15-312 Lecture on Untyped Languages

Untyped Languages

We concentrate on untyped languages, languages that do not make use of types and therefore for which there is no static semantics. We will see two of them:

- The untyped λ-calculus is a minimal language of untyped functions. Because everything is a function there, every closed expression is a well-formed program that will either evaluate to a value, or fail to terminate. This language is Turing-complete, which means that any computable function can be written in it, exactly as C, Java or ML. It is however of purely theoretical interest because of its extremely low level of abstraction (kind of like programming with bits).
- The untyped PCF is what its name says: PCF without types. Because there are now two classes of entities (functions and numbers) and there are no types to sort them out, we must now check at run time that the arguments of an operator are used consistently (e.g., to prevent taking the successor of a function). These dynamic checks must be there in any untyped language that has more than one class of expressions. Untyped PCF is indeed a simplified version of Scheme, one of the most prominent untyped languages.

The difference between how to use a typed and an untyped language is evident in figure 1: all the checks that the typechecker does ahead of execution in a typed language must be performed during evaluation. Therefore, what the typechecker flags as type errors before evaluation now become run-time errors produced by the evaluator. Note that one still needs to prove a progress theorem to make sure that evaluation never gets stuck.

The Untyped Lambda-Calculus

The untyped λ-calculus provides only functions and application:

$$\textit{Untyped expressions} \qquad u \; ::= \; x \mid \texttt{lam}(x.u) \mid \texttt{app}(u_1, u_2)$$

For clarity, we write operators in our untyped languages using a `teletype` font and use the letter u instead of e for expressions.

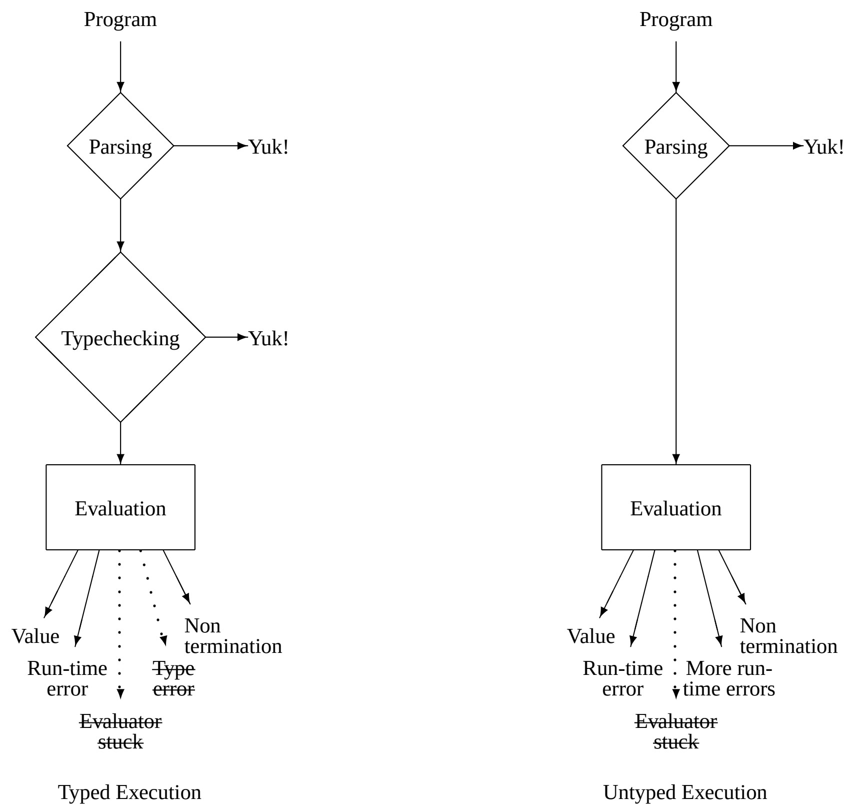

Figure 1: The Evaluation Process for Typed and Untyped Languages

The dynamic semantics of this language is as in the typed case. Chapter 22 of Harper's book goes into the details of its surprising expressiveness.

What we will do now is define a translation of the untyped λ-calculus into a typed functional language. Given an untyped expression u, we will build an expression $e = \ulcorner u \urcorner$ in this typed language so that if syntactically well formed u, then $\ulcorner u \urcorner$ is well-typed of some type D.

