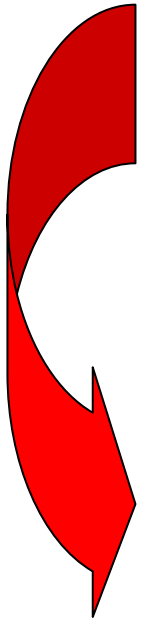


expression : type

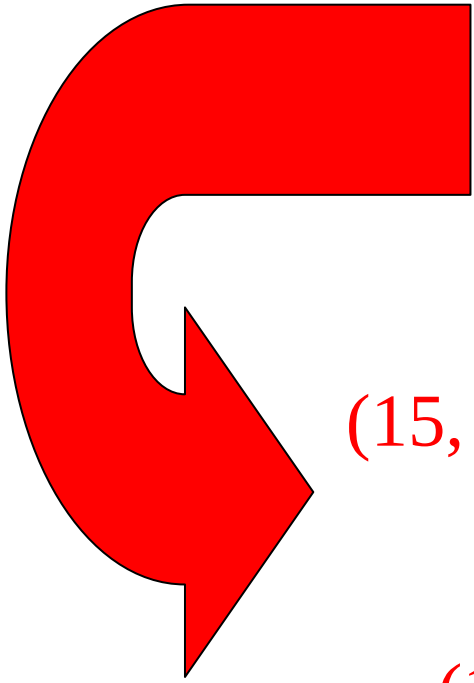
Basic types:

3.14 : real

-57 : integer

17 : nat

TRUE : bool



expression : type

cartesian product

(15, FALSE) : [nat, bool]

(1, 2, 3) : [nat, nat, nat]

(1, (2, 3)) : [nat, [nat, nat]]

These are distinct!

expression : type

record type

[age: 17, married?: FALSE] : [# age: nat, married?: bool #]

[married?: FALSE, age: 17] : [# age: nat, married?: bool #]

Records are a cosmetic variant of cartesian product.
Abstract datatypes will truely add a recursive dimension.

function : [A → B]

function of n parameters = function of one n-uplet parameter

λ-abstraction	:	**function type**
(λ(n: nat): FALSE)	:	[nat → bool]
(λ(m, n: nat): n)	:	[nat, nat → nat]
(λ(mn: [nat, nat]): mn‘2)	:	[[nat, nat] → nat]
(λ(m: nat): λ(n: nat): n)	:	[nat → [nat → nat]]

=

≠

no implicit curification

predicate : [T → **bool**]

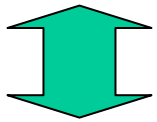
predicate	:	***predicate type***
(λ(n: nat): FALSE)	:	[nat → **bool**]
(λ(m, n: nat): m>n)	:	[nat, nat → **bool**]
(λ(mn: [nat, nat]): >(mn))	:	[[nat, nat] → **bool**]]
>	:	[real, real → **bool**]

predicates are functions
predicates are also called sets

$$\text{set} : [T \rightarrow \textbf{bool}]$$

$$\{m, n: \text{nat} \mid m>n\} \quad : \quad [\text{nat}, \text{nat} \rightarrow \textbf{bool}]$$

set

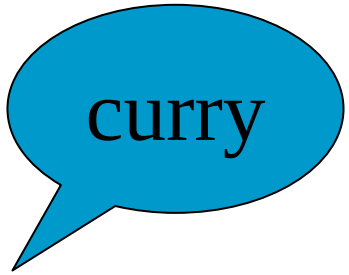


predicate

$$(\lambda(m, n: \text{nat}): m>n) \quad : \quad [\text{nat}, \text{nat} \rightarrow \textbf{bool}]$$

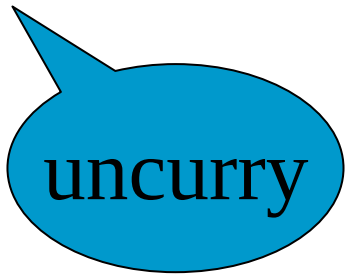
Sets are nothing but predicates.

higher order functions



$(\lambda(f: [A,B\rightarrow C]): (\lambda(a:A): (\lambda(b:B): f(a,b)))) : [[A,B\rightarrow C]\rightarrow[A\rightarrow[B\rightarrow C]]$

$(\lambda(f: [A\rightarrow[B\rightarrow C]]): (\lambda(a:A, b:B): f(a)(b))) : [[A\rightarrow[B\rightarrow C]]\rightarrow [A,B\rightarrow C]]$



formula : bool

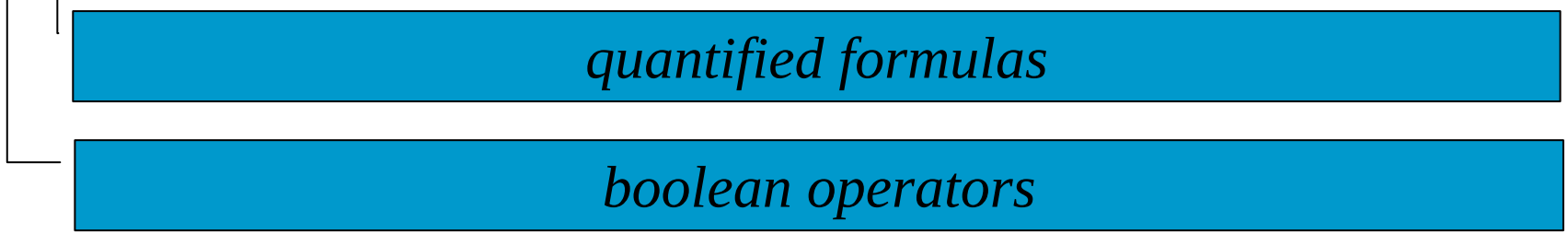
TRUE, FALSE : bool

$\wedge, \vee, \Rightarrow, \Leftrightarrow$: [bool, bool $\rightarrow$ bool]

$\neg$: [bool $\rightarrow$ bool]

