A study and implementation of the guarded λ-calculus

Naïm Favier

2nd June – 31st July 2020

Internship report — ENS Paris

Supervised by: Guilhem Jaber — Gallinette team — LS2N

1 Introduction

At the core of the liar's paradox and its thousand variants lies the notion of (negative) *self-reference*. This translates, on the other side of the Curry-Howard correspondence, to the fact that adding recursive types to the simply-typed λ-calculus loses strong normalisation : every type A is then inhabited by the non-terminating term $\Omega = \omega\,\omega = (\lambda x.\, x\,x)(\lambda x.\, x\,x)$[1] where ω is given the type $\mu\alpha.\, \alpha \to A$. Setting $A = \bot$, this type can be read as "This proposition is false".

But the strong normalisation property is quite useful: it guarantees, on one hand, that programs terminate, and on the other hand, that *co*programs (in the sense of [AM13]: programs that manipulate potentially infinite data structures) are *productive*: that is, that they always produce the next piece of output in finite time. As an example, consider the following Haskell definitions on infinite lists (*streams*):

```
nats = 0 : map (+1) nats
bottom = bottom
filter p (x : xs) | p x       = x : filter p xs
                  | otherwise = filter p xs
```

While `nats` is productive and yields the stream of natural numbers, `bottom` clearly diverges and never produces a value. `filter` is not productive either, even if its input stream is, because the predicate might fail to match any of the elements of the input stream.

This internship focuses on the use of **guarded recursion**, initially due to Hiroshi Nakano [Naka00], to achieve productivity in the presence of recursive types and definitions. It adds a *modality* to the type system, allowing to ensure that well-typed programs are productive. This approach led to the development of the **guarded λ-calculus** by Clouston et al. [CBBB16], which will be our object of study.

The initial goal of the internship was to explore the system of *temporal refinements* proposed by Guilhem Jaber and Colin Riba [JR20], which extends the guarded λ-calculus with type-level refinements allowing to express safety and liveness properties; however, given the circumstances[2], we agreed that this goal was too ambitious, and I instead focused on implementing an evaluator and type-checker for the guarded λ-calculus in Haskell, later extending it with Hindley-Milner-style polymorphism and implicit boxing.

The resulting project, **`glam`**, can be found at `https://github.com/ncfavier/glam`. It provides a command-line interpreter and REPL, as well as a web-based interface with a few example programs to play with, available at `https://glam.monade.li`.

[1] up to isomorphism, or assuming equirecursive types

[2] due to the health crisis, this internship had to be done remotely

2 The guarded λ-calculus

We now give a brief and largely informal overview of the guarded λ-calculus as presented in [CBBB16]; see there for a completely formal presentation.

N.B. — the type signatures presented in this section do not necessarily correspond to types of the guarded λ-calculus, or even type schemes, but should rather be understood as informal descriptions of its typing rules.

2.1 Types and terms

The guarded λ-calculus is based on the simply-typed λ-calculus. It is equipped with usual base constructs such as integers, a unit type, product types, a zero type, sum types, and the corresponding term formers.

Guarded recursion adds a modality (that is, a unary type former) $\blacktriangleright$[3], pronounced "later", to the type system. Intuitively, a value of type $\blacktriangleright A$ is a value of type A that is only available *later*, after one computation step (e.g. unfolding one level of a fixed-point definition). This modality can be seen as an applicative functor [MP08], and comes with the corresponding term formers:

$$\begin{aligned} \texttt{next} &: A \to \blacktriangleright A \\ \circledast &: \blacktriangleright(A \to B) \to \blacktriangleright A \to \blacktriangleright B \end{aligned}$$

