

Funcons-beta: Value-Types *

The P_LanCompS Project

Value-Types.cbs | PLAIN | PRETTY

OUTLINE

- Value Types
 - Values
 - Types
 - Option types
 - Ground values

Value Types

- [*Type* values
- Alias* vals
- Type* value-types
- Alias* types
- Type* empty-type
- Funcon* is-in-type
- Alias* is
- Funcon* is-value
- Alias* is-val
- Funcon* when-true
- Alias* when
- Type* cast-to-type
- Alias* cast
- Type* ground-values
- Alias* ground-vals
- Funcon* is-equal
- Alias* is-eq]

Values

- Built-in Type* values
- Alias* vals = values

The type **values** includes all values provided by CBS.

Some funcons are declared as value constructors. Values are constructed by applying value constructor funcons to the required arguments.

Values are immutable and context-independent. Their structure can be inspected using patterns formed from value constructors and variables. Computations can be extracted from values and executed, but the structure of computations cannot be inspected.

*Suggestions for improvement: plancomps@gmail.com.
Reports of issues: <https://github.com/plancomps/CBS-beta/issues>.

Some types of values and their funcons are declared as built-in, and not further specified in CBS. New types of built-in values can be added to CBS by its developers.

New algebraic datatypes may be declared by users of CBS. Their values are disjoint from built-in values.

Meta-variables $T, T_1, T_2 <: \text{values}$

Types

Built-in Type value-types

Alias $\text{types} = \text{value-types}$

Built-in Type empty-type

A type T is a value that represents a set of values.

The values of type types are all the types, including types itself.

The formula $V : T$ holds when V is a value of type T , i.e., V is in the set represented by the type T .

The formula $T_1 <: T_2$ holds when T_1 is a subtype of T_2 , i.e., the set represented by T_1 is a subset of the set represented by T_2 .

The set of types forms a Boolean algebra with the following operations and constants:

- $T_1 \& T_2$ (meet/intersection)
- $T_1 | T_2$ (join/union)
- $\sim T$ (complement)
- values (top)
- empty-type (bottom)

Subtyping: $T_1 <: T_2$ is the partial order defined by the algebra.

Subsumption: If $V : T_1$ and $T_1 <: T_2$ both hold, so does $V : T_2$.

Indivisibility: For each value V and type T , either $V : T$ or $V : \sim T$ holds.

Universality: $V : \text{values}$ holds for all values V .

Emptiness: $V : \text{empty-type}$ holds for no value V .

'Type N ' declares the name ' N ' to refer to a fresh value constructor and includes it as an element of types .

'Type $N \sim> T$ ' moreover specifies 'Rule $N \sim> T$ ', so that ' N ' can be used as an abbreviation for the type term ' T '.

'Type $N <: T$ ' declares the name ' N ' to refer to a fresh value constructor in types , and asserts ' $N <: T$ '.

Parametrised type declarations introduce generic (possibly dependent) types, i.e., families of individual types, indexed by types (and by other values). For example, $\text{lists}(T)$ is parameterised by the type of list elements T . Replacing a parameter by $_$ denotes the union over all instances of that parameter, e.g., $\text{lists}(_)$ is the union of all types $\text{lists}(T)$ with $T : \text{types}$.

Qualified variables $V : T$ in terms range over values of type T . Qualified variables $T_1 <: T_2$ in terms range over subtypes T_1 of T_2 .

Funcon $\text{is-in-type}(V : \text{values}, T : \text{types}) : \Rightarrow \text{booleans}$

Alias $\text{is} = \text{is-in-type}$

`is-in-type`(V, T) tests whether $V : T$ holds. The value V need not be a ground value, but T should not require testing any computation types.

$$\begin{array}{l} \text{Rule} \quad \frac{V : T}{\text{is-in-type}(V : \text{values}, T : \text{types}) \rightsquigarrow \text{true}} \\ \text{Rule} \quad \frac{V : \sim T}{\text{is-in-type}(V : \text{values}, T : \text{types}) \rightsquigarrow \text{false}} \end{array}$$

Option types

For any value type T , the elements of the option type $(T)?$ are the elements of T together with the empty sequence $()$, which represents the absence of a value. Option types are a special case of sequence types.

A funcon whose result type is an option type $(T)?$ may compute a value of type T or the empty sequence $()$; the latter represents undefined results of partial operations.

The parentheses in $(T)?$ and $()$ can be omitted when this does not give rise to grouping ambiguity. Note however that $T?$ is a meta-variable ranging over option types, whereas $(T)?$ is the option type for the value type T .

$$\begin{array}{l} \text{Funcon} \quad \text{is-value}(_ : \text{values}?) : \Rightarrow \text{booleans} \\ \text{Alias} \quad \text{is-val} = \text{is-value} \end{array}$$

`is-value`($V?$) tests whether the optional value $V?$ is a value or absent.

$$\begin{array}{l} \text{Rule} \quad \text{is-value}(_ : \text{values}) \rightsquigarrow \text{true} \\ \text{Rule} \quad \text{is-value}() \rightsquigarrow \text{false} \end{array}$$

$$\begin{array}{l} \text{Funcon} \quad \text{when-true}(_ : \text{booleans}, _ : T) : \Rightarrow (T)? \\ \text{Alias} \quad \text{when} = \text{when-true} \end{array}$$

`when-true`(B, V) gives V when B is `true`, and $()$ when B is `false`.

$$\begin{array}{l} \text{Rule} \quad \text{when-true}(\text{true}, V : \text{values}) \rightsquigarrow V \\ \text{Rule} \quad \text{when-true}(\text{false}, V : \text{values}) \rightsquigarrow () \end{array}$$

$$\begin{array}{l} \text{Funcon} \quad \text{cast-to-type}(V : \text{values}, T : \text{types}) : \Rightarrow (T)? \\ \text{Alias} \quad \text{cast} = \text{cast-to-type} \end{array}$$

`cast-to-type`(V, T) gives V if it is in T , otherwise $()$.

$$\begin{array}{l} \text{Rule} \quad \frac{V : T}{\text{cast-to-type}(V : \text{values}, T : \text{types}) \rightsquigarrow V} \\ \text{Rule} \quad \frac{V : \sim T}{\text{cast-to-type}(V : \text{values}, T : \text{types}) \rightsquigarrow ()} \end{array}$$

Ground values

$$\begin{array}{l} \text{Built-in Type} \quad \text{ground-values} \\ \text{Alias} \quad \text{ground-vals} = \text{ground-values} \end{array}$$

The elements of **ground-values** are all values that are formed entirely from value-constructors, and thus do not involve computations.

A type is a subtype of **ground-values** if and only if all its elements are included in **ground-values**.

Funcon **is-equal**($V : \text{values}, W : \text{values}$) : \Rightarrow **booleans**

Alias **is-eq** = **is-equal**

is-equal(V, W) returns **true** when V and W are identical ground values, otherwise **false**.

Rule
$$\frac{V == W}{\text{is-equal}(V : \text{ground-values}, W : \text{ground-values}) \rightsquigarrow \text{true}}$$

Rule
$$\frac{V \neq W}{\text{is-equal}(V : \text{ground-values}, W : \text{ground-values}) \rightsquigarrow \text{false}}$$

Rule **is-equal**($V : \sim \text{ground-values}, W : \text{values}$) \rightsquigarrow **false**

Rule **is-equal**($V : \text{values}, W : \sim \text{ground-values}$) \rightsquigarrow **false**