

# Contents

<b>1</b>	<b>Syntax Driven Abstract Interpretation</b>	<b>3</b>
1.1	Abstract interpretation . . . . .	3
1.1.1	The rule of signs . . . . .	4
1.1.2	Ranges as abstract domain . . . . .	5
1.1.3	Collecting semantics . . . . .	6
1.1.4	Properties of $\alpha$ and $\gamma$ . . . . .	7
1.2	Properties of the syntax . . . . .	9
1.2.1	Syntactic forms as sets . . . . .	9
1.2.2	Embedded syntactic forms . . . . .	9
1.2.3	Empty sets . . . . .	9
1.2.4	Left recursive grammars . . . . .	11
1.2.5	Uniqueness of sequences . . . . .	13
1.3	Representing sets of values . . . . .	13
1.3.1	Representing sets . . . . .	14
1.3.2	Sets with concrete values . . . . .	14
1.3.3	Symbolic sets . . . . .	14
1.3.4	Infinite sets . . . . .	15
1.3.5	Set representation of a syntax . . . . .	15
1.3.6	Defining $\alpha$ and $\gamma$ . . . . .	15
1.4	Operations on representations . . . . .	16
1.4.1	Unfolding . . . . .	16
1.4.2	Directed unfolds . . . . .	17
1.4.3	Refolding . . . . .	17
1.4.4	Resolving to a syntactic form . . . . .	18
1.4.5	Addition of sets . . . . .	18
1.4.6	Subtraction of sets . . . . .	18
1.4.7	Subtraction of a sequence from a symbolic value . . . . .	19
1.5	Algorithms using abstract interpretation . . . . .	21
1.5.1	Calculating a possible pattern match . . . . .	21
1.5.2	Calculating possible return values of an expression . . . . .	23
1.5.3	Calculating the collecting function . . . . .	23
1.5.4	Calculating the function domain . . . . .	24
1.5.5	Calculating the codomain . . . . .	25



# Chapter 1

## Syntax Driven Abstract Interpretation

In this section, functions on parsetrees are converted into functions over sets of parsetrees. This is useful to algorithmically analyze these functions, which will help to gradualize them. This section dissects the technique to do so -abstract interpretation- and is organized as following:

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

### 1.1 Abstract interpretation

Per Rice's theorem, it is generally impossible to make precise statements about all programs. However, making useful statements about some programs is feasible. Cousot (1977) introduces a framework to do so, named **abstract interpretation**: "A program denotes computations in some domain of objects. Abstract interpretation of programs consists in using that denotation to describe computations in *another domain of abstract objects*, so that the results of the abstract computations give some information on the actual computation".

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

This principle can best be illustrated, for which the 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

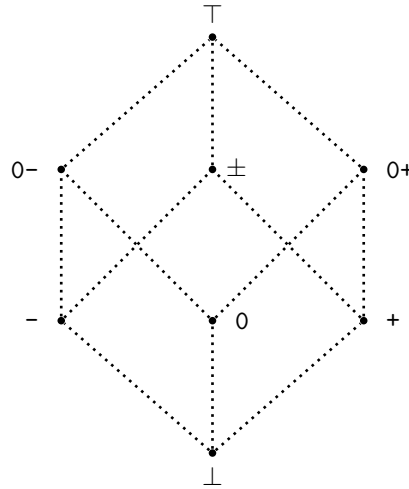
### 1.1.1 The rule of signs

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 extra 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 intuitively 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  $\gamma$ , which translates from the abstract domain to the concrete domain:

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

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

$$\alpha(n) = \begin{cases} - & \text{if } n < 0 \\ 0 & \text{if } n = 0 \\ + & \text{if } n > 0 \end{cases}$$

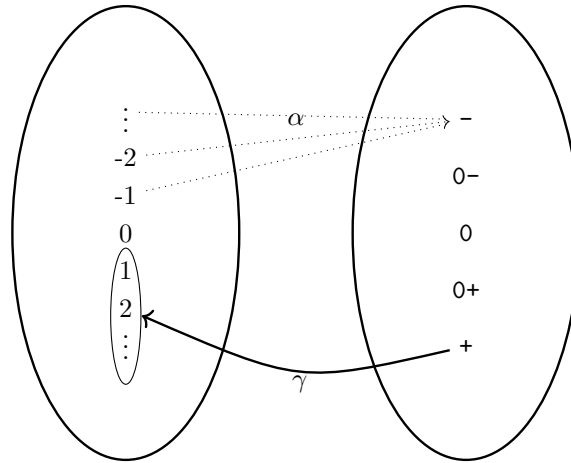


Figure 1.3: Concretization and abstraction between integers and signs

### 1.1.2 Ranges as abstract domain

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

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

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

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

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

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

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

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

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

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

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

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

### 1.1.3 Collecting semantics

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

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

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

$$\begin{aligned}
\alpha(n) &= \{n\} \\
\gamma(\{n_1, n_2, \dots\}) &= \{n_1, n_2, \dots\}
\end{aligned}$$