$(\forall(m: nat): m+1>m)$: bool

$(\forall(m: nat): \exists(n: nat): n<m)$: bool



higher order formulas

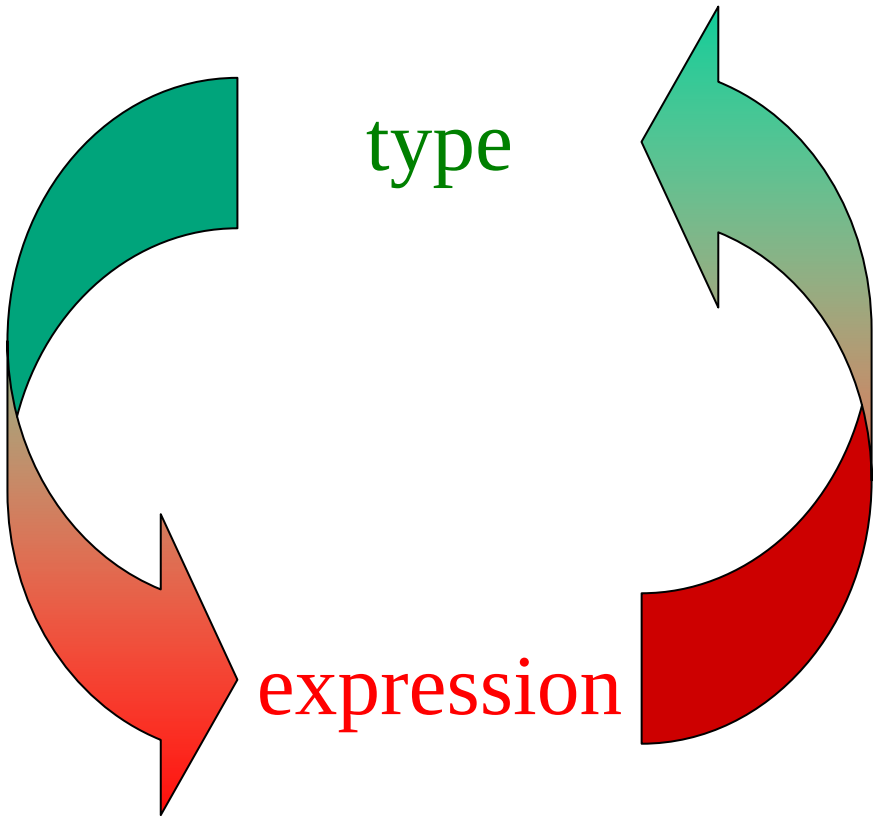
$$(\forall(R: [A, B \rightarrow bool]):$$
$$(\exists(S: [A \rightarrow [B \rightarrow bool]]):$$
$$(\forall(a:A, b:B): R(a,b) \Leftrightarrow S(a)(b))))$$

Logical equivalence is boolean equality:

$$(\forall(\alpha, \beta: bool): (\alpha \Leftrightarrow \beta) = (\alpha = \beta))$$

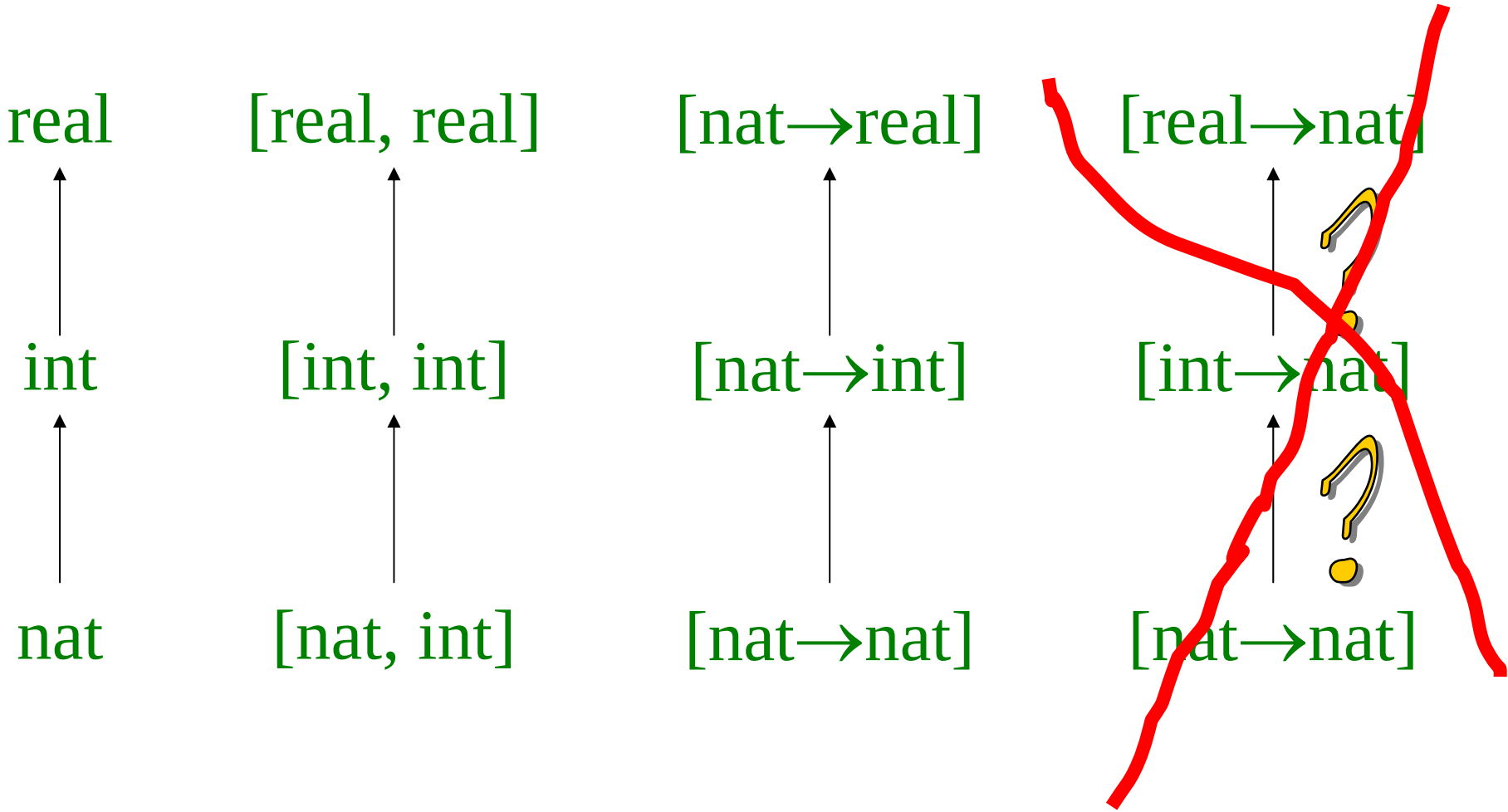
expression and type interaction

variables are typed (typed logic).



