Théo Chengkai Wang

Type-safe multi-stage programming with Lys

Computer Science Tripos – Part II

Churchill College

May, 2023

Declaration of originality

I, Théo Chengkai Wang of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed Théo Chengkai Wang

Date May 12, 2023

Proforma

Candidate Number:	**2337G**
Project Title:	**Type-safe multi-stage programming with Lys**
Examination:	**Computer Science Tripos – Part II, May, 2023**
Word Count:	**11992**[1]
Code Line Count:	**13122**[2]
Project Originator:	**The candidate and Prof. Alan Mycroft**
Supervisor:	**Prof. Jeremy Yallop and Prof. Alan Mycroft**

Original Aims of the Project

This project aims to explore the power of Contextual Modal Type Theory (CMTT) [39] for Multi-Stage Programming (MSP). To do so, we aim to design and implement an interpreter for Lys, a new language containing the relevant primitives for MSP. Lys should then be evaluated regarding its correctness, performance and expressiveness compared to other paradigms.

Work Completed

Exceeded all success criteria and completed several challenging extensions. Lys is a practical MSP language extending CMTT with recursion, System-F parametric polymorphism, existential types, algebraic datatypes, and imperative programming. Lys was also enriched with polymorphic boxes and first-class lifting. Then, I used Lys as a research platform to implement a corpus of practical, non-trivial and sometimes novel staged programs and empirically demonstrated that Lys is correctly implemented, achieves non-negligible linear speed-ups, is as expressive as the generative fragment of $\nu^{\Box}$, and has pros and cons compared to MetaOCaml. The limitations discovered can directly motivate latest CMTT research.

Special Difficulties

None.

[1]This word count was computed using `texcount` (`https://app.uio.no/ifi/texcount`), with `%TC:group table 0 1` and `%TC:group tabular 1 1` to include tables.

[2]This line count was completed with `cloc` for OCaml code, and `git ls-files | grep '.lys$' | xargs cat | wc -l` for Lys code.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to my supervisors, Professors Jeremy Yallop and Alan Mycroft, for their passion for the subject, kindness and support throughout this journey. I am also grateful to Yulong Huang for his always insightful advice. Additionally, I am sincerely grateful to my Director of Studies, Dr John Fawcett, both for his helpful comments and for his support and mentorship throughout the last three years. Finally, I would like to thank my friends, family, and Lily, who are always there for me.

Chapter 1

Introduction

This dissertation explores the power of an under-explored type-theoretical approach to multi-stage programming (MSP), both a technique in metaprogramming and a philosophy omnipresent across Computer Science. We do so by creating a language, Lys, and demonstrating its practical power. This chapter presents the motivation, the related work and the aims of the project.

1.1 Motivation

In Aho and Ullman's words, computer science is a "science of abstraction": concrete applications motivate abstract theories, abstract ideas inspire new applications. This is particularly true for programming languages, as Landin [28] and later Chatley et al. [6] argue.

Abstraction, however, does not always play well with performance, often introducing significant overheads. *Staging* is a common algorithmic technique used to eliminate this overhead, by translating higher-level to lower-level abstractions in stages. Examples include compilers, metaprogramming systems (C++ templates [1]), etc.

However, not all programs with multiple inputs are naturally separable into separate procedures, as stages tend to be more intertwined and interconnected. Thus we consider the staging of single programs, both manually and automatically. On the manual side, Lisp [32] can perform run-time metaprogramming by manipulating programs as data using `quote` and `eval`; on the automatic side, partial evaluation (PE) [21] allows for automatic symbolic evaluation of programs with respect to their known (static) inputs. However, as argued in [10], both suffer from the lack of guarantees (e.g. type preservation, variable capture (lack of *hygiene*) in Lisp, non-termination in PE ...). This is where (typed) multi-stage programming (MSP) comes in.

1.2 MSP: a taster

Consider the function `pow` computing the nth power of x.

```
let rec pow n x = if n = 0 then 1 else x * pow (n-1) x
```

In a way, `pow` provides an *abstraction.* For any n, it gives a function `int -> int` which can compute the nth power of its input. However, if we knew n in advance, this abstraction would be cumbersome: at each step, instead of just multiplying, we unroll the fixpoint, check if $n = 0$, and do a recursive call.

MSP eliminates exactly this overhead. The concept is simple: we divide this program into two *stages* and use type-theoretical constructs to force the program to be reduced exactly in that order. Stage 0 produces a piece of *compiled* code with respect to the first input n: `box (x:int |- x * x * ...  x * 1)` (n times), of type `[x:int]int` (meaning *piece of code of type int depending on a free variable x of type int*). Stage 1 can then efficiently compute x^n without any abstraction overhead.

