

Free Types

Data structures

We can model any data structure using sets, relations, or functions.

Where **structure** is important, and where different types are combined, a general mechanism is needed.

Free types

The following definition introduces a new type T consisting of elements c_1, c_2, \dots, c_n and elements obtained by applying functions d_1, d_2, \dots, d_n to set expressions E_1, E_2, \dots, E_n :

$$T ::= c_1 \mid \dots \mid c_m \mid d_1 \langle\langle E_1 \rangle\rangle \mid \dots \mid d_n \langle\langle E_n \rangle\rangle$$

Notes

- the elements c_1, c_2, \dots, c_n are called constants
- the functions d_1, d_2, \dots, d_n are called constructors
- the set expressions E_1, E_2, \dots, E_n may include instances of the type being defined

Example

The following **free type** definition introduces a new type constructed using a single constant **zero** and a single constructor function **succ**:

$$nat ::= zero \mid succ\langle nat \rangle$$

This type has a structure which is exactly that of the natural numbers (where **zero** corresponds to 0, and **succ** corresponds to the function **+1**).

Question

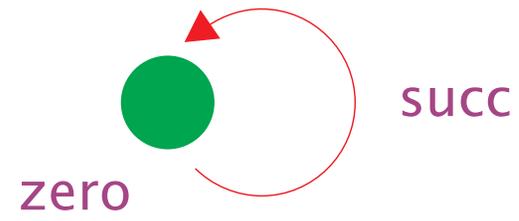
What does this mean? What would we have to do if we wanted to introduce the same set **without** a free type definition?

Attempt 1

$\text{zero} : \text{nat}$

$\text{succ} : \text{nat} \rightarrow \text{nat}$

$\forall n : \text{nat} \bullet n = \text{zero} \vee \exists m : \text{nat} \bullet n = \text{succ } m$



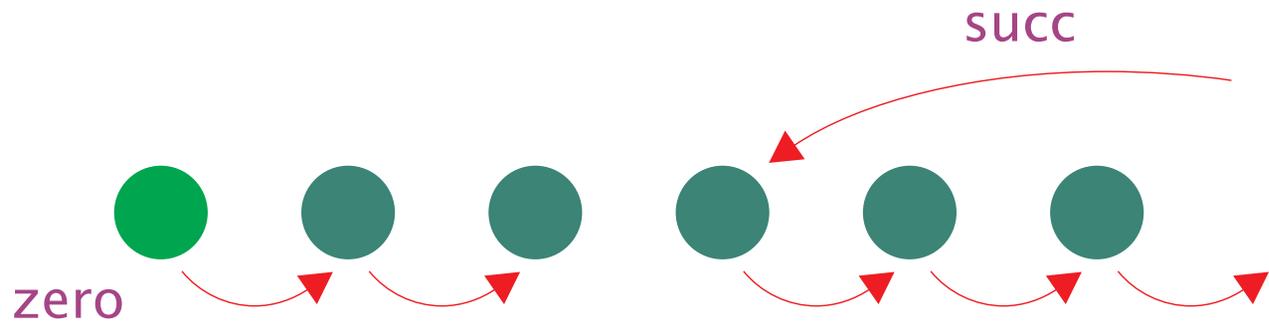
Attempt 2

$\text{zero} : \text{nat}$

$\text{succ} : \text{nat} \rightarrow \text{nat}$

$\forall n : \text{nat} \bullet n = \text{zero} \vee \exists m : \text{nat} \bullet n = \text{succ } m$

$\{\text{zero}\} \cap \text{ran succ} = \emptyset$



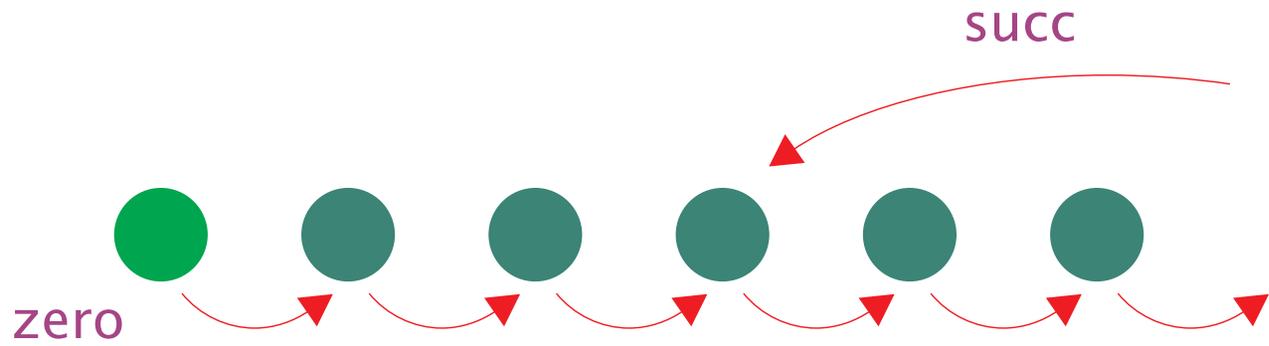
Attempt 3

$\text{zero} : \text{nat}$

$\text{succ} : \text{nat} \rightarrow \text{nat}$

$\forall n : \text{nat} \bullet n = \text{zero} \vee \exists m : \text{nat} \bullet n = \text{succ } m$