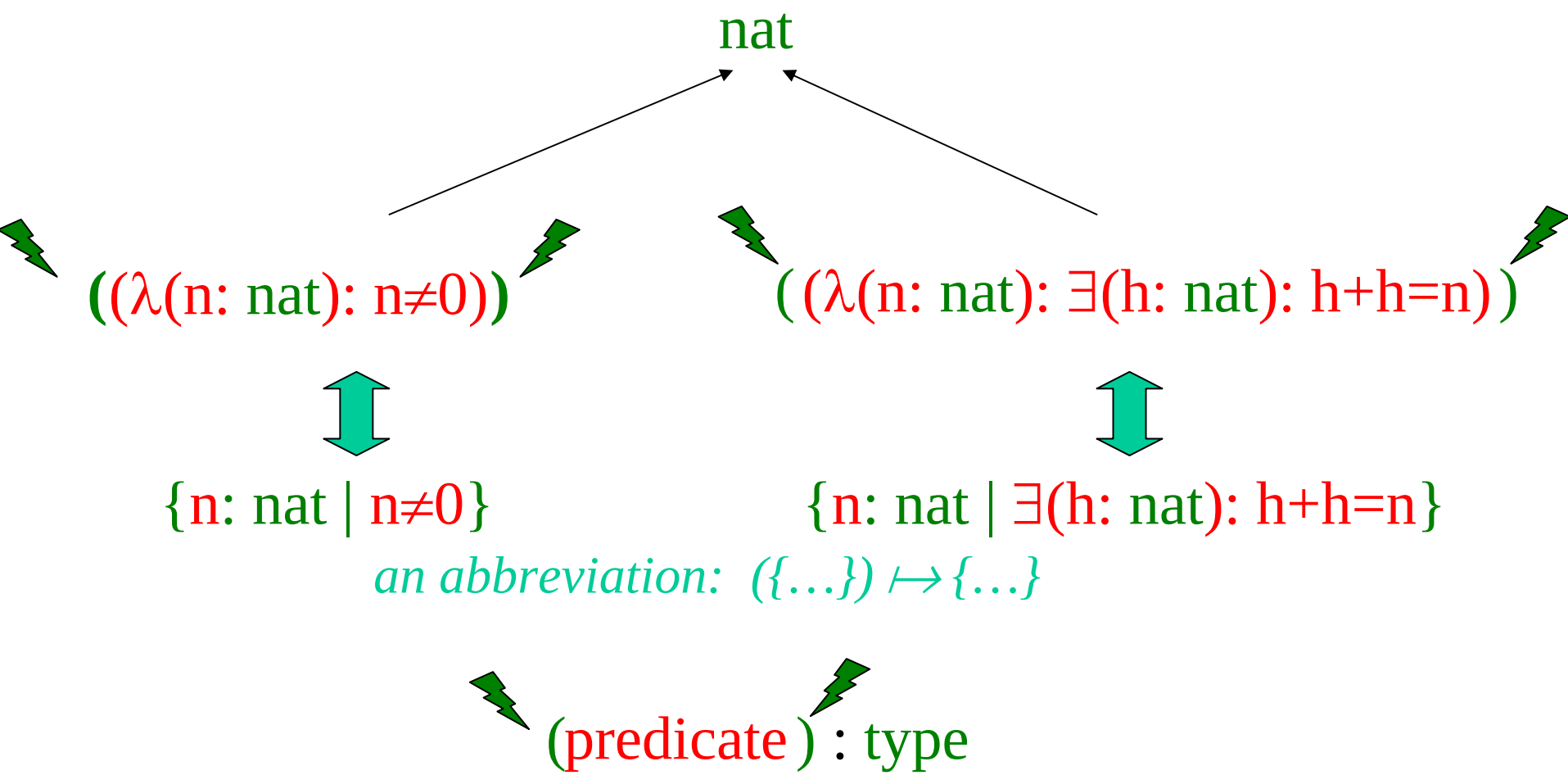
predicate subtyping, a powerful mechanism ...

subtype hierarchies



predicate : *[T → bool]*

predicate subtyping



usefulness of predicate subtyping

PVS knows *total functions* only.

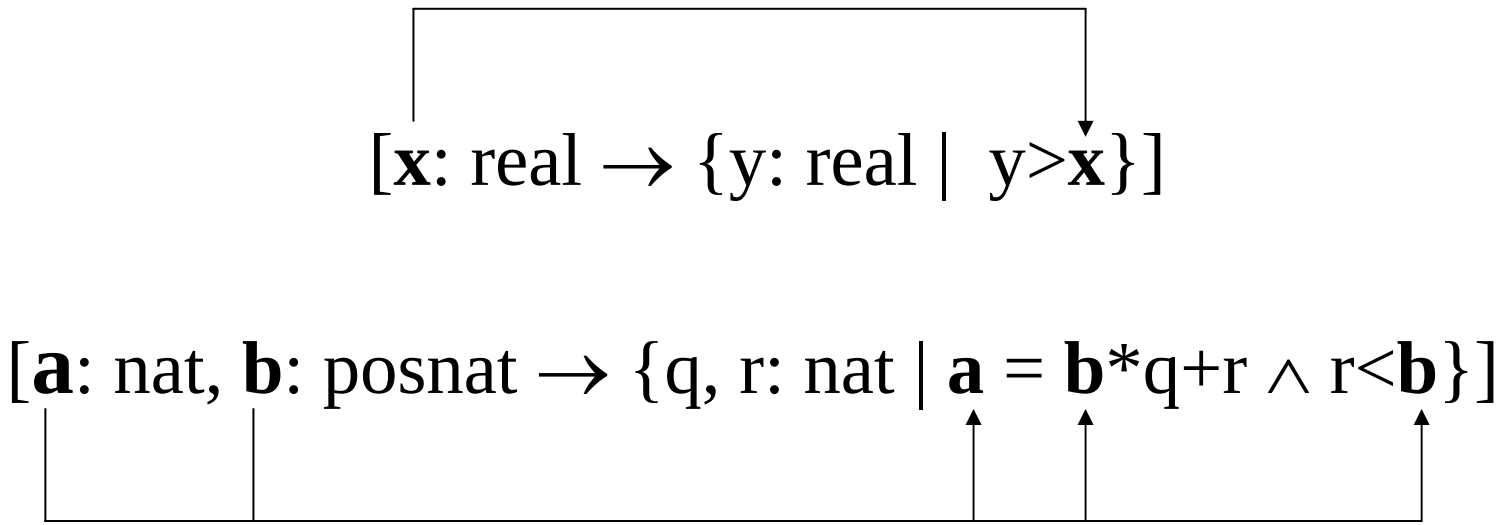
How do we deal with *partial functions*?

Restrict the domain by means of a predicate, e.g.:

- $\{a, b: \text{real} \mid b>a\}$
- $\{n: \text{nat} \mid \exists(x,y,z: \text{int}): \quad x\neq 0 \wedge y\neq 0 \wedge z \neq 0 \wedge x^n+y^n = z^n\}$

PVS type checking is undecidable!
User may have to help PVS type checker!

dependant types



Dependant types are a powerful means of specifying functions.

type declarations

Declaration of a type named T

- T: TYPE
- T: NONEMPTY_TYPE
- T: TYPE+

} *synonymous*

- T: TYPE FROM type_expression

Examples

- Vegetable: TYPE
- Potatoe: TYPE FROM Vegetable
- Thing: TYPE+

A type name cannot be declared twice.
A type name cannot be used before its declaration.

type definition

T: TYPE = type_expr

nat2: TYPE = [nat, nat]

nat22: TYPE = [nat2, nat2]

posnat: TYPE = {n: nat | n≠0}

person: TYPE = [# age: posnat, married?:bool #]

old_person: TYPE = {p: person | age(p) ≥ 99}

A type definition contains a type declaration.
A type name cannot be defined twice.

constant declaration

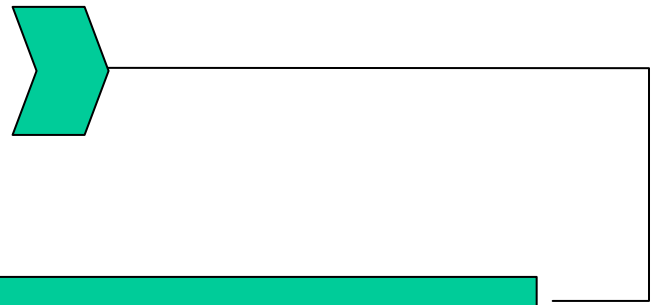
c: type_expr

ρ: {x: real | x>0}

θ: {x: real | 0≤x ∧ x<360}

opposite: [real → real]

opposite: [bool → bool]



Overloading is allowed for constant names.