Intuitively, we are changing the reduction order of the lambda calculus: this does not affect the result, by confluence, but does affect the number of reduction steps required, making later stages more efficient by moving the overhead to earlier ones. Two approaches emerged:

- $\lambda^{\bigcirc}$ [9], motivated by *binding time analysis* (program analysis associating binding times to each binder) in partial evaluation and based on linear temporal logic, allows the programmer to annotate the program with *binding times* (the stage where a binder's value will be known) and enforce staged execution in that order. This forms an elegantly compositional calculus of (open, i.e. with free variables) code, but lacks the ability to express that a piece of generated code is *closed* (no free variables), which motivated refinements along the line of work of MetaML [46].
- $\lambda^{\Box}$ [10], motivated by runtime code generation based on modal S4 logic, requires the programmer to make semantic preserving transformations to the program to make the modified evaluation order explicit. This calculus is limited because it enforces that all code generated be runnable, hence closed. This motivated relaxations along the line of work of contextual modal type theory (CMTT) [39].

1.3 Aims of the project

There is abundant previous work on practical multi-staged programming along MetaML's descendants [46, 47, 34, 22]. There is also abundant theoretical work on CMTT [39, 14, 42, 20]. However, until recently, very few have employed CMTT for the implementation of practical multi-staged programs like embedded DSLs or stream fusion libraries, due to the lack of a practical language based thereon. It is in this lacuna that this project aims to insert itself. In other words, I aim to answer: **how good is CMTT as a practical MSP paradigm**?

To explore this question, I design and implement (chapter 3) an OCaml-like MSP language based on CMTT (chapter 2) specifically for multi-stage programming. By implementing a large corpus of practical programs, I evaluate (chapter 4) Lys both in terms of its performance and its expressiveness, as compared to other paradigms.

Chapter 2

Preparation

This chapter presents the background theory: Multi-Stage Programming (section 2.1), the Curry-Howard correspondence (section 2.2), and its application in MSP, with modal type theories and Contextual Modal Type Theory (section 2.3). Then, section 2.4 presents the starting point and aims, section 2.5 analyses the requirements, and section 2.6 sets out the software engineering practices employed.

2.1 Multi-stage programming (MSP)

2.1.1 Informal definition

Let f be a curried function with arguments $s_1 ... s_m, d_1 ... d_n$, where $s_1 ... s_m$ are inputs known at the current stage (*static* inputs [21]), and $d_1 ... d_n$ are inputs known on a later stage (*dynamic* inputs [21]), then we want to transform f to some function f' such that $f'\ s_1\ ...\ s_m$ outputs **the code** of a specialised function $f'_{\vec{s}}$ which, when evaluated at the next stage with the dynamic input, outputs the result we wanted.

In general, in MSP, the two-stage blueprint f is generalised to any number of stages.

2.1.2 Applications

MSP has a wide range of applications, some of which we explore in this dissertation. Examples might include implementing runtime code optimisations, signal processing (e.g. compiling convolution kernels), or even machine learning (JIT compilation in the Jax library), or *anywhere exhibiting room for potential improvements by dividing the execution into stages and moving abstraction overheads to earlier stages.*

One common application is in efficiently embedding domain-specific languages (DSL). Consider a DSL D to be embedded in a host MSP language L. If we write an interpreter `interp_d: program -> input -> output` for D in L, and stage it, then we get a program `interp_d': program -> [input]output`, which is effectively a program generating code in L doing exactly what the D program does. We have thus created a **compiler** from D to L, also known as the first Futamura projection in PE [13].

Types	$A ::= 1 \mid A_1 \to A_2$
Terms	$e ::= () \mid x \mid \lambda x.e \mid e\ e'$

Figure 2.1: Syntax of the simply typed lambda calculus

In fact, MSP can be seen as a manual version of PE. I argue in section 4.2.4 that this lack of automation is in fact a virtue: the programmer has control over termination behaviour and power to use domain-specific knowledge to optimise code generation, impossible with an automatic partial evaluation black box.

Where do we find such constructs able to manually achieve partial evaluation while providing safety guarantees? This is where type systems come in.

2.2 Type systems, Logic and the Curry-Howard correspondence

Type systems are logical systems which assign properties (*types*) to pieces of programs, expressions or values (*terms*). For example, typical simply typed systems may assign the type `int` to `1`, function type $A \to B$ to $\lambda x.M$ where M is of type B if x is of type A. They form a fundamental building block of programming languages, as they facilitate programming, help avoid programmer mistakes, and provide guarantees on the program's correctness properties.

