Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Harald Fecher, Martin Steffen

SIEGEL

Fool

Sommersemester 2005

Serie 2

April 22, 2005

Thema: λ-calc., type operators (Aufgaben mit Lösungshinweisen)

Ausgabetermin: April 22, 2005

Abgabe:

Aufgabe 1 In Section 3.5 of [**?**], we have seen two ways to generalize subtyping into a more relaxed relation called *matching* (written $\leq_{\#}$) which helps in giving types to objects with binary methods. The two definitions are the *higher-order* interpretation and the *F-bounded* interpretation of $\leq_{\#}$.

Abadi and Cardelli state that there's no much difference in practice between the two definitions. Now, can you find a (perhaps not a real-life) example which makes a difference between the two interpretations? Or show that they are equally expressive?

Lösung: See [**?**, page 33] We can concentrate on just the definition of $\leq_{\#}$, I think.

The thing is very simple. Assume the following two type operators/protocol.

$$S_P(X) = \{m : S_2 \to S_1\} \quad \text{and} \quad T_P(X) = \{m : X \to X\}$$

Then we have

$$\frac{\dfrac{S \leq S_1 \qquad S_2 \leq S}{\{m : S_1 \to S_2\} \leq \{m : S \to S\}}}{S \leq_{\#} T}$$

However, we don't have

$$\forall X.S \leq T_P(X) = \{m : X \to X\}$$

Aufgabe 2 Let's call the "naive" substitution the one that simply replaces the formal parameters, i.e., those mentioned via λ-abstraction, but the actual parameters without taking care of variable capture (also call parasitary binding or dynamic binding).

Consider the following proposal to avoid the capture problem: Take the term, rename all bound variables[1] consistently such that they are all different. Afterwards, use the naive substitution rule for β-reduction. Have you solved the problem of "variable capture". Show this or disprove this.

[1]

Lösung: No, it doesn't work (otherwise the whole α-conversion business would not be that prominent ...) Let us make the problem harder by making it more precise. So we look not for a term that, by reduction, *may* require α-conversion, but one where independently from the reduction strategy *fails* without α-conversion. Basically one needs some term like $\lambda x.f(x,x)$, which duplicates the argument. It is however a bit tricky, so simply applying "$x\ x$" inside some term seems not to work, The basic trick is, as before, to get a term of the form $(\lambda\tilde{x}.\lambda\tilde{y}.\tilde{x})\ \tilde{y}$. Let's call this term t_{true}. Now we get:

$$\begin{aligned}(\lambda b.\lambda x'.\lambda y'.b\ (b\ x'\ y'))\ t_{true}\ x\ y &\rightarrow_\beta\\ (\lambda x'.\lambda y'.t_{true}\ (t_{true}\ x'\ y'))\ x\ y &\rightarrow^*_\beta\\ t_{true}(t_{true}\ x\ y)) &\rightarrow^*_\beta\\ t_{true}\ y &\rightarrow^*_\beta\end{aligned}$$

Note further

$$(\lambda x.x(xz_1z_2))\ (\lambda x.\lambda z.x)$$

□

Aufgabe 3 (β-reduction) As discussed in the lecture, the fundamental computation mechanism in the λ-calculus is called β-reduction. It is based on a "replacement" or substitution mechanism. We write $t\{\!|s/x|\!\}$ for the term where s replaces x in t. Formalize more rigorously the notion of β-reduction and substitution.

Aufgabe 4 (S) Consider the following "program" (i.e., term)

$$S \triangleq \lambda z(\lambda y.(\lambda x((zx)(yx)))$$

What is the intuitive meaning of this term?

Lösung: One could try to "type" it to see what it means.

$$S \triangleq \lambda f : T \rightarrow S \rightarrow U.(\lambda g : T \rightarrow S.(\lambda x : T.(fx)\ (gx)))$$

The overall type of S is given by

$$S : (T \rightarrow S \rightarrow U) \rightarrow (T \rightarrow S) \rightarrow (T \rightarrow U)$$

Aufgabe 5 (Confluence) "Is *Java* confluent?" Well, this is not a formal question for instance since we have not formalized *Java* (later we will have formal object calculi where one could pose the question formally), so discuss issues around confluence in a language like *Java*.

Lösung: In absence of multithreading, the answer is yes for a trivial reason: since the evaluation strategy is determinate. With multithreading the answer is no.