Names are not variables.
They represent undefined constants.

constant definition

c: type_expr = expr

ρ: posnat = 125

θ: {x: real | 0≤x ∧ x<360} = 60.99

opposite: [real → real] = (λ(x: real): -x)

opposite: [bool → bool] = ¬

point: [real, real] = (200+opposite(ρ), θ)

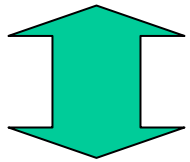
The definition body expr must be typed according to type_expr.

function declaration

f(x_1: type_expr$_1$, …, x_n: type_expr$_n$): type_expr

opposite(x: real): real

opposite(x: bool): bool



f: [x_1: type_expr$_1$, …, x_n: type_expr$_n$ → type_expr]

opposite: [real → real]

opposite: [bool → bool]

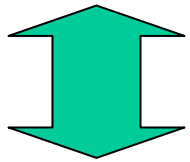
Function parameters x_1, …, x_n must be distinct symbols.
Type dependencies on previous parameters are allowed.
Constants are functions with no parameter.

function definition

$f(x_1: T_1, \ldots, x_n: T_n): T = expr$

opposite(x: real): real = -x

opposite(x: bool): bool = $\neg$x



$f: [x_1: T_1, \ldots, x_n: T_n \rightarrow T] = (\lambda(x_1: T_1, \ldots, x_n: T_n): expr)$

opposite: [real $\rightarrow$ real] = (λ(x: real): -x)

opposite: [bool $\rightarrow$ bool] = $\neg$

Parameters $x_1, \ldots, x_n$ can be used in definition body expr.
The use of parameters in declarations or definitions is cosmetic.
It is always possible to stick to constants, but it may be less readable.

from constant to parameter style

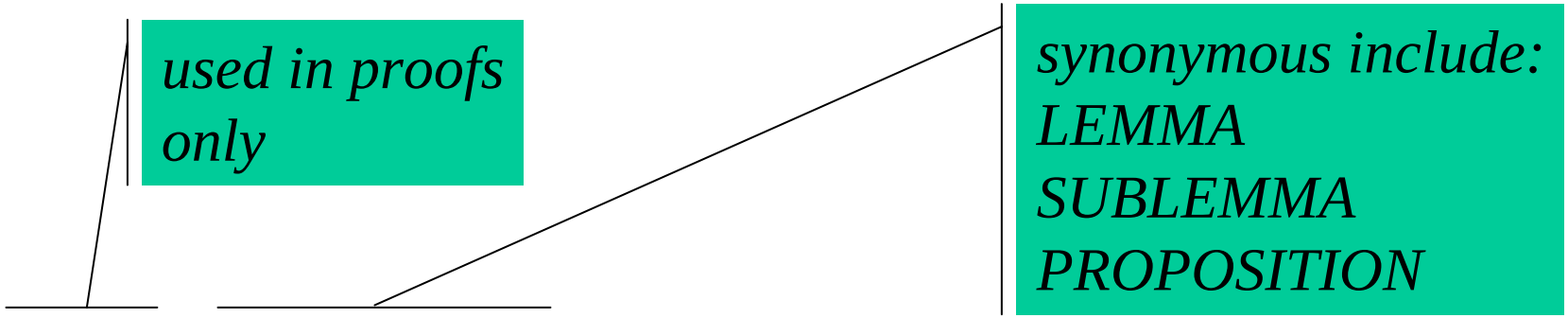
curry: [[A,B→C]→[A→[B→C]]
= (λ(f: [A,B→C]): (λ(a:A): (λ(b:B): f(a,b))))

curry(f: [A,B→C])
: [A→[B→C]]
= (λ(a:A): (λ(b:B): f(a,b)))

curry(f: [A,B→C])
(a:A)
: [B→C]
= (λ(b:B): f(a,b))

curry(f: [A,B→C])
(a:A)
(b: B)
: C
= f(a,b)

theorem declarations



name : *THEOREM* *formula*

square_dif : THEOREM $\forall(a, b: \text{REAL}): (a+b)*(a-b)=a^2-b^2$

Theorems must be proved
in the context of former declarations and definitions.

axioms

Unlike theorems, axioms do not have to be proved!
PVS generates implicit axioms from

- recursive definitions,
- inductive definitions,
- abstract datatype definitions.

Soundness of such axioms is PVS responsability.

A user should never write axioms!
Declare theorems or lemmas instead!

recursive functions
(informal presentation)

$f(x_1: T_1, \ldots, x_n: T_n)$: **RECURSIVE** T
  = expr
  MEASURE m
  BY <<

Assuming that

- <<: [M, M $\rightarrow$bool] is a *well founded* relation;
- m: $[x_1: T_1, \ldots, x_n: T_n \rightarrow M]$;
- The measure m of *each recursive call* in expr decreases according to relation <<.

A formal presentation requires much more work!

well founded relation

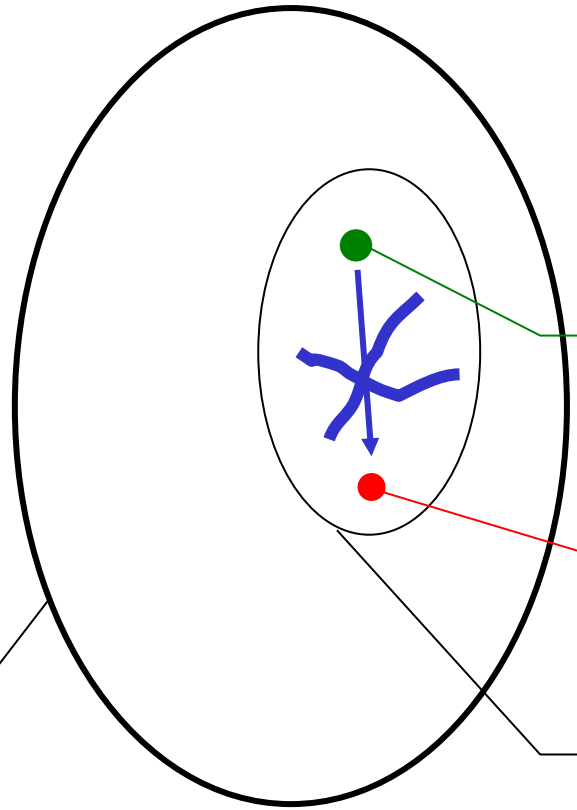
<<: [T, T →bool] is a *well founded* relation
iff every non empty set (in T) has a minimal element.

PVS definition

```
well_founded?(<<: [T, T →bool]): bool
    =    (∀(S: [T → bool]):
              (∃(t: T): S(t))
                   ⇒
              (∃(m: (S)): (∀(s: (S)): ¬ s<<m)))
```

well founded relation
(graphically)

This slide is best viewed under Microsoft PowerPoint amimation mode to see quantifiers arising in the proper order!



∀s in S

∃m in S

Type T

∀ non empty set S

well founded relation
(according to mathematicians)

Unnecessary assumption!

A preorder (reflexive & transitive relation) $\leq$ in T is well founded

iff there is no strictly decreasing infinite sequence

$t_0 > t_1 > t_2 > \ldots > t_n > t_{n+1} > \ldots$.