Because the untyped λ-calculus only provides functions, D must be an arrow type, and since both the argument and the result must be functions (and there is nowhere to start from), it makes sense to define it as the recursive type of all functions where domain and range coincide:

$$D \quad = \mu t.\, t \to t$$

This means that our typed language must contain functions and recursion.

With a type in hands, let's define the translation of the expression in the untyped λ-calculus. The one constraint that we must maintain is that for every u, its translation

$\ulcorner u \urcorner$ has type D. We obtain:

$$\begin{array}{lcl} \ulcorner x \urcorner & = & x \\ \ulcorner \texttt{lam}(x.u) \urcorner & = & \mathsf{fold}[D](\mathsf{lam}(x.\ulcorner u \urcorner)) \\ \ulcorner \texttt{app}(u_1, u_2) \urcorner & = & \mathsf{app}(\mathsf{unfold}(\ulcorner u_1 \urcorner), \ulcorner u_2 \urcorner) \end{array}$$

Let's convince ourselves that this makes sense:

- All variables will have type D.
- Let's look at translation of $\texttt{lam}(x.u)$ and check the types:

$$\underbrace{\mathsf{fold}[D](\underbrace{\mathsf{lam}(\underbrace{x}_{D}.\underbrace{\ulcorner u \urcorner}_{D})}_{D \to D})}_{D}$$

- Finally, let's look at the translation of $\texttt{app}(u_1, u_2)$:

$$\underbrace{\mathsf{app}(\underbrace{\mathsf{unfold}(\underbrace{\ulcorner u_1 \urcorner}_{D})_{D \to D}, \underbrace{\ulcorner u_2 \urcorner}_{D}}})}_{D}$$

Untyped PCF

In the untyped λ-calculus, things were easy since we only needed to deal with functions. Let's introduce numbers and see what happens. If we do so, we obtain for example an untyped version of PCF. Now things can go wrong, for example if we try to take the successor of a function. Therefore, we will also introduce an error token.

$$\begin{array}{llcl} \textit{Untyped expressions} & u & ::= & x \mid \texttt{lam}(x.u) \mid \texttt{app}(u_1, u_2) \\ & & \mid & \texttt{fix}(x.u) \\ & & \mid & \texttt{num}[n] \mid \texttt{s}(u) \mid \texttt{ifz}(u, u_0, x.u_n) \\ & & \mid & \texttt{error} \\ \textit{Numbers} & n & ::= & \texttt{z} \mid \texttt{s}(n) \end{array}$$

Again, operators in the untyped language are written in a `teletype` font. Here, n stands for numbers in unary notation.

The dynamic semantics of this languages is as follows:

Values

$$\frac{}{\texttt{num}(n)\ \mathsf{val}} \qquad \frac{}{\texttt{lam}(x.u)\ \mathsf{val}}$$

Error

$$\frac{}{\texttt{error}\ \mathsf{err}}$$

Tests

$$\frac{}{\mathtt{num}(n) \ \mathsf{isnum}\ n} \qquad \frac{}{\mathtt{lam}(x.u) \ \mathsf{isntnum}} \qquad \frac{}{\mathtt{error} \ \mathsf{isntnum}}$$

$$\frac{}{\mathtt{lam}(x.u) \ \mathsf{isfun}\ x.u} \qquad \frac{}{\mathtt{num}(n) \ \mathsf{isntfun}} \qquad \frac{}{\mathtt{error} \ \mathsf{isntfun}}$$

Transitions

$$\frac{u \mapsto u'}{\mathtt{s}(u) \mapsto \mathtt{s}(u')} \qquad \frac{u \ \mathsf{isnum}\ n}{\mathtt{s}(u) \mapsto \mathtt{num}(sn)}\,* \qquad \frac{u \ \mathsf{isntnum}}{\mathtt{s}(u) \mapsto \mathtt{error}}\,*$$

$$\frac{u \mapsto u'}{\mathtt{ifz}(u, u_0, x.u_n) \mapsto \mathtt{ifz}(u', u_0, x.u_n)} \qquad \frac{u \ \mathsf{isntnum}}{\mathtt{ifz}(u, u_0, x.u_n) \mapsto \mathtt{error}}\,*$$

$$\frac{u \ \mathsf{isnum}\ \mathsf{z}}{\mathtt{ifz}(u, u_0, x.u_n) \mapsto u_0}\,* \qquad \frac{u \ \mathsf{isnum}\ \mathsf{s}(n)}{\mathtt{ifz}(u, u_0, x.u_n) \mapsto [n/x]u_n}\,*$$