$\{\text{zero}\} \cap \text{ran succ} = \emptyset$



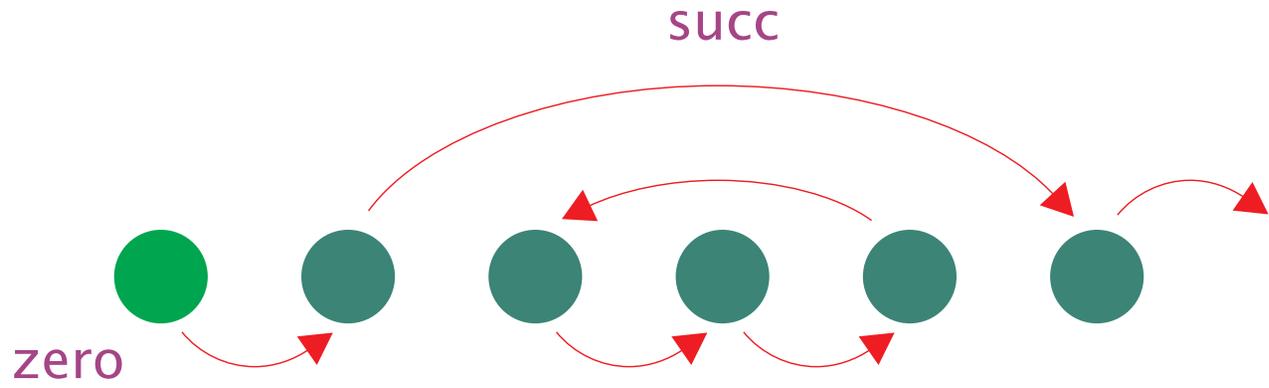
Attempt 4

$\text{zero} : \text{nat}$

$\text{succ} : \text{nat} \rightarrow \text{nat}$

$\{\text{zero}\} \cap \text{ran succ} = \emptyset$

$\{\text{zero}\} \cup \text{ran succ} = \text{nat}$



Conclusion

A free type definition involves:

- constants and constructed elements
- constructor functions
- closure

Multiple constants

Colours ::= red | orange | yellow | green | blue |
indigo | violet

Question

Is the free type definition on the previous slide equivalent to the following abbreviation?