Guarded recursive types are then defined as types of the form $\mu\alpha.\, F(\alpha)$, with the restriction that α be *guarded* in $F(\alpha)$, that is, that all occurences of α appear under an occurence of $\blacktriangleright$. This restriction means in particular that one cannot express traditional fixed-point combinators (of type $(A \to A) \to A$) using guarded recursive types, but only weaker variants of type $(\blacktriangleright A \to A) \to A$. This intuitively means that, in a recursive definition, the object being defined is only available *later*. Since we use an *isorecursive* presentation of guarded recursive types, we also need term formers for both sides of the isomorphisms:

$$\begin{aligned} \texttt{fold} &: A[\mu\alpha.\, A/\alpha] \to \mu\alpha.\, A \\ \texttt{unfold} &: \mu\alpha.\, A \to A[\mu\alpha.\, A/\alpha] \end{aligned}$$

As an example, the type of guarded streams can be defined as $\texttt{Str}^{\mathsf{g}}A = \mu\alpha.\, A \times \blacktriangleright\alpha$. One can then define the constructors and destructors

$$\begin{aligned} (::^{\mathsf{g}}) &: A \to \blacktriangleright\texttt{Str}^{\mathsf{g}}A \to \texttt{Str}^{\mathsf{g}}A \\ \texttt{hd}^{\mathsf{g}} &: \texttt{Str}^{\mathsf{g}}A \to A \\ \texttt{tl}^{\mathsf{g}} &: \texttt{Str}^{\mathsf{g}}A \to \blacktriangleright\texttt{Str}^{\mathsf{g}}A \end{aligned}$$

While the use of $\blacktriangleright$ guarantees productivity, it is actually too strong to allow certain productive definitions. For example, consider the function `every2nd`, which removes every second element from a stream; while productive, this function is *acausal*: elements of the output stream depend on deeper elements of the input stream. As a consequence, it cannot be expressed using guarded recursive types alone, because there is no way to get from $\blacktriangleright\blacktriangleright A$ to $\blacktriangleright A$.

This is solved by introducing another modality $\blacksquare$, pronounced "constant", as a way of indicating that a computation is *complete*, i.e. that the entire object is available *now*. This is a simpler presentation of the *clock quantifiers* proposed by Atkey and McBride [AM13], where $\blacksquare$ replaces $\forall\kappa$. As an additional restriction, fixed-point variables cannot appear under $\blacksquare$; this is to disallow types like $\mu\alpha.\, F(\blacksquare\blacktriangleright\alpha)$ which would defeat guardedness.

This entails a notion of *constant types*: a type A is constant if and only if all occurrences of $\blacktriangleright$ in A appear under an occurrence of $\blacksquare$.

[3]initially called $\bullet$ in [Naka00], but we follow the conventions of [CBBB16]

We can now introduce the following term formers (informally):

$$\begin{aligned}\texttt{box} &: A \to \blacksquare A\\ \texttt{unbox} &: \blacksquare A \to A\\ \texttt{prev} &: \blacktriangleright A \to A\end{aligned}$$

While `unbox` is straightforward, there must be restrictions on what terms we are allowed to use `box` and `prev` on, in order not to lose our productivity guarantees. These are expressed in the formal typing rules:

$$\frac{x_1 : A_1, \dots, x_n : A_n \vdash t : A \qquad \Gamma \vdash t_1 : A_1 \qquad \cdots \qquad \Gamma \vdash t_n : A_n \qquad A_1, \dots, A_n \text{ constant}}{\Gamma \vdash \texttt{box}[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n].\, t : \blacksquare A}$$

$$\frac{x_1 : A_1, \dots, x_n : A_n \vdash t : \blacktriangleright A \qquad \Gamma \vdash t_1 : A_1 \qquad \cdots \qquad \Gamma \vdash t_n : A_n \qquad A_1, \dots, A_n \text{ constant}}{\Gamma \vdash \texttt{prev}[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n].\, t : A}$$

There are two things to note here:

- The restriction for typing `box` t or `prev` t is that the free variables of t be all bound to constant types. This guarantees that t only depends on data that is available *now*. As a degenerate example, any closed term verifies this condition.
- The term formers `box` and `prev` are annotated with *delayed substitutions*, between square brackets. These substitutions must *close* the term t, so that it stays invariant under later substitutions. This serves the purpose of guaranteeing preservation of typing under substitution. Indeed, without delayed substitutions, one could substitute e.g. $f\ y$ for x in `box` x; assuming $x : A$ with A constant, $y : B$ and $f : B \to A$ with B non-constant, this would result in the term `box` $(f\ y)$ which is ill-typed since y has a non-constant type. Thus we close the term x to get `box`$[x \leftarrow x].\, x$, which becomes `box`$[x \leftarrow f\ y].\, x$ after substitution, and stays well-typed. This solution comes from the study of intuitionistic modal logic [Bd00].