Aufgabe 6 (Church encoding) In the lecture we have shown a way to represent the natural numbers as the most simple unbounded data structure into the untyped λ-calculus. This encoding is known as the *Church numerals.* The Church numeral corresponding to the natural number n is

$$\lambda z.\lambda s.s^n(z)\ , \tag{1}$$

where $s^n(z)$ abbreviates $s(s(\ldots s(z)\ldots))$. Define the addition on the natural numbers.

Lösung: Well, for me it's simpler to think in a (powerfully enough) typed fragment. The addition function is of the type. Well, if we would like to *define* `Nat`, one can set

$$\texttt{Nat} \triangleq \forall X.X \to (X \to X) \to X \ .$$

So, the **0** is given by $\Lambda X.\lambda z : \texttt{Nat}.\lambda s : \texttt{Nat}.z$. The natural number **n** is given by

$$\mathbf{n} : \texttt{Nat} \triangleq \Lambda X.\lambda z : \texttt{Nat}.\lambda s : \texttt{Nat} \to \texttt{Nat}.s^n(z) \tag{2}$$

It's easy to check that **n** is indeed of type `Nat`. We leave out in most cases the Λ-part.

In order to define the addition, it's wise to start with the successor function. Why? Well, addition intuitively *is* defined via the successor function (by induction, by iteration), so we should expect this also here. The type of the successor function is given as follows:

$$add : \texttt{Nat} \to \texttt{Nat} \ .$$

$$succ(n : \texttt{Nat}) \triangleq \lambda z : \texttt{Nat}.\lambda s : \texttt{Nat} \to \texttt{Nat}.(n(sz)s) \tag{3}$$

Let's see whether it works:

$$\begin{array}{rcll} succ\ \mathbf{n} & = & \lambda z.\lambda s.\mathbf{n}(sz)s & = \\ & & \lambda z.\lambda s.\ (\lambda \tilde{z}.\lambda \tilde{s}.\tilde{s}^n\tilde{z})\ (sz)s & \to_\beta \\ & & \lambda z.\lambda s.\ (\lambda \tilde{s}.\tilde{s}^n(sz))\ s & \to_\beta \\ & & \lambda z.\lambda s.\ (s^n(sz))\ s & = \\ & & \lambda z.\lambda s.\ s^{n+1}z & = \mathbf{n+1} \end{array}$$

Well, fine. Now what is the underlying idea of the thing? Well, somehow it's straightforward. The number n corresponds to a function, that takes some value for z (which represents 0) and a function value which represents the successor function. The *succ*-function that we defined simply does not start from z but from sz. How does this observation help to define the addition? Not much, at least we need to rethink what the definition of *succ* "means". Let's look again at Equation **??**. At first sight, the construction seems like some more or less meaningless term which has a "weight of n", so in some sense simply counts. It's a very rudimentary counting system, not a fancy decimal or binary representation of the numerals, but somehow a "unary" representation. Starting in a language where the is "no data, only functions" it seems like some obscure "hack" gives us the desired "data".

But that's not all of it. Looking at **n** as a function (which it is), the function takes two arguments and applies the second argument n times to the first. So, it's not just that **n** is some sort of representation of the number n, it is an n-fold *iterator* of a function onto a start-value! Seen from that angle, it looks a much less arbitrary encoding. Actually, the encoding captures the *essence* of a natural number, which is: the essence of a natural number is that you can use it for iteration, or even harder:

> the natural number n *is* the n-fold iteration!

So one can intepret the successor function very simply as: "you obtain the successor of n by iterate the successor function n times when starting at 1". That's a quite direct translation of Equation (**??**).

Well, if natural numbers are iterators, this gives us a grip onto the definition of addition: what is it else than iterated successor. Now, addition is the iterated successor. Let's define

addition by induction, which might help us for the definition using Church's trick. We use induction in the first argument:

$$\begin{array}{rcl} 0+m & = & m \\ (n+1)+m & = & (n+m)+1 \end{array}$$

Well, that's not iteration, sure. What about the following:

$$n + m = succ^n\ m$$

Now we get to something. We argued above that n simply is iteration, well, now we can make use of it:[2]

$$add\ x\ y \triangleq x\ y\ succ$$

And now we check against the above inductive definition. We must of course use the first argument for induction, because we iterated over that one.

$$\begin{array}{rcll} (\mathbf{n+1})+\mathbf{m} & \triangleq & (\mathbf{n+1})\ \ \mathbf{m}\ succ & = & \\ & & (\lambda z.\lambda s.s^{n+1}z)\ \mathbf{m}\ succ & \rightarrow_\beta & \\ & & succ^{n+1}\ \mathbf{m} & = & \text{(by definition)} \\ & & succ(succ^n\ \mathbf{m}) & = & \text{(by induction)} \\ & & succ(\mathbf{m}+\mathbf{n}) & & \end{array}$$

Voilà.

