Last updated: February 15, 2024 at 5:46pm

The operational semantics of λ^+ contains a rather strange-looking rule:

$$\frac{e[f \mapsto \mathsf{fix}\, f \text{ is } e] \Downarrow v}{\mathsf{fix}\, f \text{ is } e \Downarrow v}\ \textsc{Fix}$$

In lectures, we saw that fix is responsible for giving meaning to recursive functions. However, this doesn't explain *why* the FIX rule is defined the way it is. In this section, we will explore the motivation behind the FIX rule.

1. Recursive "Definitions"

People often say they "define" recursive functions. For example, below is the "definition" of the factorial function in λ^+:

```
fun rec fact with n =
    if n = 0 then 1
    else n * fact (n-1)
in ..
```

However, recursive "definitions" are *not* definitions in the strict sense: a true definition cannot be circular. For example, if you open a dictionary to look up some word, say "apple", you may see the following proper definition:

> **apple** (noun): the round fruit of a tree of the rose family, which typically has thin red or green skin and crisp flesh.

It is a proper definition, because the meaning of "apple" is given in terms of other words that are already defined. In contrast, consider the following improper "definition":

> **apple** (noun): a round fruit which is called apple.

If you find this in a dictionary, you should probably get a refund immediately.

The same can be said about the recursive "definition" of the factorial function, because the meaning of `fact` is given in terms of `fact` itself. This is a circular definition, and hence not a proper definition.

What we really mean when we say we "define" a recursive function is that we are giving a *equational specification* of the function we're trying to define. We're really saying that:

- Let `fact` be an arbitrary function.
- It must be the case that `fact` satisfies the mathematical equation:

  ```
  fact == lambda n. if n = 0 then 1 else n * fact (n-1)
  ```

- Given the above constraint, solve for `fact`.

Note that in an equation, a variable such as `fact` can appear on both sides of the equation. This is not a problem in mathematics, because we can solve for the unknown variable by using

the equation as a constraint. This is similar to how you might solve systems of equations in algebra. For example,

- Let x be an arbitrary number.
- It must be the case that $x = 2x + 1$.
- Given the above constraint, solve for x.

Indeed, we can solve the equation and find that $x = -1$. Similarly, when we're "defining" recursive functions, we're actually asking the programming language to solve for some unknown function that satisfies the equation we wrote down.

In math, when we "define" a function via some equation that contains the function on both sides of the equation, we say that the equation is an recurrence relation. Unfortunately, recurrence relations can be solved algorithmically only in a few special cases. In the general case, there is provably no algorithm that can solve arbitrary recurrence relations.

In other words, there is zero chance that, if we as the programmers write down an equational specification of a function, the programming language can automatically give us a concrete function that satisfies the equation by telling us what the output is for any input. However, as programmers, we use recursion in our program all the time! How can this be?

> **Paradox:** There is no algorithm that can solve arbitrary recurrence relations, but we use recursion in our programs all the time.

One solution to this paradox is that, our programming languages "fake it till they make it".

Specifically, they "pretend" that they can solve the recurrence relation for us. Instead of giving us the correct answer, they give us "fake" answers which are approximations to the correct answer. However, the "fake" answers are so good that realistically, we can never tell the difference between the fake and the real answers.

In our programs, those fake objects are manufactured by what we call the *fixed-point operator*. Whenever "recursively-defined" function is called, the programming language walks to the fixed-point operator, says "hey, give me a good-enough fake object", and proceeds to freely use the fake object, rather than belaboring itself with the impossible task of finding an actual solution to the recurrence relation.

It's ok if the above high-level explanation doesn't fully make sense to you. We're now going to explain it in more detail through an example.

2. Faking Recursion through Approximations

The running example we're gonna use in this section is the identity function on natural numbers, albeit "defined" in a recursive manner:

```
f == lambda n. if n = 0 then 0 else 1 + f (n-1)
```

With this "definition," we're saying that the desired function `f` must satisfy the above equation, and we're asking the programming language to solve for `f`, i.e., giving us a concrete expression that behaves like `f`.

We know that a correct solution is simply:

```
lambda n. n
```

However, we're able to come up with this solution because we're humans and we're smart. The programming language, on the other hand, is not as smart as us, and it can't possible solve the equation for us.