  By convention, `box` t means `box`$[x_1 \leftarrow x_1, \dots, x_n \leftarrow x_n].\, t$, where $x_1, \dots, x_n$ are the free variables of t.

As a final technicality, notice that while we can construct terms witnessing the isomorphism $\blacksquare(A \times B) \cong \blacksquare A \times \blacksquare B$, as well as the direction $\blacksquare A + \blacksquare B \to \blacksquare(A + B)$, there is no way to construct a term of type $\blacksquare(A + B) \to \blacksquare A + \blacksquare B$. Indeed, the term

$$\texttt{distribConstSum} = \lambda x.\ \texttt{case}\ \texttt{unbox}\ x\ \texttt{of}\ a.\ \texttt{in}_1\ \texttt{box}\ a;\, b.\ \texttt{in}_2\ \texttt{box}\ b$$

is ill-typed in general, because a and b have the potentially non-constant types A and B. We solve this by introducing a term former `box`$^+$, which is similar to `box` except that it takes $A + B$ to $\blacksquare A + \blacksquare B$:

$$\frac{x_1 : A_1, \dots, x_n : A_n \vdash t : A + B \qquad \Gamma \vdash t_1 : A_1 \qquad \cdots \qquad \Gamma \vdash t_n : A_n \qquad A_1, \dots, A_n \text{ constant}}{\Gamma \vdash \texttt{box}^+[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n].\, t : \blacksquare A + \blacksquare B}$$

The $\blacksquare$ modality allows us to recover *coinductive types* from guarded recursive types. For example, `Str`$A = \blacksquare$`Str`$^{\mathsf{g}}A$ is the traditional coinductive type of streams of A.

Most functions defined on guarded recursive types can be *lifted* to coinductive types by appropriate uses of `box`, `unbox`, `next` and `prev`: for example, $(::) = \lambda x.\, \lambda s.$ `box` $(x ::^{\mathsf{g}}$ `next` `unbox` $s)$ is the standard constructor for coinductive streams.

2.2 Operational semantics

The guarded λ-calculus is equipped with a simple call-by-name evaluation strategy. We can summarise the types and terms of the language (except for $\circledast$) as follows:

Type former	**Constructor**	**Destructor**
$\mathbf{0}$		`abort`
$\mathbf{1}$	$\langle\rangle$	
$\mathbf{N}$	n	
$+$	`in`$_1$, `in`$_2$	`case`
$\times$	$\langle\cdot,\cdot\rangle$	π_1, π_2
$\to$	λ	$\cdot\, t$
μ	`fold`	`unfold`
$\blacktriangleright$	`next`	`prev`
$\blacksquare$	`box`	`unbox`

With this in mind, *values* of the guarded λ-calculus are terms whose root node is a constructor; the basic reduction rules are β-reductions, simplifying away the application of a destructor to a constructor (with appropriate care taken to deal with delayed substitutions); and the standard call-by-name context rules allow to reduce under destructors.

The only outlier is $\circledast$, whose reduction rule does not correspond to a β-reduction, but rather to the *homomorphism* law of applicative functors:

$$\texttt{next}\, t_1 \circledast \texttt{next}\, t_2 \mapsto \texttt{next}\, (t_1 t_2)$$

Additional context rules allow us to reduce under each side of $\circledast$.