$$\frac{u_1 \mapsto u_1'}{\mathtt{app}(u_1, u_2) \mapsto \mathtt{app}(u_1', u_2)} \qquad \frac{u_1 \ \mathsf{val} \quad u_2 \mapsto u_2'}{\mathtt{app}(u_1, u_2) \mapsto \mathtt{app}(u_1, u_2')}$$

$$\frac{u_1 \ \mathsf{isfun}\ x.u \quad u_2 \ \mathsf{val}}{\mathtt{app}(u_1, u_2) \mapsto [u_2/x]u}\,* \qquad \frac{u_1 \ \mathsf{isntfun} \quad u_2 \ \mathsf{val}}{\mathtt{app}(u_1, u_2) \mapsto \mathtt{error}}\,*$$

$$\frac{}{\mathtt{fix}(x.u) \mapsto [\mathtt{fix}(x.u)/x]u}$$

Note that in rules marked with $*$ we need to perform a run-time test to make sure that some operand belongs to the right class before proceeding. If it doesn't, we produce an error.

Typed Representation

In the same way that we translated every valid expression in the untyped λ-calculus into well-typed expressions of type D in a typed functional language with recursive types, we will now translate the untyped PCF into an appropriate variant of PCF, making sure that valid untyped programs are mapped to well-typed PCF expressions, and also exposing all the tests that get performed during execution as well as the marking that needs to be done to enable these tests.

Let's identify our target language. We have two options: go the way of the untyped λ-calculus and use recursive types, or come up with an ad-hoc extension of PCF that works fine for our purposes. We will go the second way noting that Harper's book shows how to reduce it to the first option (so it is not all that ad-hoc after all).

The grammar for this extension of PCF is as follows:

$$\begin{array}{lrcl}
\textit{Types} & \tau & ::= & \mathsf{nat} \mid \tau_1 \to \tau_2 \mid \mathsf{dyn} \\
\textit{Expressions} & e & ::= & x \mid \mathsf{lam}[\tau](x.e) \mid \mathsf{app}(e_1, e_2) \\
 & & \mid & \mathsf{fix}(x.e) \\
 & & \mid & \mathsf{z} \mid \mathsf{s}(e) \mid \mathsf{ifz}(e, e_0, x.e_n) \\
 & & \mid & \mathsf{error} \\
 & & \mid & g\,!\,e \mid e?g \\
\textit{Tags} & g & ::= & \mathsf{num} \mid \mathsf{fun}
\end{array}$$

The extension consists of the type dyn, which will the representation target type of untyped expressions. The constructor of this type is $g\,!\,e$, where g is a *tag* (either num for numbers or fun for functions). Intuitively, $g\,!\,e$ tags expression e with tag g, supposedly related to the type of e, and ascribes to it the type dyn. Dually, $e?g$ checks that the tag of e matches g and returns error otherwise. Since we want to simulate untyped evaluation, we need to handle errors.

Note also that we cannot build an expression of type dyn without going through the tagging construct (or assume that a variable has type dyn).

Static Semantics

Standard PCF Rules

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \mathsf{lam}[\tau](e) : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \mathsf{app}(e_1, e_2) : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{s}(e) : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_n : \tau}{\Gamma \vdash \mathsf{ifz}(e, e_0, x.e_n) : \tau}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}(x.e) : \tau}$$

Additional Rules

$$\frac{}{\Gamma \vdash \mathsf{error} : \tau}$$

Since an error can arise at any point in the evaluation of an expression, error must be allowed to have any type if we want type preservation to hold.

$$\frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{num}\,!\,e : \mathsf{dyn}} \qquad \frac{\Gamma \vdash e : \mathsf{dyn} \to \mathsf{dyn}}{\Gamma \vdash \mathsf{fun}\,!\,e : \mathsf{dyn}}$$

The tagging construct $g\,!\,e$ takes an expression e of some definite type, turns it into an expression of type dyn, but along the way makes a note of the original type in the tag.

$$\frac{\Gamma \vdash e : \mathsf{dyn}}{\Gamma \vdash e?\mathsf{num} : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{dyn}}{\Gamma \vdash e?\mathsf{fun} : \mathsf{dyn} \to \mathsf{dyn}}$$

By contrast, $e?g$ takes an expression of type dyn and casts it to an expression of a definite type, again marking what it has done along the way. At evaluation time, e will be tagged with a tag g': if g and g' are the same tag all is fine, otherwise, the execution will result into an error. Note that the type system does not force tags to be used consistently: after all, we are trying to model untyped languages, in which all kinds of wacky expressions can be sent to the evaluator.