Thus, from now on, let's just keep this correct answer at the back of our mind. Instead, we'll pretend that we're the "dumb" programming language, so we don't know what the correct answer is.

As a programming language, what we *do* know, however, is the original recurrence relation specified by the programmer:

```
f == lambda n. if n = 0 then 0 else 1 + f (n-1)
```

Since a language can only manipulate things that it can express, let's first try to encode this equation into a form that the programming language can understand. Let's use the plain old λ-calculus (extended with numbers and booleans) to do this.

A first attempt might be to simply take the right-hand-side of the equation and write it down as a λ-expression:

```
lambda n. if n = 0 then 0 else 1 + (f (n-1))
```

This is almost correct, but not quite. The problem is that this expression contains `f` which is a free variable. You learned that the meaning of an expression with free variables is undefined. For example, if you type `1 + x` into an interpreter, say the Python interpreter, it's gonna complain that it doesn't know how to interpret the unbound variable `x`.

There's a simple trick to make a free variable bound: we can bind it using a λ-abstraction. In other words, we can turn the free variable `f` into a bound variable by writing:

```
lambda f. lambda n. if n = 0 then 0 else 1 + (f (n-1))
```

Now, *this* is an expression whose meaning is well-defined in λ-calculus. Conceptually, it retains all the information from our original equation/recurrence relation, but it's now in a form that the programming language can fully comprehend.

Let's call this expression $\mathbb{G}$:

```
𝔾 := lambda f. lambda n. if n = 0 then 0 else 1 + (f (n-1))
```

Of course, $\mathbb{G}$ can't be the identity function that we seek to define, because it takes some *other* function `f` as its first argument. However, since it retains all the information from our original equation, we can probably do something with it to get closer to the correct answer. In other words, let's study what the behavior of $\mathbb{G}$ is.

First and foremost, $\mathbb{G}$ is a function. To study the behavior of a function, we can apply it to some argument and see what the output is.

- The first argument of $\mathbb{G}$ is `f`. Since a candidate for `f` is the identity function on natural numbers, we know that any `f` must be a function that takes a natural number and returns a natural number.
- Let's apply $\mathbb{G}$ to the simplest possible `f`: the function that always crashes and burns on any input. Let's call this function frownie:

  ```
  ☹ := lambda n. <crash-and-burn>
  ```

  It doesn't matter what `<crash-and-burn>` is, as long as it gets stuck or goes into an infinite loop on any input. For example, it could be the following:

  ```
  ☹ := lambda n. true + 1
  ```

which will clearly get stuck.

Now, let's apply 𝔾 to ☹, and call the result $\texttt{f}_0$:

```
f₀ := 𝔾 ☹
```

To apply 𝔾 to ☹, we simply replace `f` with ☹ in the body of 𝔾, and we get:

```
f₀ := 𝔾 ☹ == lambda n. if n = 0 then 0 else 1 + ☹ (n-1)
```

Ah, we finally get something that looks like an answer to our original equation! Is this the correct answer, i.e., the identity function? Let's see:

- To see if $\texttt{f}_0$ the identify function, we can apply $\texttt{f}_0$ to the number 0, and see if we get 0 back:

  ```
  (f₀ 0)
  == if 0 = 0 then 0 else 1 + ☹ (0-1)
  == 0
  ```

  Hmm, so if we just look at the behavior of $\texttt{f}_0$ on the input 0, it seems to be the identity function. However, we can't be sure that $\texttt{f}_0$ is the identity function elsewhere, because we only tested it on one input. We need to test it on more inputs to be sure.
- Let's apply $\texttt{f}_0$ to the number 1:

  ```
  (f₀ 1)
  == if 1 = 0 then 0 else 1 + ☹ (1-1)
  == 1 + ☹ (0)
  == 1 + <crash-and-burn>
  == <crash-and-burn>
  ```

  Oh no! $\texttt{f}_0$ is not the identity function, because it crashes on the input 1.
- Similarly, we can apply $\texttt{f}_0$ to the number 2:

  ```
  (f₀ 2)
  == if 2 = 0 then 0 else 1 + ☹ (2-1)
  == 1 + ☹ (1)
  == 1 + <crash-and-burn>
  == <crash-and-burn>
  ```

  Again, $\texttt{f}_0$ crashes on the input 2. You can probably guess that $\texttt{f}_0$ will crash on any input $n \geq 1$.