2.3 Interpretation in the topos of trees

We can interpret the guarded λ-calculus semantically in the *topos of trees* $\mathcal{S}$ [BMSS12], which is defined as the category of presheaves over the first infinite ordinal ω. Concretely:

- An object X of $\mathcal{S}$ is a positive integer-indexed collection of sets $(X_i)_{i\geq 1}$ equipped with restriction functions $r_i^X : X_{i+1} \to X_i$.
- A morphism $f : X \to Y$ in $\mathcal{S}$ is a family of functions $(f_i : X_i \to Y_i)_{i\geq 1}$ verifying the naturality condition $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$ for $i \geq 1$. This can be expressed as the infinite commutative diagram:

$$\begin{tikzcd}
X_1 \arrow[d, "f_1"'] & X_2 \arrow[l, "r_1^X"'] \arrow[d, "f_2"'] & X_3 \arrow[l, "r_2^X"'] \arrow[d, "f_3"'] & \cdots \arrow[l, "r_3^X"'] \\
Y_1 & Y_2 \arrow[l, "r_1^Y"] & Y_3 \arrow[l, "r_2^Y"] & \cdots \arrow[l, "r_3^Y"]
\end{tikzcd}$$

- There is an adjunction

$$\begin{tikzcd}
\mathbf{Set} \arrow[r, shift left, "\Delta"] \arrow[r, phantom, "\perp"] & \mathcal{S} \arrow[l, shift left, "\mathrm{Hom}_{\mathcal{S}}(1,-)"]
\end{tikzcd}$$

  where Δ sends a set X to the constant $\mathcal{S}$-object ΔX:

$$\begin{tikzcd}
X & X \arrow[l, "\mathrm{id}_X"'] & X \arrow[l, "\mathrm{id}_X"'] & \cdots \arrow[l, "\mathrm{id}_X"']
\end{tikzcd}$$

and $\mathrm{Hom}_{\mathcal{S}}(1, -)$ sends an $\mathcal{S}$-object Y to its set of *global elements*, that is, morphisms from the terminal object to Y. We say that an $\mathcal{S}$-object is *constant* if it is isomorphic to some ΔX, or, equivalently, if its restriction functions are bijections.

This adjunction gives rise to a comonad $\blacksquare = \Delta \circ \mathrm{Hom}_{\mathcal{S}}(1, -)$, which is used to interpret the $\blacksquare$ modality.

- $\mathcal{S}$ is cartesian closed:
  - the initial, terminal and natural number objects are $\Delta\emptyset$, $\Delta\{*\}$ and $\Delta\mathbb{N}$ respectively
  - products and coproducts are defined pointwise on the underlying sets
  - the exponential B^A has as its ith component the set of i-tuples $(f_j : A_j \to B_j)_{1\leq j\leq i}$ satisfying the naturality condition above, and projections for restriction maps

  This allows to interpret the **0**, **1**, **N**, $\times$, $+$ and $\to$ type formers of the guarded λ-calculus.

- There is a functor $\blacktriangleright$ that sends an $\mathcal{S}$-object X to the $\mathcal{S}$-object

$$\begin{tikzcd}\{*\} & X_1 \arrow[l, "!"'] & X_2 \arrow[l, "r_1^X"'] & \cdots \arrow[l, "r_2^X"']\end{tikzcd}$$

  This allows to interpret the $\blacktriangleright$ modality, and `next` is interpreted as a natural transformation $\mathrm{id}_{\mathcal{S}} \to \blacktriangleright$:

$$\begin{tikzcd}X_1 \arrow[d, "!"'] & X_2 \arrow[l, "r_1^X"'] \arrow[d, "r_1^X"'] & X_3 \arrow[l, "r_2^X"'] \arrow[d, "r_2^X"'] & \cdots \arrow[l, "r_3^X"'] \\ \{*\} & X_1 \arrow[l, "!"] & X_2 \arrow[l, "r_1^X"] & \cdots \arrow[l, "r_2^X"]\end{tikzcd}$$

- The fixed-point type $\mu\alpha.\, F(\alpha)$ is interpreted as the unique fixed point of the functor F. The existence and uniqueness of this fixed point is given by [BMSS12, theorem 4.5], provided that α is guarded in F and doesn't appear under $\blacksquare$.

Section 2.3 of [CBBB16] ties the terms of the guarded λ-calculus with their semantic interpretation to prove (strong) normalisation and adequation results.

