A three-step plan to unifying types and values

Step 1. Start with some values.

\[
\begin{array}{ccc}
1 & c & b \\
2 & & a \\
3 & & \\
\end{array}
\]

Step 2. Identify each value with its own type.

\[
\begin{array}{ccc}
1 & c & b \\
2 & e & b \\
3 & a & \\
\end{array}
\]

Step 3. Provide an operation to fuse types together.

\[
\begin{array}{ccc}
1 & c & b \\
2 & e & b \\
3 & a & \\
\end{array}
\]

Done!

What do we gain?

Since values are their own types, we can define, for all values \( v \) and \( a \):

\[
v \text{ is of type } a \iff v \text{ is a subfusion (or part) of } a
\]

Thus, elementhood (\( \in \)) has been replaced with parthood (\( \subset \)).

NB. The type hierarchy is now flat.

NB. We are effectively switching from set theory to mereology. Apparently, then...

**type checking = subtyping!**

What else do we need?

In order to build a functional programming language on top of such a universe, we need some constructs:

- Atoms (e.g., integers \([0, 1, -1, \ldots]\); list-like symbols \([a, b, \ldots]\)
- Fusions \( a \sqcup b \)
- Pairs \((s, t)\)
- Patterns \( p \) including bound annotations \( p <: t \)
- Function literals \([\text{fun}\{p \rightarrow q\}]\) with pattern matching
- Fusion comprehensions \([\text{fun}\{p\}]\) based on patterns
- Fixed-points \( \text{fix } x \rightarrow x \)
- \( \text{let} \) expressions (for convenience)

Example code

Algebraic data types

```plaintext
let nat = fix nat \rightarrow 'zero \oplus ('succ, nat)
let list = fix list \rightarrow
  fun \{a \rightarrow 'nil \oplus ('cons, a, list a)}
let list' = fix list' \rightarrow
  fun \{a \rightarrow 'nil \oplus (a, list' a)}
let bintree = fix bintree \rightarrow
  fun \{a \rightarrow 'leaf \oplus ('inner, a, bintree a, bintree a)}
```

Dependent types

```plaintext
let vec = fix vec \rightarrow
  fun \{a \rightarrow fun ('zero \rightarrow 'vnil; ('succ, n) \rightarrow ('vcons, a, vec a n))
```

Polymorphic function types

```plaintext
let cons =
  fun \{a \rightarrow fun \{(x<:a) \rightarrow fun \{(l<:list a) \rightarrow ('cons, x, l))\}\}
```

\( \Sigma \)-types (comprehensions)

```plaintext
let veclist =
  fun \{a \rightarrow \{(n:<nat, v:<vec a n))\}
let nonempty_veclist =
  fun \{a \rightarrow \{(n:<('succ,nat), v:<vec a n))\}
```

Static checking

Our system can **statically check bounds**:

**Peano-naturals**

\[
\begin{array}{ccc}
'zero & <: & nat \\
('succ, ('succ, 'zero)) & <: & nat \\
'zero \oplus ('succ, ('succ, 'zero)) & <: & nat \\
('succ, nat) & #: & nat
\end{array}
\]

**Algebraic data types**

\[
\begin{array}{ccc}
('cons, ('succ, 'zero), 'nil) & <: & list nat \\
('succ, 'zero), 'zero, 'nil) & <: & list' nat
\end{array}
\]

**Type-level types**

\[
\begin{array}{ccc}
vec & <: & T \rightarrow nat \rightarrow T
\end{array}
\]

**Induction**

```plaintext
fix map = fix map \rightarrow
  fun \{a \rightarrow
    fun \{f \rightarrow
      fun \{'nil \rightarrow 'vnil;
      ('cons, x, xs) \rightarrow ('cons, f x, map a f xs))\}\}
    <: fun \{a \rightarrow \{(a \rightarrow a) \rightarrow list a \rightarrow list a)\}
```

**\( \Sigma \)-types**

```plaintext
fun \{(n, ('vcons, x, xs) \rightarrow x)\}
  <: nonempty_veclist a \rightarrow a
```

Formal checking rules

The bound checking algorithm has been **formalized** as a set of inductively defined predicates.

Further reading