2.2. Denotational Semantics of Haskell

2.2.1 Precise definition of the domain $\text{Dom}$ (includes values for expressions of user-defined types, polymorphic types, ...)

2.2.2 Denotational Semantics of Simple Haskell
($\approx$ Haskell without Pattern Matching)

2.2.3 Auto-Transformation from Complex to Simple Haskell (used for definition of semantics and for the implementation of Haskell)

2.2.1 Construction of Domains

We need two operations to extend domains by new objects: lift and coalesced sum.

**Def 2.2.1 (lift)**

Let $D$ be a set with a relation $\leq_D$. The lift of $D$ is $D_\perp = \{ d^D \mid d \in D \} \cup \{ \bot_D \}$. The relation $\leq_D$ is defined as follows: $e \leq_D e'$ if $e = \bot_D$ or $e = d^D$, $e' = d'^D$, and $d \leq d'$.

If $D$ is a domain, then $D_\perp$ is also a domain.
We often write "$d$ in $D_\perp$" instead of $d^D$. 
So "d in D₁" is a labeled version of d that occurs in D₁.  

E.g.: \( \mathbb{Z}_1 \) is the lift of \( \mathbb{Z} \)

1. Domains for Base Types
   So the lift is used to create the flat domains for the base types Int, Bool, Char, Float, ...

2. Domains for Tuple Types
   Now we explain how to create the domains for tuple types. If \( D_1, ..., D_n \) are the domains for the types \( \text{tyr}_1, ..., \text{tyr}_n \), then \( (D_1 \times ... \times D_n)_1 \) is the domain for the type \( (\text{tyr}_1, ..., \text{tyr}_n) \)

\[
(\mathbb{Z}_1 \times \mathbb{Z}_1)_1 \quad \ldots \quad (0,0) \quad \ldots \quad (1,0) \quad \ldots
\]

\[
C :: (\text{Int}, \text{Int})
C = C \quad \text{\( \preceq \) has semantics \( \bot_{(\mathbb{Z}_1 \times \mathbb{Z}_1)} \)}
\]

\[
d :: (\text{Int}, \text{Int})
d = \text{bot} \quad \text{\( \preceq \) has semantics \( C_{\bot_{\mathbb{Z}_1}}, \bot_{\mathbb{Z}_1} \)}
\]

\[
f :: (\text{Int}, \text{Int}) \rightarrow \text{Int}
f = \text{f c does not terminate}
f \ (x, y) = 0 \quad \text{\( \preceq \) \quad f d = 0}
\]
\[
g : (\text{Int}, \text{Int}) \rightarrow \text{Int}
\]
\[
g \circ \cdot = 0
\]
\[
g \circ \cdot = 0
\]

3. **Domains for Function Types**

Domain for function types: If \( D_1 \) is the domain for \( \text{type}_1 \) and \( D_2 \) is the domain for \( \text{type}_2 \), then

\[
\langle D_1 \rightarrow D_2 \rangle
\]

is the domain for the type \( \text{type}_1 \rightarrow \text{type}_2 \).

4. **Domains for User-Defined Types**

Now we want to define domains for user-defined data types (introduced by the keyword “data”).

4a. Just one data constructor, no recursion.

Argument types of the constructor are different from the newly defined type.

\[
data \ \text{tyconstr} = \text{Constr} \ \text{type}_1, \ldots, \text{type}_n
\]

If \( D_1 \) is the domain for \( \text{type}_1 \), \ldots, \( D_n \) is the domain for \( \text{type}_n \), then the domain for \( \text{tyconstr} \) is:

\[
(\{\text{Constr}\} \times D_1 \times \ldots \times D_n)
\]

Variant of the domain for \( (\text{type}_1, \ldots, \text{type}_n) \), where \( \text{Constr} \) is added as an additional component.

Ex: \( \text{data Pair} = \text{Combine} \ \text{Int} \ \text{Int} \)
Domain for the type Pair is \( \{ \text{Const} \} \times \mathbb{Z}_1 \times \mathbb{Z}_1 \) :

\[ 
\begin{array}{c}
\text{Const, 0, 1} \\
\text{Const, 0, 1} \\
\text{Const, 1, 1} \\
\text{Const, 1, 1}
\end{array}
\]

Instead of \((\text{Const, } d_1, \ldots, d_n)\) we also write \(\text{Const} (d_1, \ldots, d_n)\) or \(\text{Const} d_1 \ldots d_n\).