3 A practical implementation: `glam`

The main goal of this internship was to turn a theoretical language, the guarded λ-calculus, into a practically usable programming language. I will now discuss the changes that I made to the guarded λ-calculus in that direction.

N.B. — while [CBBB16] mentions an Agda implementation[4] of the guarded λ-calculus, that implementation seems more proof-oriented than programming-oriented. It was interesting to study, however, as I was not familiar with Agda, nor with the technique of embedding a target language and type system within a host language using generalised algebraic data types.

`glam`'s syntax is detailed in the file `README.md` and is intended to be close to Haskell's. We will first stick to the mathematical notations for discussing the theoretical aspects, and then give a few examples of **`glam`** programs in section 3.4.

As a first notable difference with the guarded λ-calculus, I got rid of delayed substitutions on `prev`- and `box`-terms. Indeed, while they are useful on a theoretical level to guarantee preservation of typing, this property isn't necessary in practice: programs are only type-checked once, before execution.

[4] `https://web.archive.org/web/20200811210702/https://hansbugge.dk/bin/glambda.zip`

Except for this, **glam**'s evaluation model simply implements the call-by-name operational semantics described in [CBBB16, section 1.1]. This model is very naïve, and incurs an exponential blow-up in complexity on some very simple programs. This could be improved by *sharing* the evaluation of common subexpressions resulting from the same β-reduction (call-by-need), but I didn't explore this direction, for lack of time.

3.1 Constant types

I noticed that the definition of constant types could be relaxed to allow non-constant types to appear on the *left* of a function arrow. This is based on the following observation:

Lemma 1. *Let A be an $\mathcal{S}$-object and B a constant $\mathcal{S}$-object. Then B^A is a constant $\mathcal{S}$-object.*

Proof. Since B is constant, r_j^B is an isomorphism for all j, thus the naturality condition for $(B^A)_i = (f_j)_{1 \leq j \leq i}$ implies $f_{j+1} = (r_j^B)^{-1} \circ f_j \circ r_j^A$ for $1 \leq j < i$. It follows that the f_j, $1 < j < i$ are all uniquely determined by f_1, hence $(B^A)_i \cong (B^A)_1$ and therefore B^A is a constant object. □

3.2 Implicit (un)boxing

Supporting `let`-expressions raised a technical problem: the typing rules of the guarded λ-calculus do not allow typing the term $\texttt{let}\, z = \texttt{fix}\, z.\, 0 ::^{\mathsf{g}} z \,\texttt{in}\, \texttt{box}\, z$. Indeed, z has the non-constant type $\texttt{Str}^{\mathsf{g}}\, \mathbf{N} = \mu\alpha.\, \mathbf{N} \times \blacktriangleright\alpha$, and is therefore not allowed as a free variable under `box`. However, if we replace z with its definition, we get the closed term $\texttt{box}\,(\texttt{fix}\, z.\, 0 ::^{\mathsf{g}} z)$, which has type $\texttt{Str}\,\mathbf{N}$. Rather than evaluate all `let`-expressions statically before typing the program, which would raise other issues, my proposed solution simultaneously solves this problem and allows to get rid of the cumbersome $\texttt{box}^{+}$ term former.

We introduce a new, *distinct* kind of assertion in the typing contexts, of the form $x :^{\blacksquare} A$, and we say that the x variable is *boxed*. This is to be understood semantically as $x : \blacksquare A$, except that boxing and unboxing are done implicitly by the type system. We then say that a term t is *boxable* in context Δ, written $\Delta \vdash t :^{\blacksquare} A$, when Δ only contains boxed variables and variables bound to constant types. Such contexts are interpreted semantically as constant $\mathcal{S}$-objects. The precondition for typing $\texttt{box}\, t$ and $\texttt{prev}\, t$ is then simply that t must be boxable, and we allow variables that are `let`- or `case`-bound to a boxable term to be boxed.

This makes the above example type-check, as well as the `distribConstSum` term from earlier, making $\texttt{box}^{+}$ redundant.