Thus, our first candidate $\texttt{f}_0$ does not solve our original equation. But, it's too early to throw up our hands and give up. Previously, we applied 𝔾 to the function ☹, which is a function that crashes on any input. It's not surprising that we didn't get a good answer. We should try applying 𝔾 to a better function.

But we just obtained a better function, which is $\texttt{f}_0$! Previously, ☹ crashed on any input, but $\texttt{f}_0$ only crashes on inputs $n \geq 1$, and actually gives the right output on the input 0. Thus, $\texttt{f}_0$ is a better function than ☹. Let's apply 𝔾 to $\texttt{f}_0$ and see what we get. Let's call the result $\texttt{f}_1$:

```
f₁  := 𝔾 f₀
    == lambda n. if n = 0 then 0 else 1 + f₀ (n-1)
```

Is $\texttt{f}_1$ the identity function? Let's see:

- If we apply $\texttt{f}_1$ to the number 0, we get:

  ```
  (f₁ 0)
  == if 0 = 0 then 0 else 1 + f₀ (0-1)
  == 0
  ```

So, `f1` behaves like the identity function on the input 0. However, we need to test it on more inputs to be sure.

- If we apply `f1` to the number 1, we get:

```
(f1 1)
== if 1 = 0 then 0 else 1 + f0 (1-1)
== 1 + f0 (0)
== 1 + 0
== 1
```

We've made progress! `f1` behaves like the identity function on the input 1, thanks to the fact that `f0` behaves like the identity function on the input 0! However, just to be sure, we should test `f1` on more inputs.

- If we apply `f1` to the number 2, we get:

```
(f1 2)
== if 2 = 0 then 0 else 1 + f0 (2-1)
== 1 + f0 (1)
== 1 + <crash-and-burn>
== <crash-and-burn>
```

Oh no! `f1` crashes on the input 2. And we can probably guess that `f1` will crash on any input $n \geq 2$.

Although `f1` is still not the identity function, it's better than `f0` because it behaves like the identity function on the input set $\{0, 1\}$, whereas `f0` only behaves like the identity function on the input set $\{0\}$. Thus, `f1` is closer to the correct answer than `f0`.

Can you see where this is going? We can keep applying $\mathbb{G}$ to the current function we have, and get a closer answer out. Next, let's apply $\mathbb{G}$ to `f1` to get an even better function, `f2`. It should be easy to convince yourself that `f2` behaves like the identity function on the input set $\{0, 1, 2\}$, which is indeed better than `f1`.

We can keep doing this, and we'll get a sequence of functions `f0`, `f1`, `f2`, ..., where each function is better than the previous one. If we expand the definition of each function, we'll see that the sequence of functions is:

$$
\begin{aligned}
&☹ \\
\texttt{f}_0 &= \mathbb{G}☹ \\
\texttt{f}_1 &= \mathbb{G}(\mathbb{G}☹) \\
\texttt{f}_2 &= \mathbb{G}(\mathbb{G}(\mathbb{G}☹)) \\
&\vdots \\
\texttt{f}_n &= \mathbb{G}^{n+1}(☹) \\
&\vdots
\end{aligned}
$$

where `f`$_n$ is the n-th function in the sequence, and it is obtained by applying $\mathbb{G}$ to ☹ $n-1$ times. The notation $f^k(x)$ means iteratively applying some function f to the input x for a total of k times.

If we consider what $\mathbb{G}$ does at every step, it is not hard to show that:

For any function f, if n is the maximum number for which $f(n) = n$, then we have

$$(\mathbb{G}\ f)(n+1) = n+1.$$

In other words, if f's best effort is to behave like the identity function on input number n, then applying $\mathbb{G}$ to f will give us a function that behaves like the identify function on $n+1$.

In other words, if we think of a candidate answer as some *approximation* of the correct, ground-truth answer, then applying $\mathbb{G}$ to the candidate will give us a *slightly better approximation*.

We now fully understand what the behavior of $\mathbb{G}$ is: it is an *approximation-improvement machine*. Given any "recursive-defined" function f, we can easily obtain the corresponding $\mathbb{G}$ by abstracting the recursive call into a function parameter. Given any approximation, applying $\mathbb{G}$ to it gives a one-step improvement.

