2.2.1 Construction of Domains

Mittwoch, 25. Mai 2016 11:00

2.2. Denotational Semantics of Haskell

2.2.1 Precise definition of the domain 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 Aut. 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.

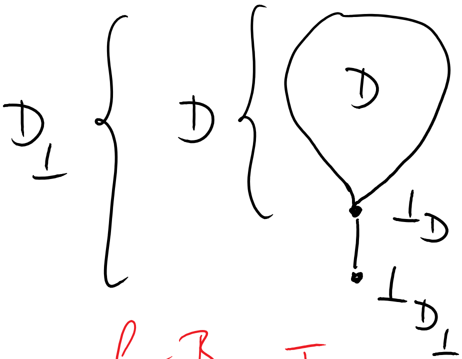
Def 2.2.1 (Lift)

Let D be a set with a relation $\sqsubseteq_D$. The *lift* of D is $D_\perp = \{ d^D \mid d \in D\} \cup \{\perp_{D_\perp}\}$. The relation $\sqsubseteq_{D_\perp}$ is defined as follows: $e \sqsubseteq_{D_\perp} e'$ iff $e = \perp_{D_\perp}$ or $e = d^D$, $e' = d'^D$, and $d \sqsubseteq_D d'$.

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 $D_\perp$" is a labeled version of d that occurs in $D_\perp$.



E.g.: $\mathbb{Z}_\perp$ is the lift of $\mathbb{Z}$

$\mathbb{B}_\perp$ $\quad$ $\mathbb{B}$

$C_\perp$ $\quad$ C

$F_\perp$ $\quad$ F

1) Domains for Base Types

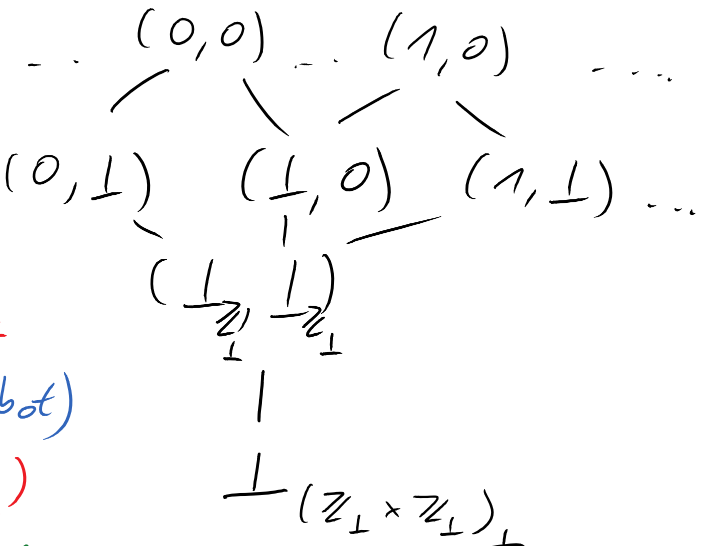
So the lift is used to create the flat domains for the base types Int, Bool, Char, Float, ...

2) Domains for Tuple Types

Now we explain how to create the domains for tuple types.

If $D_1, \dots, D_n$ are the domains for the types $\underline{type}_1, \dots, \underline{type}_n$, then $(D_1 \times \dots \times D_n)_\perp$ is the domain for the type $(\underline{type}_1, \dots, \underline{type}_n)$

$(\mathbb{Z}_\perp \times \mathbb{Z}_\perp)_\perp$




```
c :: (Int,Int)
c = c
```
← has semantics $\perp_{(\mathbb{Z}_\perp \times \mathbb{Z}_\perp)_\perp}$

```
d :: (Int,Int)
d = let bot = bot in (bot, bot)
```
← has semantics $(\perp_{\mathbb{Z}_\perp}, \perp_{\mathbb{Z}_\perp})$

```
f :: (Int,Int) -> Int
f (x,y) = 0
```

f c does not terminate

f d = 0

$g :: (Int, Int) \to Int$
$g\ z = 0$

$g\ c = 0$
$g\ d = 0$

(3) Domains for Function Types

Domain for <u>function types</u>: If D_1 is the domain for <u>$type_1$</u> and D_2 is the domain for <u>$type_2$</u>, then

$\langle D_1 \to D_2 \rangle$ is the domain for the type <u>$type_1$</u> $\to$ <u>$type_2$</u>.