The strict restriction $<$ of $\leq$ (defined as $\leq \cap \neq$) is well founded according to PVS definition.

inductive sets

$$p(\overbrace{x_1: T_1, \ldots, x_n: T_n}^{\alpha}): \textbf{INDUCTIVE}\ \text{bool}$$
$$= \text{formula}$$

Assuming that

the underlying functional

$$\tau: [[\alpha \to \text{bool}] \to [\alpha \to \text{bool}]]$$
$$= (\lambda(p: [\alpha \to \text{bool}]): (\lambda(\alpha): \text{formula}))$$

is monotonic w.r.t. set inclusion, that is:

$$\forall(p, q: [\alpha \to \text{bool}]): p \subseteq q \Rightarrow \tau(p) \subseteq \tau(q) \ .$$

Inductive sets are convenient though not essential.
They can be derived from fixpoint theory.
Fixpoint theory has been implemented in PVS.

inductive set example

even?(n: nat): **INDUCTIVE** bool
= n=0 ∨ n=2
∨ (∃(n1, n2: (even?)): n = n1+n2)

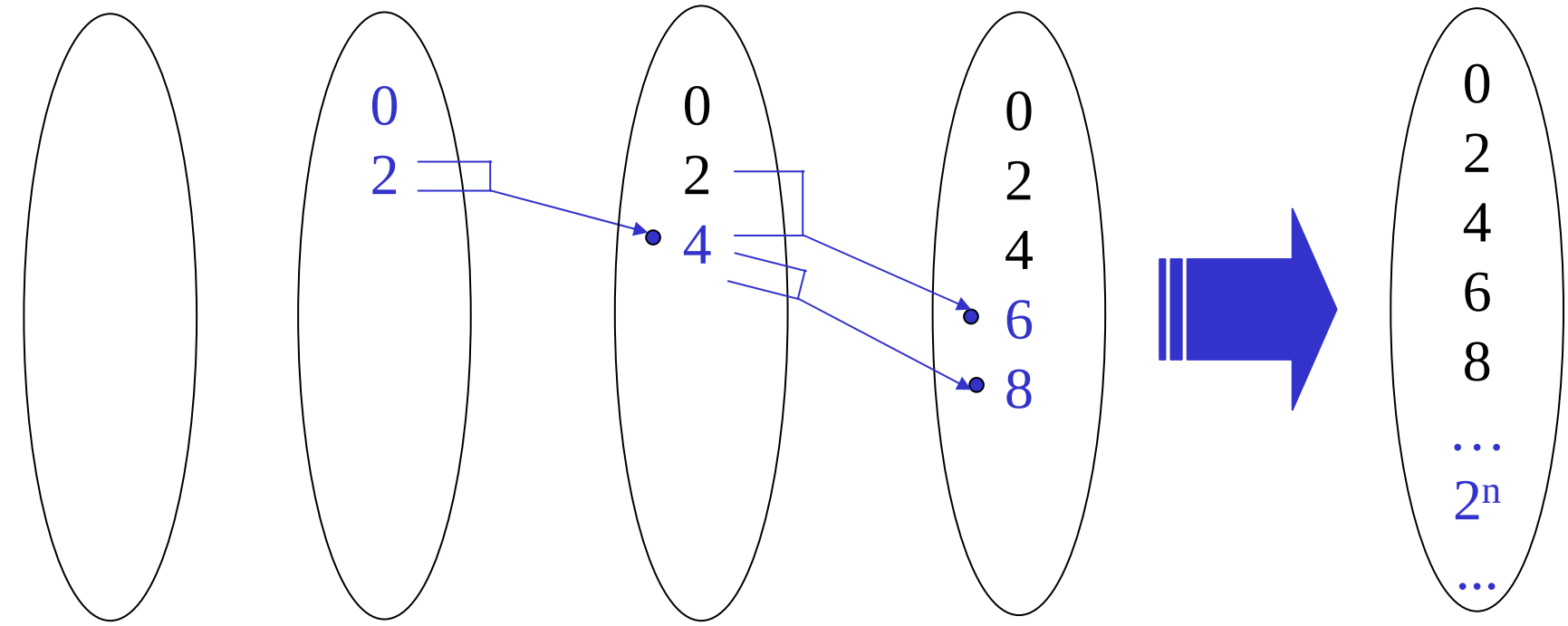
Underlying functional is

τ: [[n: nat → bool] → [n: nat → bool]]
= (λ(p: [n: nat → bool]):
(λ(n: nat): n=0 ∨ n=2 ∨ (∃(n1, n2: (p)): n = n1+n2)))

τ: [[nat → bool] → [nat → bool]]
= (λ(p: [nat → bool]):
(λ(n: nat): n=0 ∨ n=2 ∨ (∃(n1, n2: (p)): n = n1+n2)))

$$\text{even?} = \mu(\tau) = \cup\{\tau^n(\varnothing) \mid n \in \omega\}$$

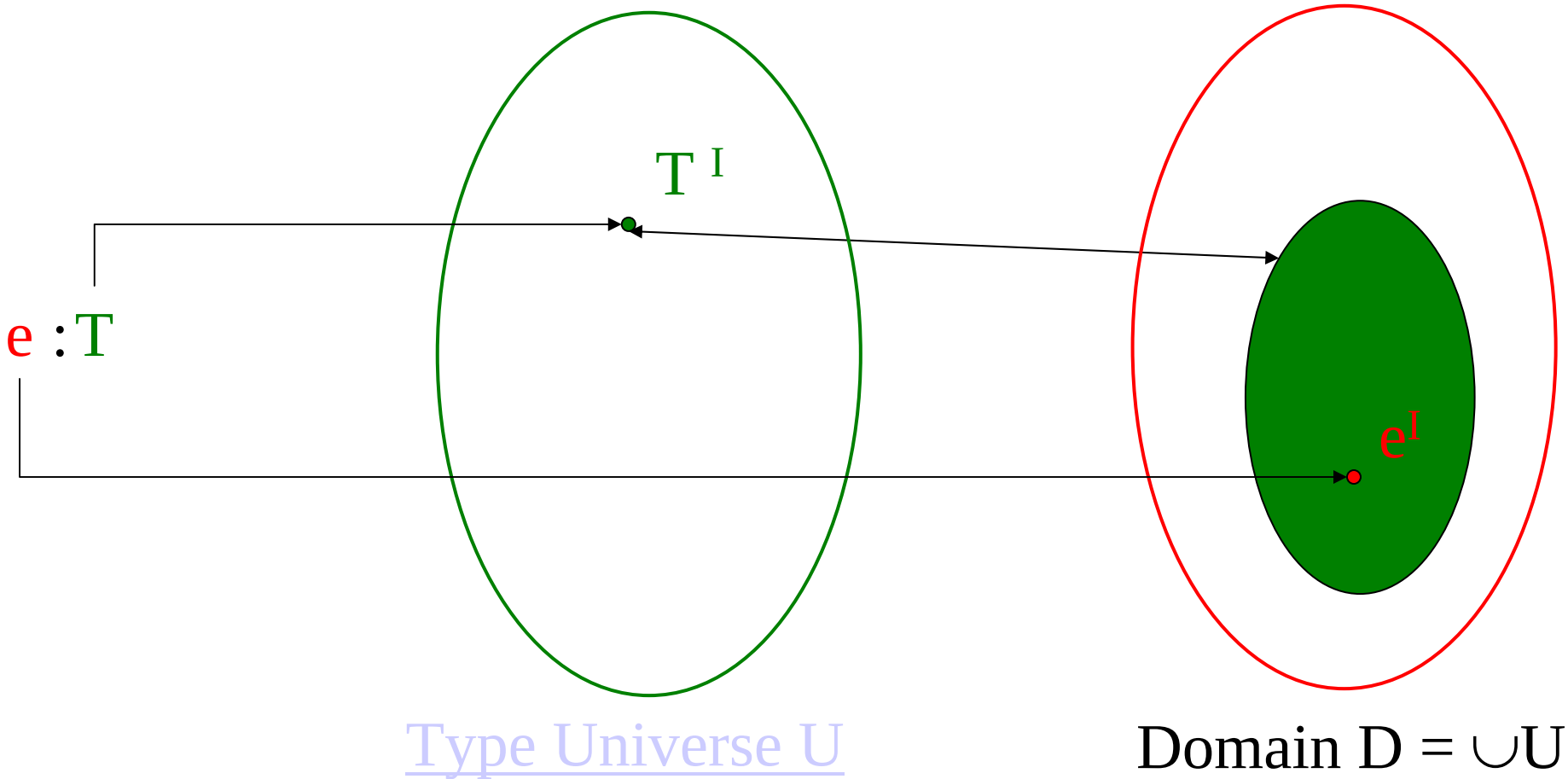
$\tau(p\text{: } [\text{nat} \to \text{bool}])\text{: } [\text{nat} \to \text{bool}]$
$= \{n\text{: nat} \mid n=0 \vee n=2 \vee (\exists(n1, n2\text{: } (p))\text{: } n = n1+n2)\}$