Performing the computation in the abstract domain of sets can be more efficient than the equivalent concrete computations, as the structure of the concrete domain can be exploited to use a more efficient representation in memory (such as ranges). Furthermore, different input might turn out to have the same result

halfway in the calculation, such as  $\mathbf{f} \ x = \mathbf{abs}(x) + 1$  with input  $\{+1, -1\}$  which becomes  $\{1\} + 1$ . This state merging might result in additional speed increases.

Using this abstract domain effectively lifts a function over integers into a function over sets of integers. Exactly this abstract domain is used to lift the functions over parsetrees into functions over sets of parsetrees. To perform these calculations, an efficient representations of possible parsetrees will be deduced later in this section, in chapter 1.3.

#### 1.1.4 Properties of $\alpha$ and $\gamma$

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

##### Monotonicity of $\alpha$ and $\gamma$

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.

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

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

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

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

##### Soundness

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

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

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

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

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

Without these properties tying  $\alpha$  and  $\gamma$  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  $\alpha$  maps 0 onto + yields following results:

$$\alpha(n) = \begin{cases} - & \text{if } n < 0 \\ + & \text{if } n = 0 \\ + & \text{if } n > 0 \end{cases}$$

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

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

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

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

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

### Galois connection

Together,  $\alpha$  and  $\gamma$  form a monotone *Galois connection*, as it obeys its central property:

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

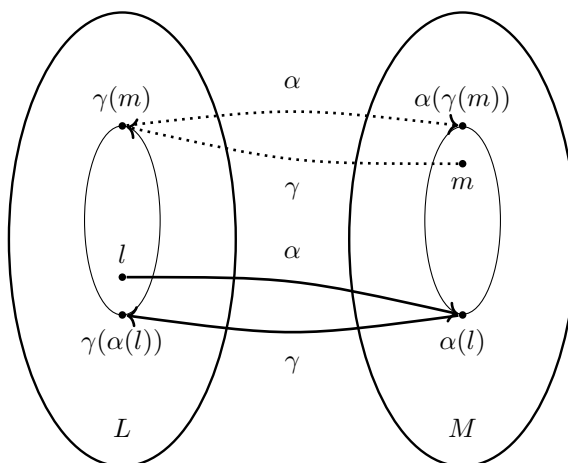


Figure 1.4: Galois-connection, visualized



## 1.2 Properties of the syntax

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

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

### 1.2.1 Syntactic forms as sets

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

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

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

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

### 1.2.2 Embedded syntactic forms

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

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

```

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

```

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

This supertype relationship is a lattice - the absence of left recursion implies that no cycles can exist in this supertype relationship. This lattice can be visualized, as in figure 1.6.

### 1.2.3 Empty sets

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

As example, consider following syntax:

```

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

```

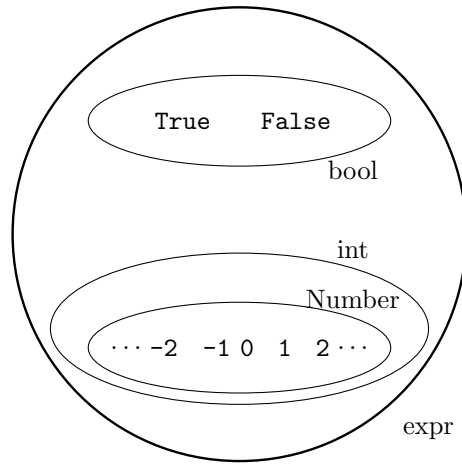


Figure 1.5: Nested syntactic forms

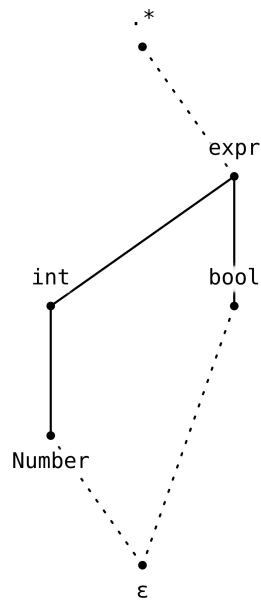


Figure 1.6: A simple subtyping relationship

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

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

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

While a syntactic form with no choices is a syntax error within ALGT, it is

possible to define an empty form through recursion:

```

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