In programming language theory, people usually call $\mathbb{G}$ the *generator* of a recursive function.

Now we understand what happens when we apply $\mathbb{G}$ one time. What if we apply $\mathbb{G}$ n times? Intuitive, we should get an n-step improvement. Indeed, this is also straightforward to show:

For any initial function i (which is not necessarily $\circledcirc$), $\mathbb{G}^n\ i$ will behave like the identity function on (at least) the input set $\{0, 1, \ldots, n\}$.

The initial function i is actually arbitrary – we chose $\circledcirc$ because it is the worst possible approximation, but we could have chosen a better approximation as our starting point, although we don't have to.

In particular, since i can be arbitrary, we can take i to be $\mathbb{G}$ itself! By the above observation, this means that $\mathbb{G}^n\ \mathbb{G} = \mathbb{G}^{n+1}$ will behave like the identity function on at least the input set $\{0, 1, \ldots, n\}$. Thus, we can consider the following sequence of functions:

$$
\begin{aligned}
\mathbb{G}^1 &= \mathbb{G} && \text{correct on input set } \varnothing \\
\mathbb{G}^2 &= \mathbb{G}\mathbb{G} && \text{correct on input set } \{0\} \\
\mathbb{G}^3 &= \mathbb{G}\mathbb{G}\mathbb{G} && \text{correct on input set } \{0, 1\} \\
\mathbb{G}^4 &= \mathbb{G}\mathbb{G}\mathbb{G}\mathbb{G} && \text{correct on input set } \{0, 1, 2\} \\
&\vdots \\
\mathbb{G}^{n+1} &= \mathbb{G}\mathbb{G}\cdots\mathbb{G} && \text{correct on input set } \{0, 1, \ldots, n\} \\
&\vdots
\end{aligned}
$$

If we repeat this process infinitely many times, we should get $\mathbb{G}^\infty$, which is correct on the input set $\{0, 1, 2, \ldots\}$. In other words, $\mathbb{G}^\infty$ will be the identity function for all possible natural numbers, so it's a correct solution to the original equation that "defines" the identity function!

> This object called $\mathbb{G}^\infty$ is a very special one. If we apply $\mathbb{G}$ to it, we should get $\mathbb{G}^\infty$ again, because $\mathbb{G}^\infty$ is already the best possible approximation, so $\mathbb{G}$ can't improve it any further.
>
> In mathematical terms, this profound phenomenon is dubbed a *fixed point*.
>
> **Definition 2.1.** Let $f : A \to A$ be a function. A point $x \in A$ is called a *fixed point* of f if $f(x) = x$.
>
> Intuitively, if we think of a function as an action that acts on an object, then a fixed-point of the function is some parts of the object that are left unchanged by the action. Examples abound:

- Let our object be an asparagus, which contains a bunch of atoms that are arranged in a certain way. Let the action be "dropping the asparagus on the ground". In this case, the position of every atom in the asparagus is changed by the action, so this action has no fixed points.
- Let our object be an asparagus, and let the action be "rotating the asparagus around the center of its stem." In this case, the position of every atom, except those ones lying on the axis of rotation, are changed by the action. Thus, we say that the fixed-points of the action are the asparagus atoms lying on the axis of rotation.
- Let our object be the set of numbers, and the action by the function $f(x) = x^2$. In this case, the fixed-points of f are the numbers that satisfy $x = x^2$, which are 0 and 1.

In our case, we're saying that the strange object $\mathbb{G}^\infty$ is a fixed-point of the function $\mathbb{G}$, because $\mathbb{G}$'s action of "improving the input function" will not have any effect on $\mathbb{G}^\infty$. This is where the names "fix" and "fixed-point operator" come from.

Of course, it is impossible to write down $\mathbb{G}^\infty$ in a programming language, because it is an infinite object – if we try to write it down, we would end up with an infinite-long program, which is not allowed.