Formally, the typing rules are modified as follows:

$$\frac{\Delta \vdash t : A \qquad \forall (x : B) \in \Delta, B \text{ constant}}{\Delta \vdash t :^{\blacksquare} A} \qquad \frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma, x :^{\blacksquare} A \vdash x : A}$$

$$\frac{\Delta \vdash t :^{\blacksquare} A + B \qquad \Gamma, \Delta, x_1 :^{\blacksquare} A \vdash t_1 : C \qquad \Gamma, \Delta, x_2 :^{\blacksquare} B \vdash t_2 : C}{\Gamma, \Delta \vdash \texttt{case}\, t\, \texttt{of}\, x_1.\, t_1; x_2.\, t_2 : C}$$

$$\frac{\Gamma \vdash t : A + B \qquad \Gamma, x_1 : A \vdash t_1 : C \qquad \Gamma, x_2 : B \vdash t_2 : C}{\Gamma \vdash \texttt{case}\, t\, \texttt{of}\, x_1.\, t_1; x_2.\, t_2 : C}$$

$$\frac{\Delta \vdash s :^{\blacksquare} A \qquad \Gamma, \Delta, x :^{\blacksquare} A \vdash t : B}{\Gamma, \Delta \vdash \texttt{let}\, x = s\, \texttt{in}\, t : B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{let}\, x = s\, \texttt{in}\, t : B} \qquad \frac{\Delta \vdash t :^{\blacksquare} \blacktriangleright A}{\Gamma, \Delta \vdash \texttt{prev}\, t : A}$$

$$\frac{\Delta \vdash t :^{\blacksquare} A}{\Gamma, \Delta \vdash \texttt{box}\, t : \blacksquare A}$$

and the interpretation of the terms is extended thus:

$$\begin{aligned}
[\![\Gamma, x :^{\blacksquare} A \vdash x : A]\!]_i(\gamma, a) &= a_i \\
[\![\Gamma, \Delta \vdash \texttt{let}\, x = s \,\texttt{in}\, t : B]\!]_i(\gamma, \delta) &= [\![\Gamma, \Delta, x :^{\blacksquare} A \vdash t : B]\!]_i(\gamma, \delta, j \mapsto [\![s]\!]_j(\delta)) && \text{if } \Delta \vdash s :^{\blacksquare} A \\
[\![\Gamma \vdash \texttt{let}\, x = s \,\texttt{in}\, t : B]\!]_i(\gamma) &= [\![\Gamma, x : A \vdash t : B]\!]_i(\gamma, [\![s]\!]_i(\gamma)) && \text{otherwise} \\
[\![\Gamma, \Delta \vdash \texttt{case}\, t \,\texttt{of}\, x_1.\, t_1; x_2.\, t_2 : B]\!]_i(\gamma, \delta) &= [\![\Gamma, \Delta, x_d :^{\blacksquare} A_d \vdash t_d : B]\!]_i(\gamma, \delta, j \mapsto a_j) && \text{if } [\![\Delta \vdash t :^{\blacksquare} A_1 + A_2]\!]_j(\delta) = [a_j, d] \\
[\![\Gamma \vdash \texttt{case}\, t \,\texttt{of}\, x_1.\, t_1; x_2.\, t_2 : B]\!]_i(\gamma) &= [\![\Gamma, x_d : A_d \vdash t_d : B]\!]_i(\gamma, a) && \text{otherwise, where } [\![t]\!]_i(\gamma) = [a, d] \\
[\![\Gamma, \Delta \vdash \texttt{box}\, t : \blacksquare A]\!]_i(\gamma, \delta) &= j \mapsto [\![\Delta \vdash t :^{\blacksquare} A]\!]_j(\delta) \\
[\![\Gamma, \Delta \vdash \texttt{prev}\, t : A]\!]_i(\gamma, \delta) &= [\![\Delta \vdash t :^{\blacksquare} \blacktriangleright A]\!]_{i+1}(\delta)
\end{aligned}$$

Just like in [CBBB16, definition 2.7], everything is well-defined because $[\![\Delta]\!]$ is a constant $\mathcal{S}$-object. The notation $[a, d]$, where $d \in \{1, 2\}$ and $a \in A_d$, refers to the injection into the **Set** coproduct $A_1 + A_2$.