Formally, consider a simple type system for the lambda calculus, the simply typed lambda calculus (STLC), the syntax of which is presented in fig. 2.1. We define *typing judgments* as $\Gamma \vdash M : \tau$, meaning *in the context* Γ *(containing binders and their types), the term* M *has type* τ. Then, we can define the rule for functions informally presented above:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \to I$$

Strikingly, when presented under this form, certain type systems for the lambda calculus like this one can correspond exactly to logical systems. Consider implication in a natural deduction system for intuitionistic propositional logic (IPL). It is easy to see that to prove the proposition $A \to B$ we simply assume A and prove B.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to I$$

As noticed by Curry and Howard, if we take types as propositions, this rule for implication corresponds exactly to the typing rule for function types presented above. Programs are then used as proofs, and type-checking is the process of checking that a proof for a particular proposition is correct. This is the *Curry-Howard correspondence.*

An incomplete list of corresponding items between type systems and logic is presented in table 2.1, taken from [27].

Logic	Type Theory
Proposition	Type
Proof	Program
Implication	Functions
Normal Form	Value
Proof normalisation	Evaluation
...	...

Table 2.1: The Curry-Howard Correspondence as presented in II types [27]

The Curry-Howard correspondence bridges logic and theoretical computer science with the design of practical programming languages: type systems serve as elegant proof systems for certain logical systems, and more importantly for us, **logical constructs can directly motivate constructs/primitives used to formalise existing concepts and practices**. We see in section 2.3 that this is exactly how we obtained a formal type system for multi-stage programming.

2.3 Types for MSP

2.3.1 Staging the power function: hacking the evaluation order

Let us try to understand staging by considering the previous example with the power function, in the style of Davies and Pfenning [10] and Nanevski et al [37].

We start with the following and keep in mind that we want to isolate out the computation involving n and force it to be done first.

```
    let rec pow n x =
        if n = 0 then 1
        else x * pow (n-1) x
```

Here, due to the call-by-value reduction order, no evaluation is done under a lambda binding, so nothing would be evaluated given n if we kept the binder `x` outside. To get around this restriction, we pull everything involving n out and push everything involving x in. This way, all the computation involving n can be done first, independently of the value of x. We do so via semantic preserving syntax tree permutations and let-bindings.

```
    let rec pow' n =
        if n = 0 then fun x -> 1
        else let pow_n_min_1 = pow (n-1) in
            fun x -> x * pow_n_min_1 x
```

Much better. In this formulation, the stages have been made explicit. Operationally, when applied on n, it reduces to a function which **does not depend on n at all**. We have thus isolated the stages in a rather natural way: in the first stage, we have the information on n, and we can specialise the function with respect to n, before getting x in the second stage.

In fact, assuming left-to-right call-by-value operational semantics we would get:

Types	$A ::= A_1 \to A_2 \mid b \mid \Box A$
Terms	$e ::= c \mid x \mid u \mid \lambda x.e \mid e\ e' \mid \mathrm{box}(e) \mid \textbf{let box}\ u = e \text{ in } e'$
Context	$\Gamma ::= \cdot \mid \Gamma, x : A$
Modal Context	$\Delta ::= \cdot \mid \Delta, u :: A$

Figure 2.2: Syntax of $\lambda^{\Box}$

pow 2 $\rightsquigarrow^*$ `fun x -> x * (fun x -> x * (fun x -> 1) x) x`

which indeed does not depend on n. It does contain some spurious redexes, which we come back to in section 2.3.3.

2.3.2 A Modal Type Theory

In section 2.3.1, `pow` and `pow'` have the same type, and there is no indication of stages, and no way of knowing if a piece of code is of the current stage, or to be generated in the next stage. We want some construct such that this code generation at run-time becomes explicit.

This forms part of the motivation behind $\lambda^{\Box}$ [10], the syntax definition of which is as in fig. 2.2.

Davies and Pfenning took the Modal operator $\Box$ from an intuitionistic version of modal S4 logic (IS4), and used the Curry-Howard correspondence (section 2.1) to obtain a primitive which they named `box`.

Recall that in modal S4 logic [19], $\Box A$ means that A is true in all worlds reachable from the current world (because $\Box$ is reflexive and transitive over the accessibility relation). Then, a `box`, i.e. a proof of $\Box A$, corresponds to a piece of code which does not depend on the context of the current world, i.e., a piece of code without free (intuitionistic) binders. Moreover, the proof of $\Box A$ must be able to serve as a proof of A in *any* world. This naturally gives a way of composing pieces of code: unboxing a piece of code and binding it to a meta-variable u to be later spliced in.