However, the key insight is that we don't need to write down the entirety of $\mathbb{G}^\infty$ at once. We can introduce a special syntax into our language that *represents* the process of applying some function to itself infinitely many times. That is, we could augment λ^+ with:

Syntactic construct			Description
$e \in Expr$	$::=$	x	variable
	$\mid$	$\mathsf{lambda}\, x.\ e$	lambda abstraction
	$\mid$	$e_1\ e_2$	function application
		$\vdots$	
	$\mid$	e^∞	infinite application

Syntactically, e^∞ is finite: we can write it down using two symbols: e and ∞. However, semantically, it represents the process of applying e to itself infinitely many times, which can be given by the following evaluation rule:

$$\frac{(e\ e^\infty) \Downarrow v}{e^\infty \Downarrow v}\ \text{E-Inf}$$

This rule says that, in order to evaluate e^∞, we first expand it into the self-application, where we apply e to e^∞ itself, and then we evaluate the application.

This is almost the same as the Fix rule we saw at the beginning of this section, except for some syntactic differences.

First, let's observe that e itself has to be a function because we need to apply e to some argument. For example, previously $e = \mathbb{G}$, and $\mathbb{G}$ itself is a function that takes some function `f` as its argument. Thus, we can refine E-Inf by replacing the generic e with a λ-abstraction of the form "$\lambda f.\ e$":

$$\frac{(\lambda f.\ e)\ (\lambda f.e)^\infty \Downarrow v}{(\lambda f.\ e)^\infty \Downarrow v}\ \text{E-Inf'}$$

Then, we can expand the hypothesis $(\lambda f.\ e)\ (\lambda f.e)^\infty$ using the evaluation rule of application. Recall that to evaluate a function application, we substitute the function parameter with the

actual input in the function body. In this case, the function parameter is f, the input is $(\lambda f.e)^\infty$, and the body is e. So we evaluate the hypothesis one more time:

$$\dfrac{\dfrac{e[f \mapsto (\lambda f.e)^\infty] \Downarrow v}{(\lambda f.\ e)\ (\lambda f.e)^\infty \Downarrow v}\ \textsc{App}}{(\lambda f.\ e)^\infty \Downarrow v}\ \textsc{E-Inf'}$$

Finally, we can collapse the above tree into a single rule, and make a small syntactic change: instead of writing $(\lambda f.e)^\infty$, let's just write $\mathsf{fix}\, f\ \mathsf{is}\ e$. This gives us the Fix rule we saw at the beginning of this section:

$$\dfrac{e[f \mapsto \mathsf{fix}\, f\ \mathsf{is}\ e] \Downarrow v}{\mathsf{fix}\, f\ \mathsf{is}\ e \Downarrow v}\ \textsc{Fix}$$

This hopefully gives you a better understanding of the Fix rule. In summary, to give meaning to recursive "definitions" in a programming language, we do the following:

(1) We observe that recursive "definitions" are not really definitions, but rather equational specifications.
(2) We encode the right-hand-side of the equation into a λ-expression, and abstract the recursive call into a function parameter. The resulting function $\mathbb{G}$ is called the generator of the recursive function.
(3) We observe that if $\mathbb{G}$ is applied to any approximation, we get a slightly better approximation.
(4) We bootstrap from this, and apply $\mathbb{G}$ to itself infinite many times, which gives the best possible approximation, i.e., a correct solution to the original equation.
(5) To syntactically represent the process of applying a function to itself infinite many times, we augment the language with the fixed-point operator.

3. Terminating Recursion

We have seemingly found a way out of the paradox. We said that there could be no algorithm that solves arbitrary recurrence relations, but we came up with a strange object $\mathbb{G}^\infty$ that seems to solve the recurrence relation.

But this is not a paradox as it seems, because $\mathbb{G}^\infty$ only exists in our imagination – if we try to evaluate the expression $\mathbb{G}^\infty$ in the real world, the process will never terminate.

This raises an important question: if $\mathbb{G}^\infty$ is used to represent recursion and it never terminates, then how can we use it in a real programming language and get anything done?