Transition Semantics

Values

$$\frac{}{\mathsf{z}\ \mathsf{val}} \qquad \frac{v\ \mathsf{val}}{\mathsf{s}(v)\ \mathsf{val}} \qquad \frac{}{\mathsf{lam}(x.e)\ \mathsf{val}} \qquad \frac{v\ \mathsf{val}}{g\,!\,v\ \mathsf{val}}$$

Standard PCF Transitions

$$\frac{e \mapsto e'}{\mathsf{s}(e) \mapsto \mathsf{s}(e')} \qquad \frac{e \mapsto e'}{\mathsf{ifz}(e, e_0, x.e_n) \mapsto \mathsf{ifz}(e', e_0, x.e_n)}$$

$$\frac{}{\mathsf{ifz}(\mathsf{z}, e_0, x.e_n) \mapsto e_0} \qquad \frac{n\ \mathsf{val}}{\mathsf{ifz}(\mathsf{s}(n), e_0, x.e_n) \mapsto [v/x]e_n}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{app}(e_1, e_2) \mapsto \mathsf{app}(e_1', e_2)} \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathsf{app}(e_1, e_2) \mapsto \mathsf{app}(e_1, e_2')}$$

$$\frac{e_2\ \mathsf{val}}{\mathsf{app}(\mathsf{lam}[\tau]e, e_2) \mapsto [e_2/x]e} \qquad \frac{}{\mathsf{fix}(x.e) \mapsto [\mathsf{fix}(x.e)/x]e}$$

Additional Transitions

$$\frac{e \mapsto e'}{g\,!\,e \mapsto g\,!\,e'} \qquad \frac{e \mapsto e'}{e?g \mapsto e'?g} \qquad \frac{}{(g\,!\,v)?g \mapsto v} \qquad \frac{g \neq g'}{(g'\,!\,v)?g \mapsto \mathsf{error}}$$

Both tagging and checking evaluate their argument until possible. When they meet, the check for a tag g must find an expression with the same tag, otherwise an error is returned. Note that this is the only place where run-time errors can arise in this language, i.e., we have isolated the possibility of error in exactly one rule.

Compilation

We will now compile the untyped PCF to the extended PCF just introduced. The invariant that we want to maintain is that whenever u is a valid expression in the untyped language, then $e = \ulcorner u \urcorner$ is well-typed of type dyn in the extended PCF.

$$\begin{array}{lcl}
\ulcorner x \urcorner & = & x \\
\ulcorner \mathtt{num}(n) \urcorner & = & \mathsf{num}\,!\,n \\
\ulcorner s \urcorner & = & \mathsf{num}\,!\,(\mathsf{s}(u?\mathsf{num})) \\
\ulcorner \mathtt{ifz}(u, u_0, x.u_n) \urcorner & = & \mathsf{ifz}(\ulcorner u \urcorner?\mathsf{num}, \ulcorner u_0 \urcorner, x'.[\mathsf{num}\,!\,x'/x]\ulcorner u_n \urcorner) \\
\ulcorner \mathtt{lam}(x.u) \urcorner & = & \mathsf{fun}\,!\,\mathsf{lam}[\mathsf{dyn}](x.\ulcorner u \urcorner) \\
\ulcorner \mathtt{app}(u_1, u_2) \urcorner & = & \mathsf{app}(\ulcorner u_1 \urcorner?\mathsf{fun}, \ulcorner u_2 \urcorner) \\
\ulcorner \mathtt{fix}(x.u) \urcorner & = & \mathsf{fix}(x.\ulcorner u \urcorner)
\end{array}$$

Here, all variables are of type dyn. When compiling the successor of an untyped expression, we must check that the operand is a number, but we also need to announce that the result will be a number. Similarly, we need to check that the first argument of ifz is a number. What is happening about the third argument of ifz is more interesting. Untyped variables are assigned type dyn, therefore, x will have type dyn. However, the binder of the third argument of ifz has type nat. Therefore, we need to replace x with a new variable, x' of type nat, and tag every occurrence in $\ulcorner u_n \urcorner$ with num. Every function is explicitly tagged as such, and every application checks that its first argument is a function.