PVS semantics

What are the semantics of a type?
What are the semantics of an expression?
How do they depend upon
  type or constant declarations,
  type or constant definitions,
  theorems,
  axioms?
What is the domain for these semantics?

type and expression semantics
(idea)



domain of PVS semantics

bool

real numbers

$$U_0 = \{\{t, f\}, \Re\}$$

$$U_{n+1} = U_n \cup \{A \times B \mid A, B \in U_n\} \cup \{S' \mid S \in U_n, S' \subset S\} \cup \{\wp(S) \mid S \in U_n\}$$

powerset

type universe

$$\mathbf{U} = \cup\{U_n \mid n \in \omega\}$$

$$\mathbf{D} = \cup\mathbf{U} = \cup\{S \mid S \in \mathbf{U}\}$$

domain

another view of **U** type universe

$$\frac{}{\{t,f\},\ \Re \in \mathbf{U}}$$

$$\frac{S \in \mathbf{U},\ S' \subset S}{S' \in \mathbf{U}}$$

$$\frac{S \in \mathbf{U}}{\wp(S) \in \mathbf{U}}$$

$$\frac{A, B \in \mathbf{U}}{A \times B \in \mathbf{U}}$$

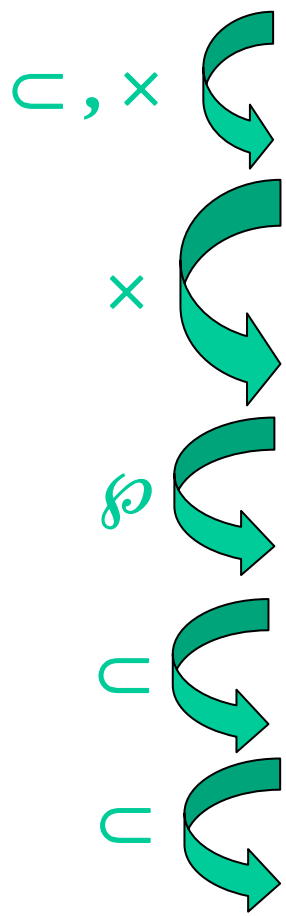
***U** closed under inclusion, powerset, cartesian product*

***U** closed under function power*

$$\frac{A, B \in \mathbf{U}}{B^A \in \mathbf{U}}$$

Set of total functions from A to B

type universe construction



$U_0 \supset \{\mathbf{\{t, f\}}, \Re\}$

$U_1 \supset \{\varnothing, \{t\}, \{f\}, \mathbf{\{t, f\}}, \{3, e, \pi\}, \omega, \Re,$
$\mathbf{\{t, f\}\times\{t, f\}}, \Re\times\{t, f\}, \ldots\}$

$U_2 \supset \{\mathbf{(\{t, f\}\times\{t, f\})\times\{t, f\}}, \ldots\}$

$U_3 \supset \{\mathbf{\wp((\{t, f\}\times\{t, f\})\times\{t, f\})}, \ldots\}$

$U_4 \supset \{\mathbf{\{t, f\}^{(\{t, f\}\times\{t, f\})}}, \ldots\}$

$U_5 \supset \{\mathbf{\{\wedge, \vee, \Leftarrow, \Leftrightarrow\}}, \ldots\}$

I semantics

Assuming that

N_T is the set of type names,

N_E is the set of expression names,

$N_T \cap N_E = \varnothing$,

$N = N_T \cup N_E$,

A semantics I is a total function defined on N as a union of two total functions

I_T: $N_T \rightarrow U$,

I_E: $N_E \rightarrow D$.

Notations

n^I denotes I(n), for $n \in N$

I semantics extensions to types and expressions

Assuming that

T is the set of type expressions,

E is the (disjoint) set of expressions,

each semantics I can be extended into

a unique partial function I composed of

I_T: T $\rightarrow$ U; I_E: E $\rightarrow$ D,

by means of PVS semantic rules.

Notations

t^I denotes I(t), for $t \in T$, when it is defined,

e^I denotes I(e), for $e \in E$, when it is defined.

type semantic rules

$$
\begin{aligned}
\mathrm{bool}^I &= \{t, f\} \\
\mathrm{real}^I &= \Re \\
[A, B]^I &\equiv A^I \times B^I \\
[A_1, A_2, \ldots, A_n]^I &\equiv A_1{}^I \times [A_2, \ldots, A_n]^I \quad \text{for } n>2 \\
[A \to B]^I &\equiv \mathrm{TOTALFUN}(A^I, B^I) \\
(\{a\colon A \mid f\})^I &\equiv \{u \in A^I \mid f^{I[a/u]} = t\} \\
[a\colon A \to (\{b\colon B \mid f\})]^I &\equiv \{\varphi \in \mathrm{TOTALFUN}(A^I, B^I) \\
&\qquad \mid \forall u \in A^I \;\; f^{I[a/u][b/\varphi(u)]} = t\}
\end{aligned}
$$

Most left members may be undefined … click on "=".

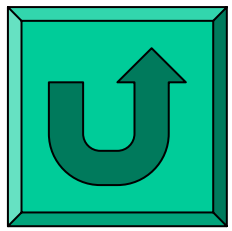
cartesian type semantic rules

$$[A, B]^I = A^I \times B^I$$

provided that A^I and B^I are defined.

$$[A_1, A_2, \ldots, A_n]^I = A_1^I \times [A_2, \ldots, A_n]^I \quad \textit{for } n>2$$

provided that $A_1^I, A_2^I, \ldots, A_n^I$ are defined.