Therefore, the typing judgement now looks like $\Delta; \Gamma \vdash e : A$, where Γ is the context of the *current* world (containing the 'normal', *object-level* variable bindings), and Δ is the modal context representing propositions true in all worlds (containing the *modal* or *meta* variables, so named because they are bound to pieces of unboxed code). We now present the box construct in terms of its introduction and elimination rules, following the tradition.

$$\frac{\Delta; \cdot \vdash e : A}{\Delta; \Gamma \vdash \textbf{box}(e) : \Box A}\ \Box I \qquad \frac{\Delta; \Gamma \vdash e : \Box A \qquad \Delta, u :: A; \Gamma \vdash e' : C}{\Delta; \Gamma \vdash \textbf{let box}\ u = e\ \textbf{in}\ e' : C}\ \Box E$$

Informally, the number of surrounding boxes indicates the stage, and no reduction happens inside boxes. Evaluation of unboxing is done with meta-substitution (as we are substituting for a meta-variable), which, unlike normal substitution, is able to cross stages (substitute into boxes).

$$\frac{}{\textbf{let box } u = \textbf{box}(e_1) \textbf{ in } e_2 \rightsquigarrow [\![e_1/u]\!]e_2}\ \text{letbox}$$

The `box` construct can be seen and implemented as a code generator [49] which generates code usable in the future stages, which is exactly what we wanted.

Now, turning back to the power function, we nicely get, using Lys syntax:

```
let rec pow: int -> [](int -> int) = fun (n:int) ->
    if n = 0 then box (fun x -> 1)
    else
        let box pow_n_min_1 = pow (n-1) in
        box (fun (x: int) -> x * pow_n_min_1 x)
```

And running this with input 2 does give, as expected:

$$\texttt{pow 2} \rightsquigarrow^* \texttt{box (fun x -> x * (fun x -> x * (fun x -> 1) x) x)}$$

This code generation procedure is still operationally unsatisfactory because of the spurious redexes. While it was argued in the original paper [10] that the elimination of such redexes could be handled by the implementation, we proceed to use a simple solution that eliminates them on the type-theoretical level.

2.3.3 Contextual Modal Type Theory (CMTT)

The requirement in $\lambda^\Box$ that all code is closed made it necessary to use a boxed function $\Box(\texttt{int} \to \texttt{int})$ and introduced redexes which cannot be symbolically reduced.

What if we could have a boxed function type where we can force the (symbolic/call-by-name) β reduction when we splice it in? This is achieved by CMTT, logically stemming from a contextual relaxation of IS4 on which $\lambda^\Box$ is based, called ICS4. Operationally, it provides an alternative **boxed function** which exactly achieves this eager substitution.

The boxed type is now written as $[\Psi]A$ (or sometimes $[\Psi \vdash A]$ [36, 20]), where Ψ is a context (a list of variables and their corresponding types). This reads: *for all worlds accessible from the current world such that the context* Ψ *is satisfied,* A *is true.* We now write **box** $(\Psi \vdash e)$ for a boxed piece of code. Then, application (which in the literature is called the "closure of the piece of unboxed open code") is expressed with the **with** keyword: u **with** (σ) where σ is an explicit substitution, i.e. a list of expressions $(e_1, e_2, ...)$ to be substituted in, of the same length as the context Ψ which u depends on (note that those expressions need not be values and are substituted in as in call-by-name semantics). The new syntax is presented in fig. 2.3.

$$\begin{array}{rl}
\text{Types} & A ::= A_1 \to A_2 \mid b \mid [\Psi]A \\
\text{Terms} & e ::= c \mid x \mid u \ \textbf{with}\ (\sigma) \mid \lambda x.e \mid e\ e' \mid \textbf{box}\ (\Psi \vdash e) \mid \textbf{let box}\ u = e \text{ in } e' \\
\text{Context} & \Gamma, \Psi ::= \cdot \mid \Gamma, x : A \\
\text{Modal Context} & \Delta ::= \cdot \mid \Delta, u :: A
\end{array}$$

Figure 2.3: Syntax of CMTT

Typing is then as in fig. 2.4.

$$\frac{}{\Delta; (\Gamma, x : A, \Gamma') \vdash x : A}\ \text{hyp} \qquad \frac{\Delta, u :: A[\Psi], \Delta'; \Gamma \vdash \sigma : \Psi}{\Delta, u :: A[\Psi], \Delta'; \Gamma \vdash u\ \textbf{with}\ \sigma : A}\ \text{ctxhyp}$$

$$\frac{\Delta; \Gamma, x : A \vdash e : B}{\Delta; \Gamma \vdash \lambda x.\ e : A \to B} \to I \qquad \frac{\Delta; \Gamma \vdash e : A \to B \quad \Delta; \Gamma \vdash e' : A}{\Delta; \Gamma \vdash e\ e' :\ B} \to E$$

$$\frac{\Delta; \cdot \vdash e : A}{\Delta; \Gamma \vdash \textbf{box}\ (\Psi \vdash e) : [\Psi]A}\ \Box I \qquad \frac{\Delta; \Gamma \vdash e : [\Psi]A \quad \Delta, u :: A[\Psi]; \Gamma \vdash e' : C}{\Delta; \Gamma \vdash \textbf{let box}\ u = e \text{ in } e' : C}\ \Box E$$

$$\frac{\Delta; \Gamma \vdash e_1 : B_1 \quad \dots \quad \Delta; \Gamma \vdash e_n : B_n}{\Delta; \Gamma \vdash (e_1, \dots, e_n) : (y_1 : B_1 \dots y_n : B_n)}\ \text{sub}$$