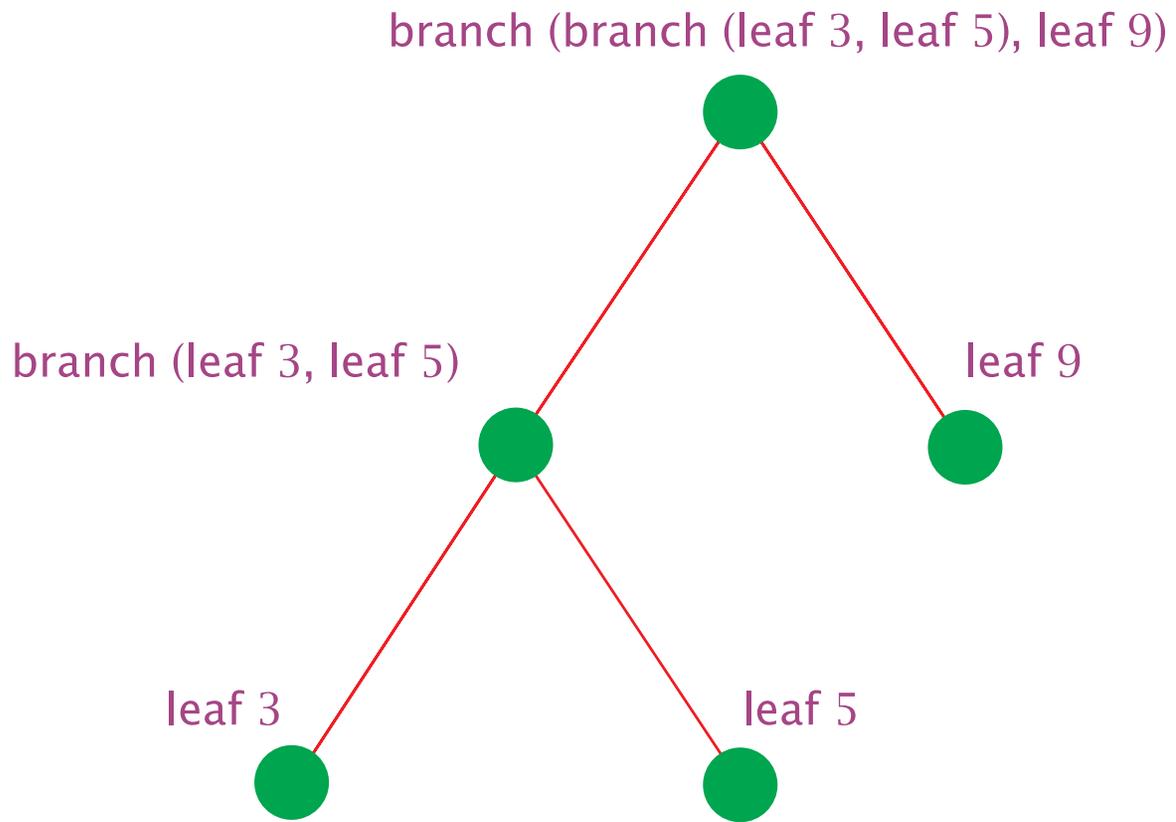
Colours ==

{ *red*, *orange*, *yellow*, *green*, *blue*, *indigo*, *violet* }

If not, why not?

Multiple constructors

$Tree ::= \text{leaf } \langle\langle \mathbb{N} \rangle\rangle \mid \text{branch } \langle\langle Tree \times Tree \rangle\rangle$



Question

What can we say about the functions *leaf* and *branch*?

Example

Degree ::= status $\langle\langle 0 \dots 3 \rangle\rangle$

Useful names for elements of *Degree*:

ba, msc, dphil, ma : Degree

ba = status 0

msc = status 1

dphil = status 2

ma = status 3

The structure is preserved:

$\leq_{status}: Degree \leftrightarrow Degree$

$\forall d_1, d_2 : Degree \bullet$

$d_1 \leq_{status} d_2 \Leftrightarrow status \sim d_1 \leq status \sim d_2$

Induction principle

A recursive free type definition gives rise to a corresponding induction principle.

The free type definition

$$T ::= c_1 \mid \dots \mid c_m \mid d_1 \langle\langle E_1 \rangle\rangle \mid \dots \mid d_n \langle\langle E_n \rangle\rangle$$

has the same effect as a basic type definition

$[T]$

followed by...

$$c_1 : T$$

$$\vdots$$

$$c_m : T$$

$$d_1 : E_1 \twoheadrightarrow T$$

$$\vdots$$

$$d_n : E_n \twoheadrightarrow T$$

disjoint $\langle \{c_1\}, \dots, \{c_m\}, \text{ran } d_1, \dots, \text{ran } d_n \rangle$

$$\forall S : \mathbb{P} T \bullet$$

$$\{c_1, \dots, c_m\} \subseteq S \wedge$$

$$d_1(\downarrow E_1[S / T]) \cup \dots \cup d_n(\downarrow E_n[S / T]) \subseteq S \Rightarrow$$

$$S = T$$

Closure rule

$$S \subseteq T$$

$$\{c_1, \dots, c_m\} \subseteq S$$

$$(d_1 \mid E_1[S / T] \mid) \cup \dots \cup d_n \mid E_n[S / T] \mid) \subseteq S$$

$$S = T$$

Inverse image

$$d_i (E_i[S / T]) \subseteq S$$

$$\Leftrightarrow E_i[S / T] \subseteq d_i^{-1} (S)$$

$$\Leftrightarrow \forall e : E_i[S / T] \bullet e \in d_i^{-1} (S)$$

$$\Leftrightarrow \forall e : E_i[S / T] \bullet d_i e \in S$$

Predicates

S may be the characteristic set of some property P :

$$S == \{t : T \mid P t\}$$

Induction principle

 $P c_1$ \vdots $P c_m$ $\forall e : E_1[S / T] \bullet P (d_1 e)$ \vdots $\forall e : E_n[S / T] \bullet P (d_n e)$

 $\forall t : T \bullet P t$

Example

$$\frac{S \subseteq \text{nat} \quad (\{\text{zero}\} \cup \text{succ}(\text{nat}[S / \text{nat}])) \subseteq S}{S = \text{nat}}$$

Alternative form

$$S == \{n : nat \mid P n\}$$

P zero

$\forall m : nat \bullet P m \Rightarrow P (\text{succ } m)$

$\forall n : nat \bullet P n$

Question

Can you suggest a suitable induction principle for *Tree*?

Consistency

- not all constructions make sense; some result in a free type with no elements
- Cartesian products, finite sequences, finite functions, and finite power sets are guaranteed to work

Example

The following type definition is inconsistent:

$$P ::= \text{power} \langle\langle \mathbb{P} P \rangle\rangle$$

Summary

- data structures
- free type definitions
- constants, constructors, and closure
- induction principle
- consistency