It easy easy to convince oneself that if u is a closed untyped expression, then $\ulcorner u \urcorner$ has type dyn. Moreover, this translation is sound and complete in the sense that whenever an untyped expression u evaluates to some untyped value u', then $\ulcorner u \urcorner$ evaluates to $\ulcorner u' \urcorner$ (soundness), and vice-versa, if e is the extended PCF representation of some untyped expression u and e evaluates to a value v, then v is the representation of an untyped value u' (completeness).

Lemma 1 (Soundness) *If $u \mapsto^* u'$, then $\ulcorner u \urcorner \mapsto^* \ulcorner u' \urcorner$.*

Lemma 2 (Completeness) *If $\ulcorner u \urcorner \mapsto^* v$ and v val, then $v = \ulcorner u' \urcorner$ for u' val and $u \mapsto^* u'$.*

Example

We have seen that addition can be defined as follows in PCF:

$$\begin{array}{l}
\mathsf{fix}[\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}](plus.\,\mathsf{lam}[\mathsf{nat}](m.\,\mathsf{lam}[\mathsf{nat}](n. \\
\quad \mathsf{ifz}(m, \\
\qquad n, \\
\qquad m'.\,\mathsf{s}(\mathsf{app}(\mathsf{app}(plus, n), m'))))))
\end{array}$$

(One can write it in other ways also.)

The untyped version then appears as:

$$\begin{array}{l}
\mathtt{fix}(plus.\,\mathtt{lam}(m.\,\mathtt{lam}(n. \\
\qquad\quad \mathtt{ifz}(m, \\
\qquad\qquad n, \\
\qquad\qquad m'.\,\mathtt{s}(\mathtt{app}(\mathtt{app}(plus, n), m'))))))
\end{array}$$

and translating it to our extended PCF, we obtain:

$$\begin{array}{l}\mathsf{fix}[\mathsf{dyn}](\mathit{plus}.\,\mathsf{fun}\,!\,\mathsf{lam}[\mathsf{dyn}](m.\,\mathsf{fun}\,!\,\mathsf{lam}[\mathsf{dyn}](n.\\ \quad \mathsf{ifz}(m?\mathsf{num},\\ \qquad n,\\ \qquad m'.\,\mathsf{num}\,!\,\mathsf{s}(\mathsf{app}(\mathsf{app}(\mathit{plus}?\mathsf{fun}, n)?\mathsf{fun}, \mathsf{num}\,!\,m'))?\mathsf{num}))))\end{array}$$

This expression, which just makes explicit what the untyped version does, contains 4 tagging operations and 4 checks. What is even worse, they all occur within the recursion, which means that on each recursive call, 4 tags are created and 4 are checked. If use it to compute $100 + 100$, that's 400 tags created and 400 tags checked.

It may appear that none of the checks involve n. This is the case only if n is z: otherwise, the last iteration will make sure that n is a number before taking its successor.

Type-Directed Optimization

Having made tag creation and checking explicit in a typed framework provides a way to correctly transform programs to improve their efficiency. This is called type-directed optimization and it has been applied in much wider settings than this. We will demonstrate it on the expression for addition. We will proceed by example rather than through a rigorous definition of the transformation and its proof of correctness (which could be done with limited effort).

Let's examine what all those tag creations and checks actually do. The two fun tag markings state that the following λ-abstractions are functions — that sounds obvious! They are there so that the two checks in the applications succeed. This is inefficient in a major way because, since plus is defined to be a binary function, it will be a binary function at every recursive invocation. A first optimization is therefore to pull the tagging of this code as a function out of the recursive call:

$$\begin{array}{l}\mathsf{fun}\,!\,\mathsf{fun}\,!\,\mathsf{fix}[\mathsf{dyn} \to \mathsf{dyn} \to \mathsf{dyn}](\mathit{plus}.\,\mathsf{lam}[\mathsf{dyn}](m.\,\mathsf{lam}[\mathsf{dyn}](n.\\ \qquad\qquad \mathsf{ifz}(m?\mathsf{num},\\ \qquad\qquad\quad n,\\ \qquad\qquad\quad m'.\,\mathsf{num}\,!\,\mathsf{s}(\mathsf{app}(\mathsf{app}(\mathit{plus}, n), \mathsf{num}\,!\,m'))?\mathsf{num}))))\end{array}$$

Note that the checks plus is a function have been removed from the recursive call, but any attempt to use this definition to add two numbers will need to rely those checks: they have been eliminated inside the recursion because it is safe to do so, but we don't know about what will happen outside, so they must still be there. One effect of moving the tagging operations outside the loop is that the type of the recursor is now $\mathsf{dyn} \to \mathsf{dyn} \to \mathsf{dyn}$.

