Type Checking Without Types

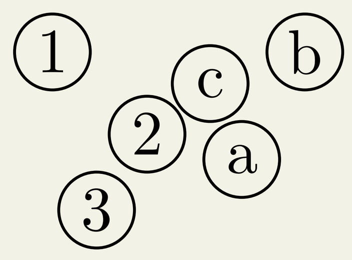
Matthias Benkard
LMU München
Diploma thesis

A three-step plan to unifying types and values

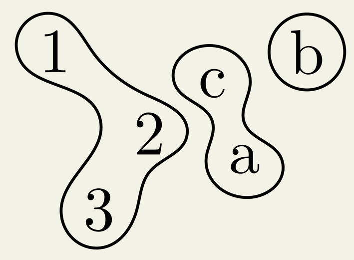
Step 1. Start with some values.

1 c b 2 a 3

Step 2. Identify each value with its own type.



Step 3. Provide an operation to fuse types together.



Done!

What do we gain?

Since values are their own types, we can define, for all values v and α:

$$v \text{ is of type } \alpha \overset{def}{\iff} v \text{ is a subfusion (or part) of } \alpha$$

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 $\boxed{0}$, $\boxed{1}$, $\boxed{-1}$, …; Lisp-like symbols $\boxed{'a}$, $\boxed{'b}$, $\boxed{'succ}$, …)
- Fusions $\boxed{s \oplus t}$
- Pairs $\boxed{(s,t)}$
- Patterns $\boxed{p}$, including bound annotations $\boxed{p <: t}$
- Function literals $\boxed{\text{fun } \{\overrightarrow{p \to e}\}}$ with pattern matching
- Fusion comprehensions $\boxed{\{p\}}$ based on patterns
- Fixed-points $\boxed{\text{fix } x \to e}$
- $\boxed{\text{let}}$-expressions (for convenience)

Example code

Algebraic data types

“constructor tags” are mostly optional

```
let nat = fix nat →
  'zero ⊕ ('succ, nat)

let list = fix list →
  fun {a → 'nil ⊕ ('cons, a, list a)}

let list' = fix list' →
  fun {a → 'nil ⊕ (a, list' a)}

let bintree = fix bintree →
  fun {a → 'leaf ⊕ ('inner, a, bintree a, bintree a)}
```

Dependent types

```
let vec = fix vec →
  fun {a → fun {'zero       → 'vnil;
               ('succ, n)  → ('vcons, a, vec a n)}}
```

Polymorphic function types

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

Σ-types (comprehensions)

```
let veclist =
  fun {a → {(n<:nat, v<:vec a n)}}

let nonempty_veclist =
  fun {a → {(n<:('succ,nat), v<:vec a n)}}
```

Static checking

Our system can statically check bounds:

Peano-naturals

```
'zero                          <:  nat
('succ, ('succ, 'zero))        <:  nat
'zero ⊕ ('succ, ('succ, 'zero))  <:  nat
('succ, nat)                   <:  nat
```

Algebraic data types

```
('cons, ('succ, 'zero), 'nil)  <:  list nat
(('succ, 'zero), 'zero, 'nil)  <:  list' nat
```

Type-level types

```
vec  <:  ⊤ → nat → ⊤
```

Induction

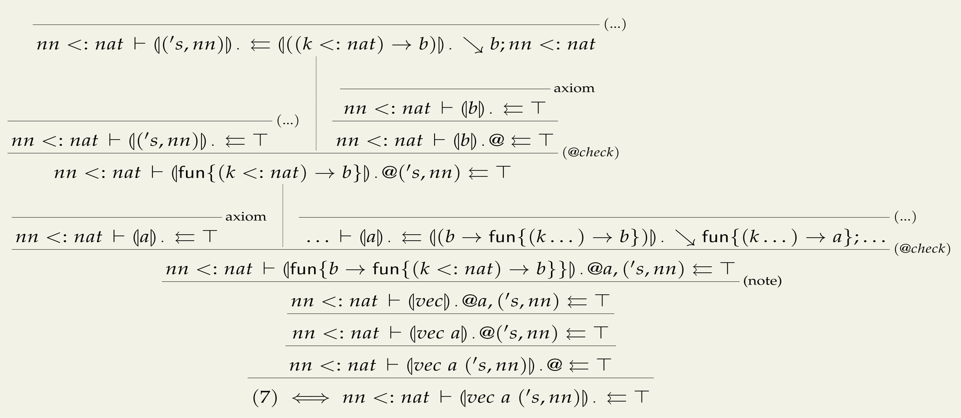
```
fix map = fix map →
  fun {a →
    fun {f →
      fun {'nil          → 'nil;
           ('cons,x,xs)  → ('cons, f x, map a f xs)}}}
  <: fun {a → ((a → a) → list a → list a)}
```

Σ-types

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

Formal checking rules

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



Further reading

- On mereology: Roberto Casati and Achille C. Varzi. *Parts and Places: The Structures of Spatial Representation.* MIT Press, Cambridge, MA, 1999.
- Related type systems: Ulf Norell. *Dependently Typed Programming in Agda.* 2008, `http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`.