeTerm "->" {}}
= {{"Int", "(" type ")"} "->" type}
= {"Int" "->" type, ("(" type ")") "->" type}
```

1.4.6 Conclusion

The set represention can be easily and efficiently transformed using the introduced operations, while still keeping the representation as small as possible. These operations form the basic building blocks for the further algorithms.

1.5 Lifting metafunctions to work over sets

In this chapter, the metafunctions are actually lifted to metafunctions over sets, using abstract interpretation. These metafunctions over sets will play a crucial role in later algorithms and the gradualization of programming languages.

The abstract interpretation framework, introduced in chapter 1.1 will guide the interpretation of patterns and expressions, the building blocks used to define functions; the set representation and operations introduced in the previous two chapters are used as data structure to make the computation feasible.

1.5.1 Translating metafunctions to the abstract domain

Metafunctions normally operate on parsetrees, which are elements of the concrete domain. In order to lift them to the abstract domain, sets of parsetrees, functions α (abstraction) and γ (concretization) are needed.

The abstraction-function is straightforward, as the smallest set representing a parsetree is the set containing only that parsetree.

Translating a set representation back to the concrete domain is done by enumerating all the elements represented in the set. The set representation, which possibly contains symbolic values, should thus be unfolded iteratively until only concrete values remain, reaching fixpoint for the unfold operation. The definitions of the abstraction and concretization functions can be found in figure 1.10.

$$\begin{array}{rcl} \alpha(\texttt{ v }) & = & \{\texttt{v}\} \\ \gamma(w) & = & unfold^*(w) \end{array}$$

Figure 1.10: Definition of abstraction (α) and concretization (γ)

The repeated unfolding of concretization is often impossible: a recursive definition (such as `type ::= baseType "->" type`) will result in infinite sets. The concretization function is practical to study the properties of the abstract interpretation, but not for actual implementation.

Monotonicity and Soundness

As seen in section 1.1.4, the functions α and γ should work in tandem to form a Galois-connection. If these functions would not form a Galois connection, the translations would mismatch, the approach breaks: the abstract computation might predict impossible computations.

The functions do form a Galois-connection, as they fullfill monotonicity and soundness:

Lemma 1. *α (over sets) is monotone:*

$$\begin{array}{rcll} X \subseteq Y & \Rightarrow & \alpha(X) \subseteq \alpha(Y) & \\ & \Rightarrow & X \subseteq Y & \textit{(As } \alpha(X) = X\textit{)} \end{array}$$

Lemma 2. *γ is monotone:*

$$\begin{array}{rcll} X \subseteq Y & \Rightarrow & \gamma(X) \subseteq \gamma(Y) & \\ & \Rightarrow & unfold^*(X) \subseteq unfold^*(Y) & \textit{(Unfold does not change the contents)} \end{array}$$

Lemma 3. *α and γ are sound:*

$$\begin{array}{rcll} & n \in \gamma(\alpha(n)) & & \\ = & n \in \gamma(\{n\}) & \textit{(Definition of } \alpha\textit{)} & \\ = & n \in unfold^*\{n\} & (\texttt{n}\textit{ is a concrete value, thus can not be unfolded)} & \\ = & n \in \{n\} & & \end{array}$$

and

$$\begin{array}{rcll} & M \subseteq \alpha(\gamma(M)) & \textit{(α over a set returns that set)} & \\ = & M \subseteq \gamma(M) & \textit{(definition of } \gamma\textit{)} & \\ = & M \subseteq unfold^*(M) & \textit{(Unfolding a set representation has no influence on the represented set)} & \\ = & M \subseteq M & & \end{array}$$

This implies that the set representation actually

1.5.2 Calculating a possible pattern match

Pattern matching can be interpreted in the domain of sets too - an important step in lifting metafunctions to the abstract domain. As pattern matching

against a given pattern can be seen as a function from a parsetree to a variable store, the framework of abstract interpretation can be used to deduce the behaviour of pattern matching over sets.

There are three kinds of patterns that are considered:

- A pattern assigning to a variable
- A pattern comparing to a concrete value
- A pattern deconstructing the parse tree

In the following paragraphs, the behaviour of each of these patterns is explored in isolation. The symbol `~` is used to denote the pattern matching function, which takes a pattern and a parsetree to construct a variable store. Variable stores in the concrete domain will be denoted either `{Variable --> Parsetree}`, whereas variable stores in the abstract domain are denoted `{Variable --> {a, b, c}}` or `{Variable --> SomeSet}`.

As a variable store can be seen as a function from the variable name to the value, abstraction and concretization can be freely applied on the store, where $\alpha(\{$ `{ Var --> a }` , `{ Var --> b }` $\}) =$ `{ Var -->` α`a, b }` .

The second aspect of pattern matching is the aspect that a match can *fail*. This is modeled by, apart from the variable store, returning a set which *might* match. This function is denoted by `expr ~? Set`

Variable assignment

The first pattern to consider is variable assignment:

```
1 Var ~ Set
```

As each variable in the set could be applied in some concrete computation, the intuition for the abstract computation is that the entire set is applied to the variable. This intuition is correct and can be formalized with the general formula of abstract interpretation, namely α`(map(f,` γ`(set)))`, where `f` is the pattern match against `Var`:

$$
\begin{aligned}
& \alpha(map(\ Var, \gamma(Set))) \\
= \ & \alpha(map(\ Var, unfold^*(Set))) \\
= \ & \alpha(map(\ Var, ..., a, b, c, ...)) \\
= \ & \alpha(..., \{\ \texttt{Var --> a}\ \}, \{\ \texttt{Var --> b}\ \}, \{\ \texttt{Var --> c}\ \}, ...\}) \\
= \ & \{\ \texttt{Var --> ..., a, b, c, ...}\ \} \\
= \ & \{\ \texttt{Var --> Set}\ \}
\end{aligned}
$$

Assignment to a variable can never fail, so the matching set is exactly the input set: `Var ~? Set = Set`.

The behaviour of the pattern store when the variable `Var` is matched twice is handled by the deconstruction pattern, as it is the responsibility of the deconstruction to merge the variable store.

Concrete value

The second pattern is matching against a literal:

```
1 "Literal" ~ Set
```

This pattern does not add variables to the store, but only checks if the match might hold, returning an empty store for the match and a fail-message if the concrete literal is not the same.

The behaviour over sets is thus as following, and mainly driven by the question of `"Literal"` is an element of the set or not:

$$
\begin{aligned}
& \alpha(map(\ \texttt{"Literal"}\ , \gamma(Set))) \\
= \ & \alpha(map(\ \texttt{"Literal"}\ , unfold^*(Set))) \\
= \ & \begin{cases} \alpha(map(\ \texttt{"Literal"}\ , \{..., a, b, c, \ \texttt{"Literal"}\ ...\})) & \texttt{"Literal"} \text{ is member of the set} \\ \alpha(map(\ \texttt{"Literal"}\ , \{..., a, b, c, ...\})) & \texttt{"Literal"} \text{ is not member of the set} \end{cases} \\
= \ & \begin{cases} \alpha(\{..., FAIL, FAIL, FAIL, \{\}...\})) & \texttt{"Literal"} \text{ is member of the set} \\ \alpha(\{..., FAIL, FAIL, FAIL, ...\})) & \texttt{"Literal"} \text{ is not member of the set} \end{cases} \\
= \ & \begin{cases} \{\} & \texttt{"Literal"} \text{ is member of the set} \\ FAIL & \texttt{"Literal"} \text{ is not member of the set} \end{cases}
\end{aligned}
$$

In other words, pattern matching a set is possible if this literal is an element of the set. If not, the match fails. The matching set is thus: `"Literal" ~ Set = {"Literal"} ∩Set`

Deconstructing pattern

The last fundamental pattern is a sequence deconstructing a parsetree:

```
1 x y ~ Set
```

The first aspect is whether the sequence might match. Intuitively, only sets containing an instance of the sequence could match the deconstruction. As each element of the deconstruction could be a subpattern, not every instance of the sequence will match, but only the instances having the right subparts. Luckily, the instances matching is exactly the cartesian product of the matching subparts.

$$
\begin{aligned}
& \alpha(map(\ ?xy, \gamma(Set))) \\
= \ & \alpha(map(\ ?xy, \{..., x' \times y', b, c, ...\})) \\
= \ & \alpha(\{..., xy\ ?x1y1, xy\ ?x1y2, xy\ ?x2y1, ..., xy\ ?b, xy\ ?c, ...\}) \\
= \ & \alpha(\{..., xy\ ?x1y2, xy\ ?x1y2, xy\ ?x2y1, ..., FAIL, FAIL, ...\}) \\
= \ & \alpha(\{..., (x\ ?x1 \times y\ ?y1), (x\ ?x1 \times y\ ?y2), (x\ ?x2 \times y\ ?y1), ...\}) \\
= \ & \alpha((x\ ?x1 + x\ ?x2 + ...) \times (y\ ?y1 + y\ ?y2 + ...)) \\
= \ & (x\ ?x') \times (y\ ?y')
\end{aligned}
$$

This suggests a very practical algorithm for abstract pattern matching: - As sequences are supposed to be unique (section 1.2.5), this implies that only a single subset should be considered. - For these sequences, pattern match the parts and combine their results

The store generated by the deconstructing pattern is the sum of all the stores. There is one special case that should be handled, namely the case of a single variable that is declared on multiple places. In the concrete domain, the merge of the variable store checks that the value is the same in both stores, failing otherwise. This implies that, in the abstract domain, it should be possible that both contain the same value; in other words, there should be a common subset which the variable might be.

$$\begin{aligned} & merge(\ \{\ \texttt{T --> Set1}\ \}\ ,\ \{\ \texttt{T --> Set2}\ \}\ ) \\ = \quad & \{\ \texttt{T --> Set1} \cap \texttt{Set2}\ \} \end{aligned}$$

If the intersection is empty, this means that no concrete values could ever match this pattern, indicitating a dead clause. This also indicates that the earlier calculation of the matchin set (`x ~ x' ×y ~ y'`) is too simplistic, but working out the full details would lead to far. A simpler way to calculate the matching set is to use the resulting variable store, and backfill the pattern as if it was an expression; this is introduced in the next section.

Multiple arguments

Functions with multiple arguments can be handled just as sequences are. Consider a function taking two arguments `f : a -> b -> c`. The two arguments can be considered a new syntactic form, `arg0 ::= a b`, underpinning the our intuition.

In other words, all functions can be considered functions with one input argument and one output argument.

1.5.3 Calculating possible return values of an expression

Given set a variable might be, the set representation of a function can easily be calculated.. 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 in the expression
- A function call is replaced by the syntactic form it might return. Typically, the signature is used to deduce this the return type, but some algorithms use more accurate set representations
- The set produced by a sequence is the cartesian product of the parts.

This gives us all the tools needed to lift single-clause functions to functions over sets.

1.5.4 Combining clauses

Combining multiple clauses is the last step in lifting arbitrary functions to functions over sets.

For each clause, the output of that clause can be calculated if the input set for that clause is known. The trick lies in the calculation of what each clause might get as input. Given the input set `I` for the function, the first clause will receive this entire input and matches a subset `I1` of the input. The second clause will receive the nonmatching part, thus `I - I1`, a gain matching a subset. This way, the input for each clause can be calculation, together with the subset of the input *not* matching any clause of the function, giving the **totality check** for free. Furhtermore, if a single clause never matches an input, a **dead clause** is detected, giving another check for free.

```
1 f       : a -> b
2 # Input I
3 f(x:x)  = ...   # Matches I1
```

```
# Input I - I1
f(y:y)  = ...
# Input I - I2
...
# I - In is fallthrough and will never match
```

This gives all the pieces needed to lift functions over parsetrees into functions over sets, in a practical and computable way.

Per example, if one would like to know the result of the domain function over the set `{"Bool", "Bool" "->" type, ("Int") "->" "Bool"}`, this can be calculated. As reminder, the definition of `domain` can be found in figure 1.11). The actual calculation of the function happens clause per clause:

- Clause 1 matches nothing of the set, no element of the set has parentheses
- Clause 2 matches `("Int") "->" "Bool"`, returning `"Int"`, leaving `{"Bool", "Bool" "->" type}` for the next clause
- Clause 3 matches `"Bool" "->" type`, returning "Bool"
- `"Bool"` is never matched and falls through, giving no result

The sum of the calculation is `{"Bool", "Int"}` which can be refolded to `{baseType}'`.

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

Figure 1.11: Definition of the domain function, as introduced in chapter **??**.

1.6 Conclusion

The techniques presented allow metafunctions to be lifted to functions over sets automatically. The framework for abstract interpretation gives a mathematical underpinning, whereas the compact representation and efficient operations ensure the practicallity and computablitiy, even in the face of infinite sets.

Not only is this a crucial step in the gradualization of the target language, quite some checks can be implemented warning the programmer for missed cases. Lifting functions to functions over sets is thus a powerfull technique that can be applied broadly.

Bibliography

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