Figure 2.4: Typing rules of CMTT

Operationally, when unboxing and splicing a box in, we need to *close* it with an explicit substitution σ (fig. 2.4 ctxhyp rule). The unboxing is, as before, done with meta-substitution, as shown below.

$$\frac{}{\textbf{let box}\ u = \textbf{box}(\Psi \vdash e_1)\ \textbf{in}\ e_2 \rightsquigarrow [\![\Psi.e_1/u]\!]e_2}\ \text{letbox}$$

However, we define meta-substitution to take the context of the box in and traverse the expression e_2. When it encounters an occurrence of u coupled with explicit substitution σ, instead of leaving a redex, it would **close** e_1 eagerly with σ via simultaneous substitution, as follows:

$$\begin{array}{lll}
[\![\Psi.e/u]\!]\ u\ \textbf{with}\ \sigma & = [([\![\Psi.e/u]\!]\sigma)/\Psi]e & \\
[\![\Psi.e/u]\!]\ u'\ \textbf{with}\ \sigma & = u'\ \textbf{with}\ ([\![\Psi.e/u]\!]\sigma) & (u' \neq u)
\end{array}$$

This way, we get a construct very suitable for expressing open code, but which at the same time permits the evaluation of code very naturally via unboxing, as we are guaranteed that apart from what is specified in the context, the boxed expression must be closed.

```
let rec pow: int -> [x:int]int = fun (n: int) ->
    if n = 0 then box (x: int |- 1)
    else
        let box pow_n_min_1 = pow (n-1) in
        box (x: int |- x * pow_n_min_1 with (x))
```

Using exactly the same operational semantics as $\lambda^{\square}$ with different definitions for substitutions, we evaluate `pow` on 2 and obtain indeed `box (x: int |- x * x * 1)`, without any spurious redex.

In Lys, I implement CMTT with a few tweaks and additions. The process of staging programs resembles what we have seen so far: we pull everything in earlier stages out and push everything in later stages inside boxes. We present the implementation of a few example programs, e.g. staged interpreters, staged regular expression matchers [38], and as we see in section 4.3, staging is often not as straightforward as it seems.

2.4 Starting Point

The starting point is the same as declared in the proposal. I had no previous experience in writing interpreters, compilers or designing/implementing programming languages, nor with doing so in OCaml, apart from the relevant courses[1]. Prior to the project, I read related literature and played with MSP languages, but never implemented MSP mechanisms.

2.5 Requirements Analysis

The project's requirements are largely based on the success criteria stated in the proposal. We choose to implement CMTT over $\nu^{\square}$ because the former is a refinement of the later. The main goal being the evaluation of CMTT's practical strength, we choose to implement an OCaml-like language as a research platform on top of the barebone type theory that Nanevski et al. [39] present. This section overviews the refinements and choices made in the project planning stage.

Because extending type theories can have unforeseeable complexities, the Core deliverables focus only on **implementing a full interpreter for a Turing-complete extension of CMTT** with basic constructs, primitives, simple composite types (binary products and sums) and algebraic datatypes (stretch requirement which turned out essential). This includes a lexer & parser, an implementation of the CMTT type system and an evaluator.

OCaml-like features like polymorphism and data abstraction are left as **extension deliverables**, for they are not critical to the project's success, but broaden the scope of the evaluation. Unfortunately, the implementation of certain features conflicts directly with other features, and trade-offs were made with respect to their relevance to evaluation: for example, System-F style polymorphism renders type inference undecidable [48], and generatively polymorphic boxes make it hard to perform code pattern-matching without complex extensions [20], but they enable the expression of complex, real-world applications like stream fusion.

[1]IA Foundations of Computer Science (FoCS), IB Semantics, Compiler Construction, Concepts of Programming Languages, II Type Theory, Denotational Semantics, Category Theory, Optimising Compilers

Phase	ID	Deliverable	Priority	Risk
PREP.	LIT	Literary review	Critical	Low
	SPECS	Language specs	Critical	Low
	FEAS	Feasibility of extensions	Medium	Low
IMPL.	CORE-parser	Lexer and Parser	Critical	Low
	CORE-types	Type checker	Critical	Low
	CORE-int	Interpreter for the core language	Critical	Low
	EXT-adt	Abstract Datatypes and Pattern-Matching	Critical	Medium
	EXT-usab	Appropriate extensions for usability (support for strings, a top-level REPL, etc.)	Medium	Low
	EXT-poly	Generative polymorphism (System F-style)	Low	High
	EXT-ref	References and Imperative programming	Low	High
	EXT-exist	Existential types	Low	High
EVAL.	CORR	Test the correctness of the project	Critical	Low
	PROG-main	Program to test: corpus of programs to test (regular expression matcher, convolution, list processing, interpreters)	Critical	Medium
	PROG-ext	Program to test using the extensions	Low	Medium
	QUANT-perf	Quantitative: Measure the relative performance of staged vs non-staged programs	High	Low
	QUAL-diff	Qualitative: investigate the differences with other paradigms like MetaOCaml	Medium	Medium

Table 2.2: Table of deliverables and Risk Analysis

Finally, we provide a **corpus of programs** relevant to each feature and evaluate the implementation with respect to **correctness, performance benefits and expressiveness**.