So essentially, the domain consists of all ground terms built from data constructors and \(\perp\) (here, pre-defined types like \(\text{Int}\) have infinitely many data constructors \(0, 1, -1, \ldots\)).

This also works for data constructors of arity 0:

\[
data \text{tyconst} = \text{Const}
\]

Then the domain for \(\text{tyconst}\) is:

\[
\{ \text{Const} \} \perp = \\
\{ \perp, \text{Const} \}
\]

To handle user-defined types with several data constructors, we need another operation on domains.

Def. 222 (Coalesced Sum (versehmolzene Summe))
**Def. 222** (Coalesced Sum (verschmolzene Summe))

Let \( D_1, \ldots, D_n \) be domains. Their coalesced sum is

\[
D_1 \oplus \ldots \oplus D_n = \{ d^D | d \in D_i, d \neq \bot_{D_i} \} \cup \{ d^D | d \in D_i, d = \bot_{D_i} \} \\
\cup \{ \bot_{D_1 \oplus \ldots \oplus D_n} \}
\]

The relation \( \vdash_{D_1 \oplus \ldots \oplus D_n} \) is defined as follows:

\[
e \vdash_{D_1 \oplus \ldots \oplus D_n} e' \iff e = \bot_{D_i} \quad \text{or} \quad e = d^D_i, e' = d'^{D_i}, \quad \text{and} \quad d \vdash_{D_i} d'.
\]

\( D_1 \oplus \ldots \oplus D_n \) is again a domain. We often write "\( d \) in \( D_1 \oplus \ldots \oplus D_n \)" instead of "\( d^D_i \)." So "\( d \) in \( D_1 \oplus \ldots \oplus D_n \)" is a labeled variant of \( d \) that occurs in \( D_1 \oplus \ldots \oplus D_n \).

---

**46** user-defined data structure: several data constructors, no recursion (Slide 36)

**data** ty constr = constr₁ type₁₁ ... type₁ₙ₁ | ... | constrₖ typeₖ₁ ... typeₖₙ

gets the domain \( D_1 \oplus \ldots \oplus D_k \)

where \( D_i = (\{\text{constr}_{i,j}\} \times D_{i,1} \times \ldots \times D_{i,n_i}) \perp, \ldots) \)

\( D_n = (\{\text{constr}_{k,n}\} \times D_{k,1} \times \ldots \times D_{k,n_k}) \perp) \)
and \( D_{ij} \) is the domain for \( \text{type}_{ij} \)

Ex: data Pair = Combine Int Int \( \mid \) Empty

Corresponding domain is:
\[
(\{\text{Combine}\} \times \mathbb{Z}_1 \times \mathbb{Z}_1) \uplus \{\text{Empty}\} = \\
\{\bot, \text{Empty}, (\text{Combine}, 1, 1), (\text{Combine}, 0, 1), \ldots\} \\
\ldots \text{Combine} \ldots \\
\ldots \text{Combine} \bot \ \text{Combine} \bot \\
\downarrow \\
\text{Combine} \bot \bot \ \text{Empty} \\
\uplus
\]

4c) several data constructors, recursion (but still no polymorphism)

data Nats = Zero \( \mid \) Succ Nats

If \( D_{\text{Nats}} \) is the domain for Nats, then
\[
D_{\text{Nats}} = \{\text{Zero}\} \uplus (\{\text{Succ}\} \times D_{\text{Nats}}) \uplus \{\text{Empty}\}
\]  

(8)

Problem: \( D_{\text{Nats}} \) occurs on the rhs of its own definition

Solution: Define \( D_{\text{Nats}} \) as the smallest domain that satisfies the constraint (8).

Alternatively, \( D_{\text{Nats}} \) is the \( \text{LFP} \) of the function \( \text{id} \) on domains, where
\[ \text{F} = \{ \text{Zero} \}_\bot \oplus (\{ \text{Succ} \} \times \text{D}) \_ \]

Domains are sets where \( E_D \) is a cpo. The smallest domain is \( \{ \bot \} \).