The short answer is that, the fixed-point operator *can* terminate for carefully chosen equations. For example, consider the evaluation of $\mathsf{fix}\, f\ \mathsf{is}\ 2 + 3$. Conceptually, we should read this as a polite request for the programming language to solve the following equation:

$$\boxed{\texttt{f} \text{ is some value that satisfies the equation } \texttt{f} = 2 + 3}$$

Clearly, this equation has a solution if we take $\texttt{f} = 5$. Indeed, if we evaluate $\mathsf{fix}\, f\ \mathsf{is}\ 2 + 3$, we get:

$$\dfrac{(2+3)[f \mapsto \mathsf{fix}\, f\ \mathsf{is}\ 2+3] = 2+3 \qquad \dfrac{\dfrac{}{2 \Downarrow 2}\ \textsc{Int} \quad \dfrac{}{3 \Downarrow 3}\ \textsc{Int} \qquad 2+3=5}{2+3 \Downarrow 5}\ \textsc{Arith}}{\mathsf{fix}\, f\ \mathsf{is}\ 2+3 \Downarrow 5}\ \textsc{Fix}$$

So the evaluation rule for the fixed-point operator indeed gives us a correct solution to the equation. In this case, the "solving process" terminates: we never refer to `f` on the right-hand-side of the equation, so when we're substituting `f` with the fixed-point operator, no substitution is needed, and we simply evaluate the expression $2 + 3$ to get the result.

Usually, however, the body of fix will be a lambda abstraction. In this case, we also do not (immediately) get into an infinite loop, since lambda abstractions evaluate to themselves. For example, consider the expression $\lambda x.\ \mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x$. This expression is, again, a polite request for the programming language to solve the following equation:

$$\boxed{f \text{ is some value that satisfies the equation } f = \lambda x.\ f\ x}$$

To "solve" this equation, the programming language simply evaluates the expression $\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x$ to get:

$$\dfrac{(\lambda x.\ f\ x)[f \mapsto \mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x] = \lambda x.\ (\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x)\ x \qquad \dfrac{}{\lambda x.\ ..\ \Downarrow \lambda x.\ ..}\ \textsc{Lam}}{\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x \Downarrow \lambda x.\ (\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x)\ x}\ \textsc{Fix}$$

The solving process is able to terminate, thanks to the Lambda rule: any lambda abstraction evaluates to itself. The huge price is that in this case we the "solution" doesn't solve the equation! Indeed, applying this "solution" – $\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x$ – to some arguments leads to an infinite loop while the evaluation attempts to evaluate the same thing again and again. For example, if we apply the "solution" to 10, we will get:

$$\dfrac{\dfrac{}{\lambda x.\ ..\ \Downarrow \lambda x.\ ..}\ \textsc{Lam} \qquad (..)[x \mapsto 10] = (\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x)\ 10 \qquad \dfrac{\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x \Downarrow \lambda x.\ .. \qquad (\lambda x.\ ..)\ 10 \Downarrow ??}{(\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x)\ 10 \Downarrow ??}\ \textsc{App}}{(\lambda x.\ (\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ f\ x)\ x)\ 10 \Downarrow ??}\ \textsc{App}$$

Note that the two red nodes are the same, and the derivation tree will grow indefinitely.

The upshot is that the fixed-point operator can terminate, but it may also not terminate. Whether it terminates or not depends on the equation specified by the programmer.

This should also agree with your previous experience with recursive functions: the fact that a recursive algorithm terminates needs to be explicitly proven by the algorithm designer. The use of recursion itself doesn't guarantee termination; on the contrary, most recursive functions do not terminate! Thus, the fixed-point operator, which gives meaning to recursive definitions, is an extremely powerful "double-edged sword" that allows the programmer to express both terminating and non-terminating recursion.

In general, infinite loops can be circumvented by stuffing the fixed-point operator under any language construct that is *lazy* – the construct doesn't evaluate all of its arguments unless absolutely necessary. For example, if you're writing a recursive function on natural numbers, the recursive call will usually appear as part of the `then` branch of an `if-then-else` expression. However, in an if-then-else expression, the `then` branch is only evaluated if the condition is true, so the recursive call will only be evaluated if it is needed. In this way, the infinite application $\mathbb{G}^\infty$

is only expanded once per each recursive call. This is the reason that carefully crafted recursive functions can terminate in practice.

Exercise: Draw the derivation tree for the evaluation of

$$(\mathsf{fix}\, f\ \mathsf{is}\ \lambda x.\ \mathsf{if}\, x = 0\, \mathsf{then}\, 1\, \mathsf{else}\, x * f(n-1))\ \ 3.$$