(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 <u>tyconstr</u> = <u>Constr</u> <u>$type_1$</u> ... <u>$type_n$</u>

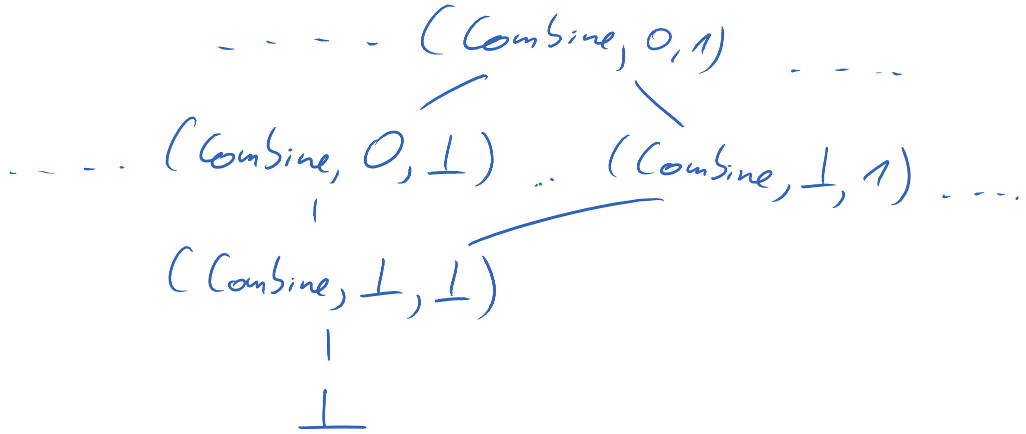
If D_1 is the domain for <u>$type_1$</u>, ..., D_n is the domain for <u>$type_n$</u>, then the domain for <u>tyconstr</u> is:

$(\{\underline{Constr}\} \times D_1 \times \ldots \times D_n)_\perp$

Variant of the domain for (<u>$type_1$</u>, ..., <u>$type_n$</u>), where <u>Constr</u> is added as an additional component

Ex: data Pair = Combine Int Int

Domain for the type Pair is $(\{\underline{Combine}\} \times \mathbb{Z}_\perp \times \mathbb{Z}_\perp)_\perp$:



Instead of $(\underline{constr}, d_1, \dots, d_n)$ we also write
$\underline{Constr}(d_1, \dots, d_n)$ or $\underline{Constr}\ d_1 \dots d_n$.

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

This also works for data constructors of arity 0:

$$\text{data } \underline{tyconstr} = \underline{constr}$$

Then the domain for $\underline{tyconstr}$ is: $\{\underline{constr}\}_\perp =$
$$\{\perp, \underline{constr}\}.$$

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

Def. $\underline{222}$ (Coalesced Sum (verschmolzene Summe))

Def. 222 (Coalesced Sum (verschmolzene Summe))

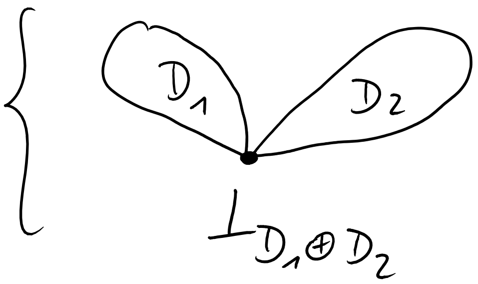
Let $D_1, \ldots, D_n$ be domains. Their Coalesced sum is

$$D_1 \oplus \ldots \oplus D_n = \{ d^{D_1} \mid d \in D_1, d \neq \perp_{D_1} \} \cup \ldots \cup \{ d^{D_n} \mid d \in D_n, d \neq \perp_{D_n} \}$$
$$\cup \{ \perp_{D_1 \oplus \ldots \oplus D_n} \}.$$

The relation $\sqsubseteq_{D_1 \oplus \ldots \oplus D_n}$ is defined as follows:

$e \sqsubseteq_{D_1 \oplus \ldots \oplus D_n} e'$ iff $e = \perp_{D_1 \oplus \ldots \oplus D_n}$ or $e = d^{D_i}$, $e' = d'^{D_i}$, and $d \sqsubseteq_{D_i} d'$.