Table 2.2 presents the deliverables and fig. 2.5 the dependency graph. Note that `EXT-adt` has been placed in the core deliverables, as mentioned above.

2.6 Software Engineering

2.6.1 Methodology

Based on the requirements analysis, this project is particularly amenable to the spiral model [2]. Structurally, it can be split into two phases: accomplishing the core deliverable, and using it as a research platform to explore different language features.

Due to the research adjacent nature of the project, the feasibility of both the core project (hence the choice of calculus stated in the proposal) and almost all of my extensions was unknown *a priori*. Thus, for each deliverable, I first perform a thorough risk analysis (cf. table 2.2) by reviewing the literature to ensure that (1) the feature is desirable and has particular applications and (2) it is feasible and the trade-offs are acceptable according to the design principles (section 3.1). For example, System F polymorphism was implemented because (1) it enables applications

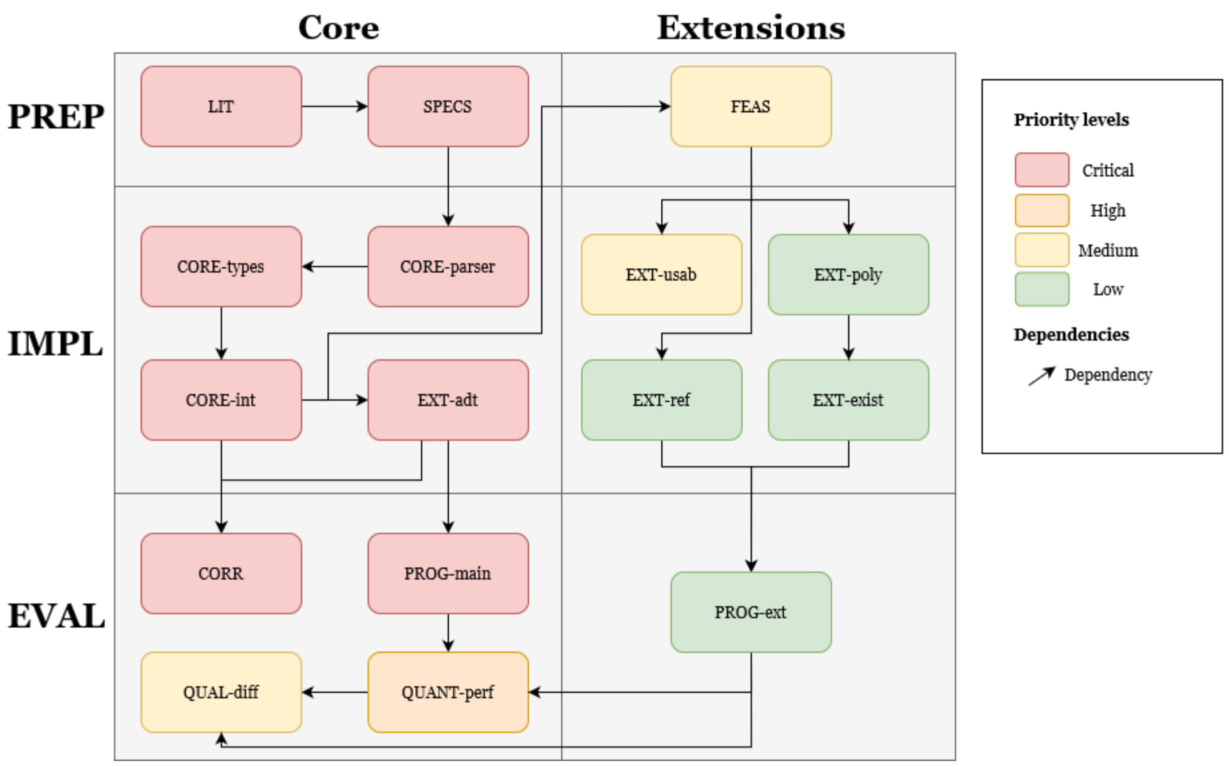


Figure 2.5: Dependency graph of the project

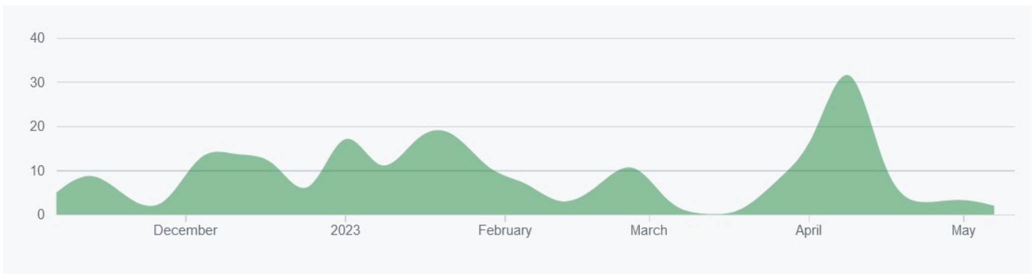


Figure 2.6: GitHub commit graph: contributions to main, excluding merge commits and bot accounts throughout the project

like stream fusion and (2) because it is feasible [20] and trade-offs are acceptable (loss of type inference and code pattern-matching).

Then, I produce a particular design, discuss the trade-offs with my supervisors during the weekly meetings, and proceed to implement it. To structure each deliverable's implementation plan, I adapt methods from the Agile model, structuring my work into two-week sprints and tracking them through GitHub's Kanban board [30] (counting a total of 72 tickets). This proved particularly fruitful when re-planning after unexpected delays or infeasible extensions, as I was able to maintain regularity of contribution throughout (fig. 2.6).

Finally, I evaluate the extension by attempting to implement example programs for the corpus, inspecting their qualitative behaviour and estimating speed-ups.

Moreover, throughout the project, I ensure that software engineering practices are respected. I use Git and GitHub for version control and standard build tools like `dune`. I use the OCaml `Core` standard library which supports an elegant module-oriented programming style, taking

Tool	Use	License
OCaml	Main implementation language	LGPL
dune	Build system	MIT
opam	OCaml's package management system	LGPL
core	Standard library	MIT
core_bench	Benchmarking library	MIT
mtime	Monotonic clock for benchmarking	ISC
ounit2	Unit tests	MIT
ppx_jane	Type-driven code generation extensions	MIT
ppx_deriving	Type-driven code generation extensions for generating a pretty-printed `show` function	MIT
menhir	Parser generator	LGPL
ppx_csv_conv	Type-driven code generation extensions for generating CSVs	MIT
Python	Language for data analysis	PSF
numpy	Vectorised computations	BSD
matplotlib	Data visualisation	PSF
pandas	Dataframe processing	BSD
scikit-learn	For simple regressions in data analysis	BSD