\[ \begin{align*}
\text{dd}^0(\{ \bot \}) &= \{ \bot \} \\
\text{dd}^1(\{ \bot \}) &= \{ \text{Zero} \}_\bot \oplus (\{ \text{Succ} \} \times \{ \bot \}) \_ = \{ \bot, \text{Zero}, (\text{Succ}, \bot) \} \\
\text{dd}^2(\{ \bot \}) &= \{ \text{Zero} \}_\bot \oplus (\{ \text{Succ} \} \times \{ \bot, \text{Zero}, \text{Succ,} \bot \}) \_ = \{ \bot, \text{Zero}, (\text{Succ}, \bot), (\text{Succ,} \text{Zero}) \} \\
&\vdots \\
\text{Succ} \_ \bot \\
\text{Succ(Succ Zero)} \\
\text{Succ(Succ(Succ \bot))} \\
\text{Succ Zero} \\
\text{Succ(Succ \bot)} \\
\text{Succ \bot} \\
\text{Zero} \\
\bot
\end{align*} \]

\[ \text{let } x = \text{Succ } x \text{ in } x \]

has the semantics \( \text{Succ} \_ \bot \).

\[ \textcolor{red}{\text{4d) user-defined types with polymorphism}} \]

If type is a polymorphic type (with type variables) and an object has this type, then this object also has all types that result from instantiating the type variables of type.

\[ \Rightarrow \text{Domain for type is obtained by taking the intersection of the domains for all instantiations of type.} \]
Ex: Consider the polymorphic type \([a :\]\).
If an objection has the type \([a :\]\) then it also has the types \([\text{Int}, \; \text{Bool}, \; \text{Int} \to \text{Int}]\), ....

E.g.: \([1] : \; [a :\]\), but \([1]\) can also be used as a list of type \([\text{Int}, \; \text{Bool}]\), ....

The domain for \([a :\]\) is the intersection of the domains for \([\text{Int}, \; \text{Bool}]\), ....

To simplify the definitions, we won't consider pre-defined lists in the following, but we have to define lists ourselves:

```haskell
data List a = Nil | Cons a (List a)
```

The domain \(D_{\text{List Nats}}\) for the type \(\text{List Nats}\) is the smallest domain that satisfies:

\[
D_{\text{List Nats}} = \{ \text{Nil} \} \perp \bigoplus (\{ \text{Cons} \} \times D_{\text{Nats}} \times D_{\text{List Nats}}) \perp
\]

\[
\begin{array}{ccccccc}
\text{Cons} & \text{Cons} & \text{Cons} & \text{Cons} & \text{Cons} & \text{Cons} & \text{Cons} \\
\text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\
\end{array}
\]

\[
D_{\text{List a}} = \bigcap \quad D_{\text{List type}} = \{ \perp, \text{Nil}, \text{Cons} \perp, \text{Cons} \uplus \text{Nil}, \text{Cons} \perp, \text{Cons} \uplus \text{Nil} \}
\]

- type is
  - a Haskell-type

FP 2016 Seite 8
To summarize: Domain for a user-defined type contains all terms built from data constructors and \( \perp \), where terms can also be infinite.

Now, we can define a full domain \( \text{Dom} \) for a Haskell program that contains the sub-domains for all different types. To ease presentation, we exclude type synonyms, type classes, and pre-defined lists.

**Def 2.2.3 (Domain of a Program)**

For a Haskell program as above, let \( \text{Con}_{n} \) be the set of all \( n \)-ary data constructors (where \( \mathbb{N}, \mathbb{B}, \mathbb{C}, \mathbb{F} \) are regarded as data constructors of arity 0). Then \( \text{Dom} \) is the smallest domain that satisfies the following equation:

\[
\text{Dom} = \text{Functions} \oplus \text{Tuples}_0 \oplus \text{Tuples}_2 \oplus \text{Tuples}_3 \oplus \ldots \oplus \text{Constructions}_0 \oplus \text{Constructions}_2 \oplus \ldots
\]

where

- \( \text{Functions} = \langle \text{Dom} \to \text{Dom} \rangle \)
- \( \text{Tuples}_0 = \{ () \} \)
- \( \text{Tuples}_n = (\overbrace{\text{Dom} \times \ldots \times \text{Dom}}^{\text{n times}}) \) for \( n \geq 2 \)
- \( \text{Constructions}_n = (\overbrace{\text{Con}_n \times \text{Dom} \times \ldots \times \text{Dom}}^{\text{n times}}) \) for \( n \geq 0 \)

\( \text{Dom} \) also contains elements like \((\text{Succ}, \text{True})\) in \( \text{Con}_n \).
that are not needed for the semantics of well-typed Haskell-expressions (but that’s no problem).

Don depends on the names of the data constructors of the actual Haskell program.