$D_1 \oplus \ldots \oplus D_n$ 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 $D_1 \oplus \ldots \oplus D_n$.

$D_1 \oplus D_2$ 

(4b) user-defined data structure: several data constructors, no recursion (Slide 36)

data $\underline{tyconstr}$ = $\underline{constr_1}$ $\underline{type_{1,1}} \ldots \underline{type_{1,n_1}}$ | ... | $\underline{constr_k}$ $\underline{type_{k,1}} \ldots \underline{type_{k,n_k}}$

gets the domain $D_1 \oplus \ldots \oplus D_k$

where $D_1 = (\{\underline{constr_1}\} \times D_{1,1} \times \ldots \times D_{1,n_1})_\perp, \ldots,$

$D_{..} = (\{\underline{constr_k}\} \times D_{k,1} \times \ldots \times D_{k,n_{..}})$

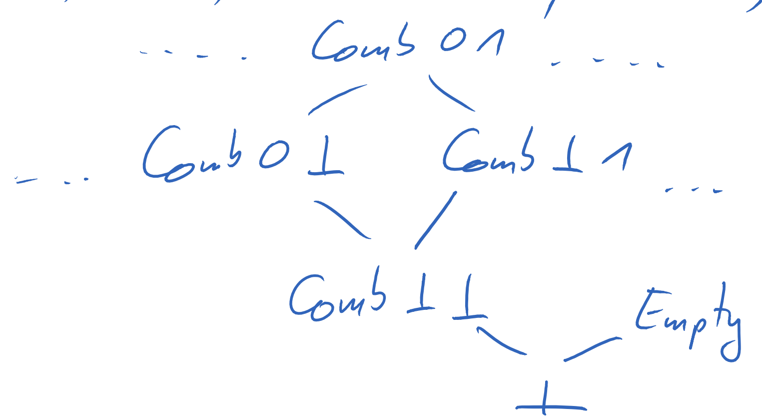
and $D_{i,j}$ is the domain for ~~type~~ $type_{i,j}$

Ex: data Pair = Combine Int Int | Empty

Corresponding domain is:

$$(\{Combine\} \times \mathbb{Z}_\perp \times \mathbb{Z}_\perp)_\perp \oplus \{Empty\}_\perp =$$

$$\{\perp, Empty, (Combine, \perp, \perp), (Combine, 0, \perp), \ldots\}$$



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

data Nats = Zero | Succ Nats

If D_{Nats} is the domain for Nats, then

$$D_{Nats} = \{Zero\}_\perp \oplus (\{Succ\} \times D_{Nats})_\perp \qquad (*)$$

<u>Problem</u>: D_{Nats} occurs on the rhs of its own definition

<u>Solution</u>: Define D_{Nats} as the smallest domain that satisfies the constraint $(*)$.

w.r.t. $\subseteq$, i.e. w.r.t the subset relation

Alternatively, D_{Nats} is the lfp of the function dd on domains, where

$$dd(D) = \{Zero\}_\perp \oplus (\{Succ\} \times D)_\perp$$

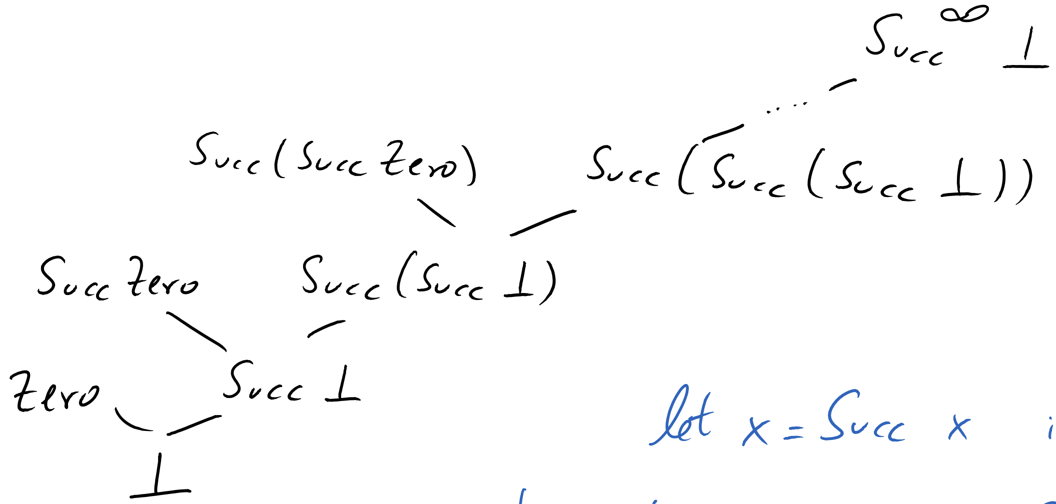
Domains D are sets where $\sqsubseteq_D$ is a cpo.
The smallest domain is $\{\perp\}$.