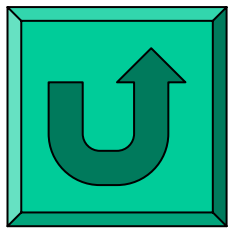


function type semantic rule

the set of total functions from A^I to B^I

$$[A \rightarrow B]^I = \text{TOTALFUN}(A^I, B^I)$$

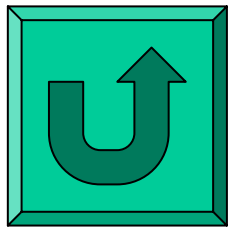
provided that A^I and B^I are defined



predicate subtype semantic rule

$$(\{a\colon A \mid f\})^I = \{u \in A^I \mid f^{I[a/u]} = t\}$$

provided that
A^I and $(\lambda(a\colon A)\colon f)^I$ are defined,
$(\lambda(a\colon A)\colon f)^I \in \mathrm{TOTALFUN}(A^I, \{t, f\})$.



dependant type semantic rule

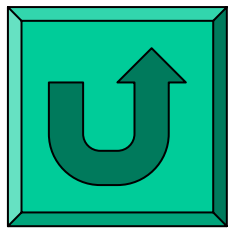
$$[a\colon A \to (\{b\colon B \mid f\})]^I = \{\varphi \in \mathrm{TOTALFUN}(A^I, B^I) \mid \forall u \in A^I \ f^{I[a/u][b/\varphi(u)]} = t\}$$

provided that

A^I, B^I,

$(\lambda(a\colon A)\colon (\lambda(b\colon B)\colon f))^I$ are defined,

$(\lambda(a\colon A)\colon (\lambda(b\colon B)\colon f))^I \in \mathrm{TOTALFUN}(A, \mathrm{TOTALFUN}(B, \{t,f\}))$.



expression semantic rules

$$r^I = r \quad \text{if } r \in \Re$$

$$(f(a))^I \equiv f^I(a^I)$$

$$(\lambda(a: A): f)^I \equiv \{<x, y> \in D^2 \mid x \in A^I, y = f^{I[a/x]}\}$$

application semantic rule

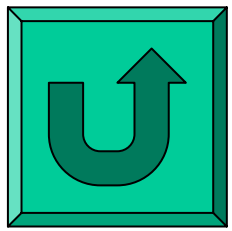
$$(f(a))^I = f^I(a^I)$$

provided that

f^I and a^I are defined,

$\exists A,B \in U \quad f^I \in TOTALFUN(A,B)$

$\wedge \quad a^I \in A.$



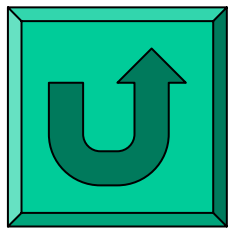
abstraction semantic rule

$$(\lambda(a: A): f)^I = \{<x, y> \in D^2 \mid x \in A^I, \; y = f^{I[a/x]}\}$$

provided that

A^I is defined,

$\forall\, x \in A^I \;\; f^{I[a/x]}$ is defined.



formula semantic rules

$$\begin{aligned}
\mathrm{TRUE}^I &= t \\
\mathrm{FALSE}^I &= f \\
\neg^I &= \{\langle t,f\rangle, \langle f,t\rangle\} \\
\vee^I &= \{\langle\langle t,t\rangle,t\rangle, \langle\langle t,f\rangle,t\rangle, \langle\langle f,t\rangle,t\rangle, \langle\langle f,f\rangle,f\rangle\} \\
\wedge^I &= \{\langle\langle t,t\rangle,t\rangle, \langle\langle t,f\rangle,f\rangle, \langle\langle f,t\rangle,f\rangle, \langle\langle f,f\rangle,f\rangle\} \\
\Rightarrow^I &= \{\langle\langle t,t\rangle,t\rangle, \langle\langle t,f\rangle,f\rangle, \langle\langle f,t\rangle,t\rangle, \langle\langle f,f\rangle,t\rangle\} \\
\Leftrightarrow^I &= \{\langle\langle t,t\rangle,t\rangle, \langle\langle t,f\rangle,f\rangle, \langle\langle f,t\rangle,f\rangle, \langle\langle f,f\rangle,t\rangle\} \\
=_A{}^I &= \{\langle\langle d,d\rangle,t\rangle \mid d\in A^I\} \\
&\quad \cup \{\langle\langle d,e\rangle,f\rangle \mid d,e\in A^I, d\neq e\} \\
(\forall(a\colon A)\colon \alpha)^I &\equiv ((\lambda(a\colon A)\colon \alpha) =_A (\lambda(a\colon A)\colon \mathrm{TRUE}))^I \\
(\exists(a\colon A)\colon \alpha)^I &\equiv (\neg(\forall(a\colon A)\colon \neg\alpha))^I
\end{aligned}$$

a type

quantifier semantic rules
(classical style)

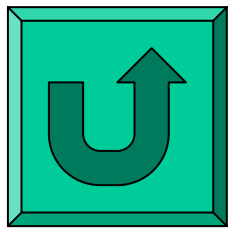
$$(\forall(a\colon A)\colon \alpha)^I = \wedge\{\alpha^{I[a/x]} \mid x \in A^I\}$$
$$(\exists(a\colon A)\colon \alpha)^I = \vee\{\alpha^{I[a/x]} \mid x \in A^I\}$$