Note that we have $[\![\Gamma, x :^{\blacksquare} A \vdash x : A]\!]_i(\gamma, a) = [\![\Gamma, x : \blacksquare A \vdash \texttt{unbox}\, x : A]\!]_i(\gamma, a) = a_i$, justifying the phrase "implicit unboxing".

3.3 Polymorphism

Hindley-Milner polymorphism is a simple yet useful system for adding polymorphism to the simply-typed λ-calculus. It consists of defining *polytypes* (or *type schemes*) as universally quantified versions of the monomorphic types; this is rank-1, predicative polymorphism because quantifiers can only appear at the top-level of a type.

Variables that are `let`-bound (and only those; this is called `let`-*polymorphism*) can be assigned a polymorphic type by *generalising*, that is, quantifying over the type variables that aren't free in the environment, and every use of a variable leads to a separate *instantiation* of its polymorphic type variables.

For example, the term $\texttt{let}\, id = \lambda x.\, x \,\texttt{in}\, id\, id\, 42$ is well-typed: id is given the most general type $\forall a.\, a \to a$, which is then instantiated to $(\mathbf{N} \to \mathbf{N}) \to (\mathbf{N} \to \mathbf{N})$ and $\mathbf{N} \to \mathbf{N}$ respectively in the subsequent uses of id.

`glam` extends the guarded λ-calculus with a similar system: we first define polytypes of the form

$$\forall a_1, \ldots, a_n.\, \forall^{\blacksquare} b_1, \ldots, b_m.\, A$$

where A is a monomorphic type possibly containing $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ as free variables (quantifiers can appear several times and in any order). The intuitive meaning of $\forall^{\blacksquare} a$ is "for all *constant* types a".

Extending the definition of constant types so that a polymorphic type variable is constant if and only if it is bound by $\forall^{\blacksquare}$, we can now infer polymorphic types using a variation on the standard Hindley-Milner algorithm. The inference algorithm needs to keep track of which unification variables refer to constant types, so that it can generalise them to $\forall^{\blacksquare}$-bound polymorphic variables; it also needs to be able to *constrain* a unification variable to be constant if its use requires it. For example, consider the term $\lambda x.\, \texttt{box}\, x$: under the λ-abstraction, x is assigned a fresh unification variable a as its type, then the use of `box` forces a to be constant and we get the inferred type $\forall^{\blacksquare} a.\, a \to \blacksquare a$ after generalisation. Finally, unification of a non-constant type like $\blacktriangleright A$ with a constant type variable should fail.

Alternatively, one can think of constant types as forming a *type class* in the sense of [WB89]. The constant quantification $\forall^{\blacksquare} a.\ \ldots$ would then be written as `forall a. Constant a => ...` in Haskell syntax.

Guilhem and I briefly discussed the semantic interpretation of this kind of polymorphism, but didn't have time to reach anything satisfying as the internship was ending. However, I have an intuition that this simple kind of `let`-polymorphism might be easily justified semantically by expanding all the `let`-bindings and considering only monomorphic types.

3.4 Examples

As a concrete example of **glam** code, let us start by defining the types of guarded recursive streams and coinductive streams, as well as associated constructors and destructors:

```
-- Guarded recursive streams
type StreamG a = a * >StreamG a

consG : forall a. a -> >StreamG a -> StreamG a
consG x s = fold (x, s)

headG : forall a. StreamG a -> a
headG s = fst (unfold s)

tailG : forall a. StreamG a -> >StreamG a
tailG s = snd (unfold s)

-- Coinductive streams
type Stream a = #StreamG a

cons x s = box (consG x (next (unbox s)))
head   s = headG (unbox s)
tail   s = box (prev (tailG (unbox s)))
```

The symbols `>` and `#` denote the ▶ and ■ modalities, respectively.