$$dd^0(\{\perp\}) = \{\perp\}$$

$$dd^1(\{\perp\}) = \{Zero\}_\perp \oplus (\{Succ\} \times \{\perp\})_\perp = \{\perp, Zero, (Succ, \perp)\}$$

$$dd^2(\{\perp\}) = \{Zero\}_\perp \oplus (\{Succ\} \times \{\perp, Zero, (Succ\ \perp)\})_\perp = \{\perp, Zero, (Succ, \perp), (Succ, Zero), (Succ, (Succ, \perp))\}$$

$\vdots$



let $x = Succ\ x$ in x

has the semantics $Succ^\infty \perp$

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*.

$\Rightarrow$ 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 [Int], [Bool], [Int → Int], ...
E.g.: [ ] :: [a], but [ ] can also be used as a list of type [Int], [Bool], ...

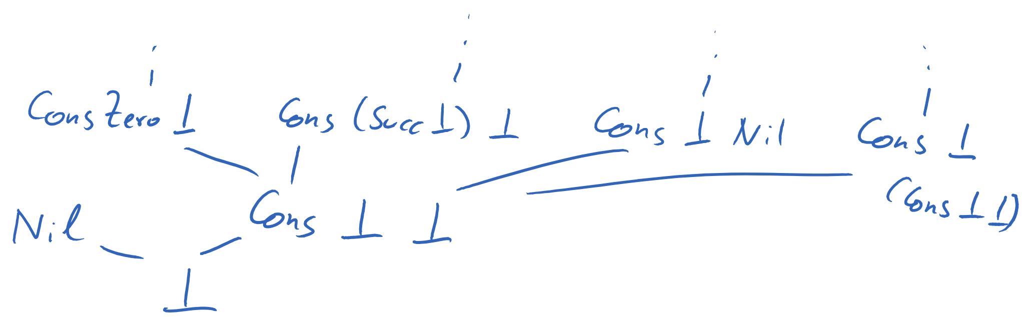
⇝ Domain for [a] is the intersection of the domains for [Int], [Bool], ...

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

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

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

$$D_{List\ Nats} = \{Nil\}_\perp \oplus (\{Cons\} \times D_{Nats} \times D_{List\ Nats})_\perp$$



$$D_{List\ a} = \bigcap_{type} D_{List\ type}$$

$\underline{type}$ is a Haskell-type

$D_{List\ \underline{type}} = \{\perp, Nil, Cons\ \perp\ \perp, Cons\ \perp\ Nil, Cons\ \perp\ (Cons\ \perp\ \perp), \ldots\}$

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 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) (Slide 37)

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

$$Dom = Functions \oplus Tuples_0 \oplus Tuples_2 \oplus Tuples_3 \oplus \ldots \oplus Constructions_0 \oplus Constructions_1 \oplus \ldots$$

where
$$Functions = \langle Dom \to Dom \rangle_\perp$$
$$Tuples_0 = \{()\}_\perp$$
$$Tuples_n = (\underbrace{Dom \times \ldots \times Dom}_{n \text{ times}})_\perp \quad \text{for } n \geq 2$$
$$Constructions_n = (Con_n \times \underbrace{Dom \times \ldots \times Dom}_{n \text{ times}})_\perp \quad \text{for } n \geq 0$$

Dom also contains elements like (Succ, True) in Con-

structions,
that are not needed for the semantics of well-typed Haskell-expressions (but that's no problem).

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