provided that

A^I is defined,

$(\lambda(a\colon A)\colon \alpha)^I$ is defined,

$(\lambda(a\colon A)\colon \alpha)^I \in \text{TOTALFUN}(A^I, \{t, f\})$



Every $?^I$ must have a defined semantics

$?^I$

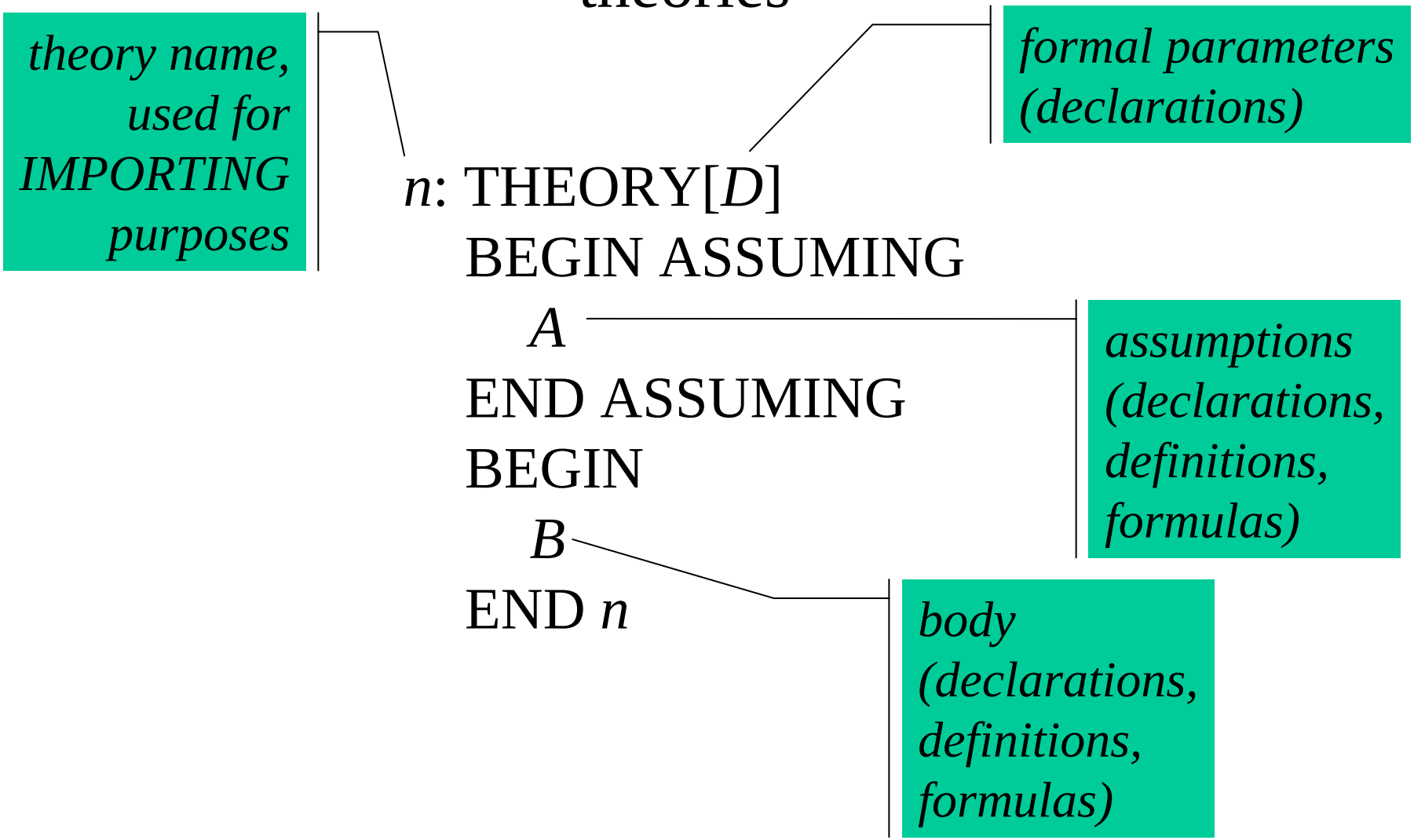
I *is a model of ...*

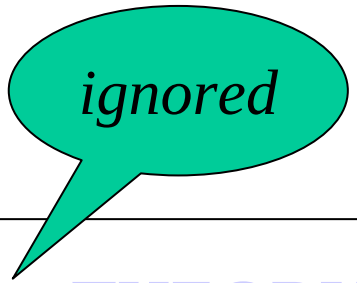
$\alpha|A$ *entails* $\beta|B$
$\beta|B$ *is a consequence of* $\alpha|A$

$$
\begin{array}{lcl}
 & \models & \\
I \models U{:}TYPE+ & \equiv & U^I \neq \emptyset \\
I \models U \leq V & \equiv & U^I \subseteq V^I \\
I \models U = V & \equiv & U^I = V^I \\
I \models \alpha : U & \equiv & \alpha^I \in U^I \\
I \models \alpha & \equiv & \alpha^I = t \\
I \models A & \equiv & \forall a \in A \;\; I \models a \\
\alpha \models \beta & \equiv & \forall I \, (I \models \alpha \;\Rightarrow\; I \models \beta) \\
\alpha \models B & \equiv & \forall b \in B \;\; \alpha \models b \\
A \models \beta & \equiv & \forall I \, (I \models A \;\Rightarrow\; I \models \beta) \\
A \models B & \equiv & \forall b \in B \;\; A \models b
\end{array}
$$

I: a semantics,
U, V: types,
α, β: expressions or formulas (lemmas),
A, B: sets of TYPE declarations or definitions or lemmas.

theories


theory name, used for IMPORTING purposes
formal parameters (declarations)
n: THEORY[D]
BEGIN ASSUMING
A
END ASSUMING
BEGIN
B
END n
assumptions (declarations, definitions, formulas)
body (declarations, definitions, formulas)



theory semantics

n: THEORY[D]
BEGIN ASSUMING A END ASSUMING
BEGIN B END

$$
\begin{aligned}
I \models \text{THEORY[D]}\ldots A \ldots B &\equiv I \models D \cup A \Rightarrow I \models B \\
\alpha \models \text{THEORY[D]}\ldots A \ldots B &\equiv \forall I\,(I \models \alpha \Rightarrow I \models \text{THEORY...}) \\
&\equiv \{\alpha\} \cup D \cup A \models B \\
C \models \text{THEORY[D]}\ldots A \ldots B &\equiv \forall I\,(I \models C \Rightarrow I \models \text{THEORY...}) \\
&\equiv C \cup D \cup A \models B
\end{aligned}
$$

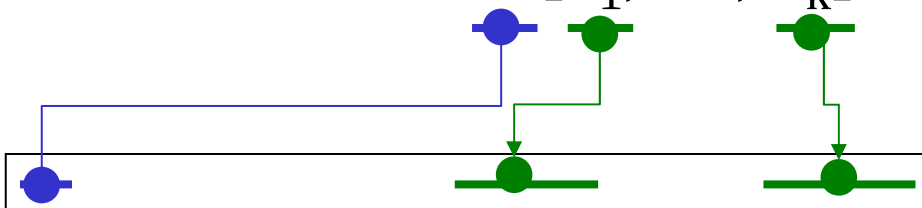
theory instances

like macro expansion

must agree with theory parameters $p_1, \ldots, p_k$, and assumptions A.

actuals

IMPORTING th[a_1, …, a_k]



th: THEORY[p_1: d_1, …, p_k: d_k]

BEGIN ASSUMING A END ASSUMING

BEGIN B END

importing semantics

$I \models \text{IMPORTING th[actuals]} \equiv I \models (D \cup A \cup B)[P/\text{actuals}]$

where $D = \text{th}^I.\text{declarations}$,
$A = \text{th}^I.\text{assumptions}$,
$B = \text{th}^I.\text{body}$,
$P = \text{th}^I.\text{parameters}$

$\alpha \models \text{IMPORTING th[actuals]} \equiv \forall I\,(I \models \alpha \Rightarrow I \models \text{IMPORTING ...})$
$\equiv \alpha \models (D \cup A \cup B)[P/\text{actuals}]$
where … same as above ...

$C \models \text{IMPORTING th[actuals]} \equiv \forall I\,(I \models C \Rightarrow I \models \text{IMPORTING ...})$
$\equiv C \models (D \cup A \cup B)[P/\text{actuals}]$
where … same as above ...

We assume a set of theory names and an extension of basic definition of I semantics such that th^I *is the theory associated with th name.*

abstract data types

```
dt[type_declarations]: DATATYPE
    BEGIN
        constructor_declarations
    END dt
```

```
list[T: TYPE]: DATATYPE
    BEGIN
        null: (null?)
        cons(car: T, cdr: list): (cons?)
    END list
```

Abstract datatype definitions are like macros that expand into theories, with lot's of axioms, whose coherence is guaranteed by PVS.