Table 2.3: Used tools

advantage of type-driven generation extensions (`ppx` extensions) for pretty-printing, serialisation and generating utility functions, and including docstrings and comments to ensure code readability. I also perform unit testing and make sure to include regression tests after each debugging session.

2.6.2 Language choice

For the main implementation, I made the language choice between Haskell [31] and OCaml [29] during the first work package. The former is a lazy call-by-need functional language with features such as monadic I/O etc. and the latter is a strict functional language based on ML. Although Haskell is elegant and has a much bigger community and ecosystem, **I chose OCaml** for 3 reasons. Firstly, I am more familiar with OCaml thanks to IA FoCS. Secondly, OCaml compiles faster and gives a more predictable performance, vital for having meaningful results in my evaluation section. Finally, the existence of big industry contributors to open-source libraries can make up for a smaller community as mentioned above.

For data analysis during evaluation, I used Python 3.8 because of its great support for data science applications and visualisation.

2.6.3 Tools and Licensing

I include a list of tools I used, their purposes, and their licenses in table 2.3. I open-sourced my implementation under the MIT license, which is compatible with all listed licenses, including LGPL-licensed OCaml tools thanks to their linking exception.

Chapter 3

Implementation

This chapter covers the implementation of the project in four sections. I lay down the core design and implementation principles, present an overview of Lys' design, and dive into the technical challenges. Finally, I present the implementation of the core deliverable of the project – the Lys interpreter – along with the software engineering techniques and practices employed.

3.1 Core Design Principles

Recall from section 1.3 that Lys is designed as a practical OCaml-like language extended with CMTT's features to explore its power and limitations as an MSP paradigm by *writing real programs*. With this in mind, I present some of the core design and implementation principles fueling the various subtle trade-offs made.

1. Aim for expressive power comparable to MetaOCaml, even if detrimental to purity and safety.
2. Avoid features that make it hard to extend the language, even if they are easy to implement in the interpreter.
3. Ensure that the implementation is correct and easily formally verifiable. This means that **simplicity of implementation is more important than usability or performance**, which is acceptable as the evaluation focuses on relative rather than absolute performance.
4. Avoid unnecessary formalisation, especially when correctness is evident.

3.2 Lys: from CMTT to a practical MSP language

CMTT is a theoretical calculus, suitable for proving properties but not for any practical programming. Lys, on the other hand, is a **practical OCaml-like language embodying its features**. We present its syntax in fig. 3.1.

