Funcons-beta: Value-Types *

The PLanCompS Project

`Value-Types.cbs` | PLAIN | PRETTY

OUTLINE

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 <: values

Types

Built-in Type value-types

Alias types = 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)
- $T_1 \mid T_2$ (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 : values holds for all values V.

Emptiness: V : 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 ~> T' moreover specifies 'Rule N ~> 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, 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., lists(_) is the union of all types lists(T) with T : 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 is-in-type(V : values, T : types) : ⇒ booleans

Alias is = 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.

Rule $$\frac{V : T}{\text{is-in-type}(V : \text{values}, T : \text{types}) \rightsquigarrow \text{true}}$$

Rule $$\frac{V : \sim T}{\text{is-in-type}(V : \text{values}, T : \text{types}) \rightsquigarrow \text{false}}$$

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 grouing ambiguity. Note however that $T?$ is a meta-variable ranging over option types, whereas $(T)?$ is the option type for the value type T.

Funcon is-value(_ : values?) : ⇒ booleans

Alias is-val = is-value

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

Rule is-value(_ : values) ⇝ true

Rule is-value( ) ⇝ false

Funcon when-true(_ : booleans, _ : T) : ⇒(T)?

Alias when = when-true

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

Rule when-true(true, V : values) ⇝ V

Rule when-true(false, V : values) ⇝ ( )

Funcon cast-to-type(V : values, T : types) : ⇒(T)?

Alias cast = cast-to-type

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

Rule $$\frac{V : T}{\text{cast-to-type}(V : \text{values}, T : \text{types}) \rightsquigarrow V}$$

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

Ground values

Built-in Type ground-values

Alias ground-vals = ground-values

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

Alias $\text{is-eq} = \text{is-equal}$

$\text{is-equal}(V, W)$ returns true when V and W are identical ground values, otherwise false.

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

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

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

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