Iteration

As a special case of $\mathbb{N}$-elimination, where the family is constant, say at a type P, and the step argument doesn't depend on its first argument, we get the *iterator* over the natural numbers,

$$\mathrm{iter}_{\mathbb{N}}^{P} : P \to (P \to P) \to \mathbb{N} \to P,$$

with judgmental equalities, for $p_0 : P$ and $p_s : P \to P$,

$$\begin{aligned} \mathrm{iter}_{\mathbb{N}}(p_0, p_s, 0) &\equiv p_0 \\ \mathrm{iter}_{\mathbb{N}}(p_0, p_s, \mathrm{succ}\, n) &\equiv p_s(\mathrm{iter}_{\mathbb{N}}(p_0, p_s, n)). \end{aligned}$$

The Ackermann-Péter function

The Ackermann-Péter function $A : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ is "defined" by pattern matching as follows

$$\begin{aligned} A(0, n) &:\equiv \mathrm{succ}\, n \\ A(\mathrm{succ}\, m, 0) &:\equiv A(m, 1) \\ A(\mathrm{succ}\, m, \mathrm{succ}\, n) &:\equiv A(m, A(\mathrm{succ}\, m, n)). \end{aligned}$$

To define this in our type theory, we first define an auxiliary function $B : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$ such that

$$\begin{aligned} B(g, 0) &:\equiv g(1) \\ B(g, \mathrm{succ}\, n) &:\equiv g(B(g, n)). \end{aligned}$$

This is easy: $B :\equiv \lambda g.\, \mathrm{iter}_{\mathbb{N}}(g\, 1, g)$, using the iterator at type $\mathbb{N}$.
Then we define: $A :\equiv \mathrm{iter}_{\mathbb{N}}(\mathrm{succ}, B)$, using the iterator at type $\mathbb{N} \to \mathbb{N}$.

Let us check the judgmental equalities using the computation rules, using that $A\, 0 \equiv \mathrm{succ}$ and $A\, (\mathrm{succ}\, m) \equiv B\, (A\, m) : \mathbb{N} \to \mathbb{N}$,

$$\begin{aligned} A(0, n) &\equiv \mathrm{succ}\, n \\ A(\mathrm{succ}\, m, 0) &\equiv B(A\, m, 0) \equiv A(m, 1) \\ A(\mathrm{succ}\, m, \mathrm{succ}\, n) &\equiv B(A\, m, \mathrm{succ}\, n) \equiv A(m, B(A\, m, n)) \equiv A(m, A(\mathrm{succ}\, m, n)). \end{aligned}$$

We see that the Ackermann-Péter function embodies iterated iteration. It grows faster than any primitive recursive function (i.e., any function that can be defined using only the recursor at type $\mathbb{N}$).

In type theory, all functions terminate, and hence as a programming language it is not Turing complete. But you can run any Turing machine for, say, $A(5, 5)$ many steps, and that's *practically* the same as letting it run forever.

turn

Injectivity of type and term constructors

There was a question yesterday concerning injectivity of type and term constructors, for instance the (local form) rules,

$$\frac{\vdash \prod_{(x:A)} B(x) \equiv \prod_{(x:A')} B'(x)\ \text{type}}{\vdash A \equiv A'\ \text{type}} \qquad \frac{\vdash \prod_{(x:A)} B(x) \equiv \prod_{(x:A')} B'(x)\ \text{type}}{x : A \vdash B(x) \equiv B(x')\ \text{type}}$$

and

$$\frac{\vdash \operatorname{succ} n \equiv \operatorname{succ} m : \mathbb{N}}{\vdash n \equiv m : \mathbb{N}}\ .$$

We'll not dwell on how to establish that these are admissible rules, but we'll refer instead to Streicher [2, Ch. 4] for one approach and Altenkirch-Kaposi [1] for another. The keyword is the *method of logical relations*.

References

[1] Thorsten Altenkirch and Ambrus Kaposi. "Normalisation by evaluation for type theory, in type theory". In: *Logical Methods in Computer Science* 13.4 (2017). Id/No 1, p. 26. DOI: 10.23638/LMCS-13(4:1)2017.

[2] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness and Independence Results*. Progress in Theoretical Computer Science. Birkhäuser, Basel, 1991. DOI: 10.1007/978-1-4612-0433-6.