Aufgabe 7 (Fixpoint operator) A fixpoint a of a function (very generally) is a value a such that $f(a) = a$. The untyped λ-calculus[3] is powerful enough that one can *define* a fixpoint operator or fixpoint combinator, i.e., a function which, given a function, returns (one of) its fixpoints. Indeed, the possibility to encode a fixpoint operator lies at the heart of the expressiveness of the untyped λ calculus, rendering it Turing complete.[4]

There are two more well-known encodings of fixpoint operators, the one of Curry and the one of Turing. They look as follows

$$Y_c \triangleq \lambda y.VV \quad \text{where} \quad V = \lambda x.y(xx) \tag{4}$$

$$Y_t \triangleq ZZ \quad \text{where} \quad Z = \lambda y.\lambda x.x(yyx) \tag{5}$$

Show that they are indeed fixpoint combinators. Does even $Yy \rightarrow_\beta^* y(Yy)$ hold?

Talking about fixpoint operators, Church numerals and the fact that "everything is a function": what is the fixpoint of "5" (or "0")?

Lösung: Let's start with the fixpoint of 0. Let's take the encoding of Exercise **??**, i.e., the one of Equation (**??**), and Church's fixpoint operator.

[2] We use freely infix notation and other conveniences.

[3] for various typed versions, the issue has to be re-addressed.

[4] Think about it: the fixpoint combinator in a non-functional setting corresponds to a looping constructs, so it means that you can *program* a while-loop.

$$\begin{array}{rcll} Y0 & = & (\lambda y.(\lambda x.y(xx))\ (\lambda x.y(xx)))0 & \to_\beta \\ & & (\lambda x.0(xx))\ (\lambda x.0(xx)) & = \\ & & V_0V_0 & = \\ & & (\lambda x.0(xx))\ V_0 & \to_\beta \\ & & 0(V_0\ V_0) & \to_\beta^* \\ & & 0(0(V_0V_0)) & \to_\beta^* \\ & & 0^n(V_0V_0) & \to_\beta^\omega \\ & & 0^\omega & \end{array}$$

The last "reduction step" is of course meant informally. Anyway, the fixpoint diverges. Note that the fact that $Y_c\ 0$ diverges does not depend on the reduction strategy.

The diverging behavior is somehow strange, even if we interpret 0 as "zero-time iterator". Actually, the representation as given in Equation **??** is not the only possible, we might as well swap the order of the arguments and use

$$n : \mathtt{Nat} \triangleq \Lambda X.\lambda s : \mathtt{Nat} \to \mathtt{Nat}.\lambda z : \mathtt{Nat}.s^n(z) \tag{6}$$

With this choice of representation we obtain:

$$\begin{array}{rcll} Y0 & = & (\lambda y.(\lambda x.y(xx))\ (\lambda x.y(xx)))0 & \to_\beta \\ & & (\lambda x.0(xx))\ (\lambda x.0(xx)) & \to_\beta^* \\ & & (\lambda x.\lambda z.z)\ (\lambda x.\lambda z.z) & \to_\beta \\ & & \lambda z.z & = id \end{array}$$

So the rather arbitrary choice of representation leads to two rather different answers to the question of "what is the fixpoint of 0?". Think also of that: the two representations of 0 can be interpreted also as the Church encodings of true and false: the first projection $\lambda t.\lambda f.t$ corredponds to true, and the second $\lambda t.\lambda f.f$ represents false.[5]

Let us try also Turing's fixpoint operator.

$$\begin{array}{rcll} Y0 & = & (ZZ)0 & = \\ & & ((\lambda y.\lambda x.x(yyx))(\lambda y.\lambda x.x(yyx)))0 & \to_\beta \\ & & (\lambda x.x(ZZx))0 & \to_\beta \\ & & 0(ZZ0) & \end{array}$$

This means we have $(ZZ0) \to_\beta^* 0(ZZ0)$, which means we may continue in this manner to $0^n(ZZ0)$ and thus *diverge.* Using the definition of 0 from Equation (**??**), we *may* also continue by $0(ZZ0) \to_\beta \lambda z.z$, which is a normal form. □

[5]Obviously, which term corresponds to which interpretation is a matter of convention.