Note that while explicit type signatures are required for `consG`, `headG` and `tailG` because of the use of **fold** and **unfold**, types for `cons`, `head` and `tail` can be inferred. The inferred signatures are (up to α-renaming):

```
cons : forall #a. a -> #(Fix StreamG. a * >StreamG) -> #(Fix StreamG. a * >StreamG)
head : forall a. #(Fix StreamG. a * >StreamG) -> a
tail : forall a. #(Fix StreamG. a * >StreamG) -> #(Fix StreamG. a * >StreamG)
```

where `forall #a` means $\forall^{\blacksquare} a$, and `Fix` means μ. Since `StreamG` and `Stream` are only type *synonyms*, they appear fully expanded. Notice here that the polymorphic variable a in the signature for `cons` gets a constant quantification, because we are boxing a value of type a.

Let's define a `zipWithG` function which zips two guarded streams together using a combining function:

```
zipWithG f = let { go s1 s2 = consG (f (headG s1) (headG s2))
                                    (go <*> tailG s1 <*> tailG s2) }
             in go
```

This has the inferred type:

```
zipWithG : forall a b c. (a -> b -> c) -> (Fix StreamG. a * >StreamG)
                                        -> (Fix StreamG. b * >StreamG)
                                        -> (Fix StreamG. c * >StreamG)
```

Finally, we can define the Fibonacci sequence as follows:

```
fibG = consG 0 ((\f. consG 1 (zipWithG (\x y. x + y) f <$> tailG f)) <$> fibG)
fib = box fibG
```

where `f <$> x` is desugared to `next f <*> x`.

This is similar to the classic Haskell definition `fib = 0 : 1 : zipWith (+) fib (tail fib)`, only more verbose.

We can now compute the nth Fibonacci number by evaluating `head (tail`^{n} `fib)`.

4 Conclusion

There are lots of things I didn't have time to do. In particular, **`glam`** suffers from a whole class of subtle bugs stemming from a lack of distinction between the various kinds of variables (fixed-point variables, unification variables, rigid polymorphic variables, etc.). The evaluation model is also horribly inefficient.

Nonetheless, this experience has allowed me to become more familiar with the world of research and papers, learn more about category and topos theory, logic and λ-calculus, and become more proficient with Haskell and its library ecosystem[5], as well as the implementation of functional languages in general.

My hope is that **`glam`** is at least a step in the direction of making productive programming accessible and practical. As I understand it, Guilhem has long-term plans in that direction to make the `next`, `prev`, `box`, `unbox`, `fold` and `unfold` term formers implicit.

I thank Guilhem for his supervision.

References

[AM13] Robert Atkey and Conor McBride. 'Productive Coprogramming with Guarded Recursion'. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 197–208. DOI: `10.1145/2500365.2500597`.

[Naka00] Hiroshi Nakano. 'A Modality for Recursion'. In: *Proceedings of LICS'00*. IEEE Computer Society, 2000, pp. 255–266.

[CBBB16] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl and Lars Birkedal. 'The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types'. In: *Logical Methods in Computer Science* 12.3 (2016).

[JR20] Guilhem Jaber and Colin Riba. 'Temporal Refinements for Guarded Recursive Types'. working paper or preprint. July 2020. URL: `https://hal.archives-ouvertes.fr/hal-02512655`.

[MP08] Conor McBride and Ross Paterson. 'Applicative programming with effects'. In: *Journal of Functional Programming* 18.1 (2008). DOI: `10.1017/S0956796807006326`.

[Bd00] Gavin M. Bierman and Valeria C. V. de Paiva. 'On an intuitionistic modal logic'. In: *Studia Logica* 65.3 (2000), pp. 383–416.

[BMSS12] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer and Kristian Støvring. 'First steps in synthetic guarded domain theory: step-indexing in the topos of trees'. In: *Logical Methods in Computer Science* 8.4 (2012).

[WB89] Philip Wadler and Stephen Blott. 'How to Make Ad-Hoc Polymorphism Less Ad Hoc'. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: `10.1145/75277.75283`. URL: `https://doi.org/10.1145/75277.75283`.

[5] notably Ed Kmett's `lens` and `bound` libraries, although I didn't use the latter for **`glam`**