```

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

### 1.2.4 Left recursive grammars

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 grammars where syntactic forms are recursive. An example would be types:

```

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

```

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

```

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

```

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

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

We can easily detect this left recursion algorithmically. The algorithm itself can be found in figure ???. To make this algorithm more tangible, consider following syntax:

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

1  c      ::= d
2  d      ::= c

```

At this point, no rules can be removed anymore. Only rules containing left recursion remain, for which an error message can be generated (lines 24 till 26).

```

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

```

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

### 1.2.5 Uniqueness of sequences

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

```

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

```

A parsetree containing "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 refactor this, without making things overly complicated. Instead, runtime annotations are used to keep track of which form originated a parsetree.

The uniqueness-constraint is merely added to keep things simpler and force the language designer to write a language with as little duplication as possible. Furthermore, it helps the typechecker to work more efficient and with less errors.

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

## 1.3 Representing sets of values

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

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

### 1.3.1 Representing sets

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

```
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 succinct 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`:

```

1 type ::= { baseType, "(" type ")" }
2       = { "Bool", "Int", "(" typeTerm "->" type ")" , "(" typeTerm ")" }
3       = { "Bool", "Int", "(" "Bool" ")", "(" "Int" ")", ...
4       = ...

```

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.

```

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

```

becomes

```

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

```

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

### 1.3.6 Defining $\alpha$ and $\gamma$

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

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

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

**Lemma 1.**  $\alpha$  (over sets) is monotone:

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

**Lemma 2.**  $\gamma$  is monotone:

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

**Lemma 3.**  $\alpha$  and  $\gamma$  are sound:

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

and

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

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

## 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:



```

1 {baseType}
2 == { {"Bool", "Int"} }
3 == { "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:

```

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

```

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

```

1 { "Bool" "->" "Bool"
2   , "Bool" "->" "Int"
3   , "Int" "->" "Bool"
4   , "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.

```

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

```

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

```

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

```

... and unfold these recursively:

```

1 "Bool" "->" baseType
2 "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 whether this is contained in it.

### 1.4.4 Resolving to a syntactic form

Given a set, it can be useful 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 useful 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 enables a lot of useful 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 whether 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 whether 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 subtract 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 whether 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  $((" \text{ type } ")) - ((" \text{ (\"Bool\" \"->\" type) } "))$ . It is intuitively clear that the parentheses  $($  and  $)$  should remain, and that we want to subtract  $\text{type} - (\text{\"Bool\" \"->\" type})$ .
2. Here we subtract a sequence again. We unfold  $\text{type}$ , to yield: <sup>1</sup>  
 $\{\text{typeTerm}, \text{typeTerm \"->\" type}\} - (\text{\"Bool\" \"->\" type})$
3. In the set, we now have two values we should subtract:
  - $\text{typeTerm} - (\text{\"Bool\" \"->\" type})$  yields  $\text{typeTerm}$ , as neither shadows the other.
  - $(\text{typeTerm \"->\" type}) - (\text{\"Bool\" \"->\" type})$  can be aligned. It is clear we'll want to have  $\text{typeTerm} - \text{\"Bool\"}$  and  $\text{type} - \text{type}$  and let  $\text{\"->\"}$  unchanged.
4.  $\text{typeTerm} - \text{\"Bool\"}$  gives us  $\{\text{baseType}, (" \text{ type } ") - \text{\"Bool\"}\}$ , resulting in  $\{\text{Int}, (" \text{ type } ")$ . It is important to note that  $(\text{Bool})$  is *still* an element of this set, despite  $\text{\"Bool\"}$  without parentheses is not. Despite having the same semantics, these are two syntactically different forms!
5.  $\text{type} - \text{type}$  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  $(\text{typeTerm \"->\" type}) - (\text{\"Bool\" \"->\" type})$  together. Intuitively, we'd expect it to be  $\{\text{Int \"->\" type}, (" \text{ type } ") \text{ \"->\" type}\}$ . More formally, the directed unfold of  $\text{typeTerm \"->\" type}$  is  $\{\text{\"Bool\" \"->\" type}, \text{Int \"->\" type}, (" \text{ type } ") \text{ \"->\" type}\}$ . Subtracting  $\text{\"Bool\" \"->\" type}$  yields our former result.
6. We put together the results of the recursive calls, being:
  - $\{\text{typeTerm}\}$
  - $\{\text{Int \"->\" type}, (" \text{ type } ") \text{ \"->\" type}\}$
 This results in  $\{\text{Int \"->\" type}, (" \text{ type } ") \text{ \"->\" type}, \text{typeTerm}\}$
7. We unfolded  $\text{type}$  to subtract the sequence, yielding  
 $\{\text{Int \"->\" type}, (" \text{ type } ") \text{ \"->\" type}, \text{typeTerm}\}$
8. We put the parentheses back around each expression:  
 $\{(" \text{ (\"Int\" \"->\" type) } ), (" \text{ (\" type )\" \"->\" type } ), (" \text{ typeTerm } ") \}$ ,  
 giving the desired result.

---

<sup>1</sup>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  $a_1, a_2, a_3, \dots$  and  $b$  unfolds to  $b_1, b_2, b_3, \dots$ . This means that  $a \ b$  unfolds to  $a_1 \ b_1, a_1 \ b_2, a_1 \ b_3, \dots, a_2 \ b_1, a_2 \ b_2, \dots$ .

Now, if we would subtract sequence  $a_1 \ b_1$  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<sup>2</sup> and calculate the difference:

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

Now, the resulting sequences are  $\{(a - a_1) \ b, a \ (b - b_1)\}$ .

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

