

Exercise 1 (Church Numerals in System F)

Encode the natural numbers in System F with Church numerals. Use the construction for recursive types from the lecture.

Exercise 2 (Programming in System F)

System F allows us to define functions that go far beyond what was possible in the simply typed λ -calculus. In particular, we can also define some non-primitively recursive functions in System F. As a prominent example, consider the Ackermann function:

$$\begin{aligned}\text{ack } 0 \ n &= n + 1 \\ \text{ack } (m + 1) \ 0 &= \text{ack } m \ 1 \\ \text{ack } (m + 1) \ (n + 1) &= \text{ack } m \ (\text{ack } (m + 1) \ n)\end{aligned}$$

Define the Ackermann function in System F based on the encoding of natural numbers from the last exercise. *Hint*: First define a function g such that $g \ f \ n = f^{n+1} \ \underline{1}$

Exercise 3 (Existential Quantification in System F)

System F can also be defined with additional existential types of the form $\exists \alpha. \tau$. To make use of these types, we add the following constructs to our term language

- `pack` τ with t as τ' ,
- `open` t as τ with m in t' ,

together with the reduction rule:

$$\text{open } (\text{pack } \tau \text{ with } t \text{ as } \exists \alpha. \tau') \text{ as } \alpha \text{ with } m \text{ in } t' \rightarrow t'[\tau/\alpha][t/m]$$

- Specify the typing rules for \exists .
- Show how \exists can be used to specify an abstract module of sets that supports the empty set, insertion, and membership testing.
- Show how to implement this module with lists.
- How do these concepts relate to the SML (or OCaml) concepts of signatures, structures, and functors?

Homework 4 (Finger Exercises on Typing in System F)

a) Give a type τ such that

$$\vdash \lambda m : \text{nat}. \lambda n : \text{nat}. \lambda \alpha. (n (\alpha \rightarrow \alpha)) (m \alpha) : \tau$$

is typeable in System F and prove the typing judgement. Recall that

$$\text{nat} = \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha .$$

b) Is there any typeable term t (in System F) such that if we remove all type annotations and type abstractions from t we get $(\lambda x. x x) (\lambda x. x x)$?

Homework 5 (Programming in System F)

Define (in System F) a function `zero` of type `nat` \rightarrow `bool` that checks whether a given Church numeral is zero. Use the encoding that was introduced in the tutorial.

Homework 6 (Disjunction in System F)

Prove \forall_{I_1} and \forall_E from

$$A \vee B = \forall C. (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$$

in System F. Use pure logic without lambda-terms.

Homework 7 (Progress and Preservation)

We have proved the properties of *progress* and *preservation* for the simply typed λ -calculus. Extend our previous proofs to show that these properties also hold for System F.