This looks (and runs) better already! Next, m is checked as a number at every recursion and its predecessor m' is tagged as such. This is pointless: if m is a number, then m' is a number, and it is this very m' that will be used as m in the next recursive call. What we would like to do, is to check that m is a number even before any recursive call. We can do so by means of an operation called *η-expansion*: it says that if we know that e is a function, then it can be expanded as $\mathsf{lam}(x.\,\mathsf{app}(e, x))$. By doing so, we can rewrite our latest version to:

$$\begin{array}{l}
\mathsf{fun}\,!\,\mathsf{lam}[\mathsf{dyn}](m_\eta.\\
\quad \mathsf{app}(\mathsf{fun}\,!\,\mathsf{fix}[\mathsf{nat} \to \mathsf{dyn} \to \mathsf{dyn}](plus.\,\mathsf{lam}[\mathsf{nat}](m.\,\mathsf{lam}[\mathsf{dyn}](n.\\
\qquad\qquad\qquad\qquad \mathsf{ifz}(m,\\
\qquad\qquad\qquad\qquad\quad n,\\
\qquad\qquad\qquad\qquad\quad m'.\,\mathsf{num}\,!\,\mathsf{s}(\mathsf{app}(\mathsf{app}(plus, n), m'))?\mathsf{num}))),\\
\qquad\quad m_\eta?\mathsf{num}))
\end{array}$$

By η-expanding our function on its first argument, we can check that m (now renamed m_η for clarity) is a number exactly once. We do not need to play the tag and check game within the loop. Note that this has the effect of changing the type of the recursive function from $\mathsf{dyn} \to \mathsf{dyn} \to \mathsf{dyn}$ to $\mathsf{nat} \to \mathsf{dyn} \to \mathsf{dyn}$. The cast on m_η makes sure that the type of the overall expression is still $\mathsf{dyn} \to \mathsf{dyn} \to \mathsf{dyn}$.

Another apparently pointless tag and check still remains in the "else" branch of the ifz: as it checks that the recursive calls returns a number, takes its successor, and then tags it as a number before returning. But the previous recursive call will do the same! Can we just eliminate both the tagging and the testing? Not quite. At least not quite this simply: if we remove these operations, the overall type of the "else" branch will be nat, which requires the "then" branch to be nat; an easy way to do so is to check that n is a number. If we do so, we change the semantics of this function since we saw that, if $m = \mathsf{z}$, then anything can be given as n, even an expression that is not a number, and the call will be successful, returning n unaltered. One way to overcome the recursive tag and test, is to cascade two ifz, the outermost returning values of type dyn and the inner one doing most of the work computing values of type nat. But that's complicated.

What if we knew that n must be a number? We could then η-expand it out like we did for m:

$$\begin{array}{l}
\mathsf{fun}\,!\,\mathsf{lam}[\mathsf{dyn}](n_\eta.\,\mathsf{fun}\,!\,\mathsf{lam}[\mathsf{dyn}](m_\eta.\\
\quad \mathsf{num}\,!\,\mathsf{app}(\mathsf{app}(\mathsf{fun}\,!\,\mathsf{fix}[\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}](plus.\,\mathsf{lam}[\mathsf{nat}](m.\,\mathsf{lam}[\mathsf{nat}](n.\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ifz}(m,\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad n,\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad m'.\,\mathsf{s}(\mathsf{app}(\mathsf{app}(plus, n), m')))))),\\
\qquad\qquad\quad n_\eta?\mathsf{num}), m_\eta?\mathsf{num}))
\end{array}$$

Note how we changed the type of the recursor from $\mathsf{nat} \to \mathsf{dyn} \to \mathsf{dyn}$ to $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ and how we now need to cast the result to a number for whatever computation uses the result of calling this function. More importantly, observe that the recursive definition is not only pure PCF code, but it is the very PCF expression we started with.