```
1 | a b c d ... - a1 b1 c1 d1 ...
2 | = { (a - a1) b c d ...
3 |   , a (b - b1) c d ...
4 |   , a b (c - c1) d ...
5 |   , a b c (d - d1) ...
6 | }
```

We apply this on the example  $(\text{typeTerm} \rightarrow \text{type}) - (\text{Bool} \rightarrow \text{type})$ , yielding following pointwise differences:

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

Resulting in:

```
1 | { (typeTerm - "Bool") "→" type
2 |   , typeTerm ("→" - "→") type
3 |   , typeTerm "→" (type - type)}
4 | = {"Int", "(" type ")"} "→" type
5 |   , typeTerm {} type
6 |   , typeTerm "→" {}
7 | = {"Int", "(" type ")"} "→" type
8 | = {"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:

```
1 | {type} ~ (T1 "→" T2)
```

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

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

<sup>2</sup>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  $\tau$ , we update the store  $\sigma$  with  $T \rightarrow S$ .

If the variable  $\tau$  were already present in the store  $\sigma$ , 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  $\{\text{type}\}, \{\text{baseType}\}$  against  $(\tau, \tau)$ , which would yield  $\sigma = \{T \rightarrow \{\text{baseType}\}\}$ .

### Concrete parsetree

When a set representation is compared against a concrete parsetree, we check whether 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  $\{\text{type}\} \sim (T1 \text{ "->" } T2)$ .

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

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

### 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}`, thus `type`.

### 1.5.4 Calculating the function domain

#### Single argument functions

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

```
1 { "(" 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:

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

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

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

#### Multiple argument functions

Consider the function `equals`

```
1 equals : basetype -> basetype -> basetype
2 equals("Bool", "Bool") = "Bool"
3 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:

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

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

```
1 [{"Int", baseType}, [baseType, "Int"]] - ["Int", "Int"]
2 = { ["Int", baseType] - ["Int", "Int"]
3   , [baseType, "Int"] - ["Int", "Int"] }
4 = { ["Int" - "Int", baseType]
5   , ["Int", baseType - "Int"]
6   , [baseType - "Int", "Int"]
7   , [baseType, "Int" - "Int"] }
8 = { [ {} , baseType]
9   , ["Int", "Bool"]
10  , ["Bool", "Int"]
11  , [baseType, {} ] }
12 = { ["Int", "Bool"]
13  , ["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 algorithm is practical for small inputs, but can become slow in the presence of large types; the sequence set might contain up to  $O(\#Args * \#TypeSize)$  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 useful checks for functions, namely a **pattern match totality checker** and a **clause livability 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  $\text{dom} \rightarrow \{\text{type}\}$ ), we initialize it with empty sets ( $\text{dom} \rightarrow \{\}$ ). 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  $\text{dom} \rightarrow \{\text{baseType}\}$  and rerun.

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

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

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

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

```

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

```

we would yield:

```

1 | {f → {}}

```

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

```

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

```