(Program *)*

$P ::= \cdot \mid D;; P$

(Top level definitions and commands *)*

$D ::= \texttt{let}\ x : \tau = e \mid \texttt{let rec}\ f_1 : \tau_1 = e_1\ \texttt{and}\ ...f_n : \tau_n = e_n \mid e \mid \texttt{REPL\_COMMAND}$

(Expession *)*

$e ::= x \mid c \mid \texttt{fun}\ x : \tau \to e \mid e\ e' \mid \texttt{let rec}\ f_1 : \tau_1 = e_1\ \texttt{and}\ ...f_n : \tau_n = e_n\ \texttt{in}\ e' \mid ...$

(System F polymorphism with ADTs *)*

$\mid \alpha.e \mid e[\tau] \mid C[\tau]\ (e_1, ...e_n)$

(Polymorphic CMTT *)*

$\mid \texttt{box}\ (\alpha_1, ...\alpha_m; x_1 : \tau_1 ... x_n : \tau_n \vdash e) \mid \texttt{let box}\ u = e\ \texttt{in}\ e' \mid u\ \texttt{with}\ [\tau_1, ...\tau_m](e_1, ...e_m)$

$\mid \texttt{lift}\ [\tau](e)$

(Pattern matching (for ADTs) *)*

$\mid \texttt{match}\ (x_1, x_2, ..., x_n)\ \texttt{with}\ (p_{1,1}, p_{1,2}, ..., p_{1,n}) \to e_1 \mid ... \mid (p_{m,1}, p_{m,2}, ..., p_{m,n}) \to e_m$

(OCaml style Imperative Programming *)*

$\mid !e \mid e := e' \mid \texttt{ref}\ e \mid \texttt{[|}\ e_1, ...e_n\ \texttt{|]} \mid e.(e') \mid e.(e') \leftarrow e'' \mid e; e' \mid \texttt{while}\ e\ \texttt{do}\ e'\ \texttt{done}$

(Existential types *)*

$\mid \texttt{pack}\ (\exists \alpha.\tau, \tau', e) \mid \texttt{let pack}\ (\alpha, x) = e\ \texttt{in}\ e'$

Figure 3.1: Lys syntax (where the CMTT fragment is in blue)

We start with CMTT (in blue), translated to an ML-like syntax for familiarity, and type annotations because we do not yet support type inference. We provide a wrapping structure suitable for read-evaluation-print loop (REPL) implementations. A Lys program P is then defined as a list of top-level commands, D, which can be variable definitions, expressions to be executed, or commands specific to the Lys REPL.

Then, we gradually extend the language following the desiderata of practical programming languages below:

- **Turing completeness**: recursive functions
- **Basic usability**: strings, composite types (n-ary products, sum types), primitives, basic control flow etc.
- **Data structures**: polymorphic algebraic datatypes (ADT) + shallow pattern-matching [15] (which we did not extend to deep pattern matching by principle 3). Lys ADTs are implemented as OCaml's tagged variant types. [12]. Type-theoretically, they correspond to *isorecursive types*: types of the form $\mu\alpha.\tau$ where α is a type variable and τ is a type, such that the following isomorphism holds: $\mu\alpha.\tau \cong \{\mu\alpha.\ \tau/\alpha\}\tau$, (where $\{\tau_1/\alpha\}\tau_2$ corresponds to capture-avoiding type substitution). In other words, μ is a fixpoint combinator for types, and a straightforward generalisation *à la* System F_ω enables the expression of

polymorphic ADTs. More detail in appendix A.1.

- **Stateful/imperative computations**: OCaml-like imperative features like references and arrays.
- **Polymorphism/Genericity**: System F (polymorphic lambda calculus) style parametric polymorphism (extended with isorecursive types).
- **Data abstraction**: existential types of the form $\exists\alpha.\tau$ [27], which resemble OCaml's module signatures with one hidden type denoted by α. Its constructor, of the form `pack (interface_t, hidden_t, e)`, provides the hidden type and implements the signature $\exists\alpha.\tau$. The destructor `let pack('a, x) = e in e'` has similar effects with OCaml's `let open M in e` construct: it unpacks the abstracted value and exposes it to client code `e'`. More detail in appendix A.2.

Moreover, interactions between such features and contextual modal types compel us to modify CMTT as follows:

- We add first-class **lifting**, enabling serialisation of a current-stage value, e.g. type A, to a box containing its intensional representation[1], of type $\Box A$. As discussed in section 3.3.2, this construct is restricted to ground (non-functional and non-polymorphic) values and has interesting interactions with references.
- We also extend boxes to support polymorphism within them as we detail in section 3.3.1.

We explore these interactions in more detail in section 3.3.

3.3 Zooming in the design

Lys' simplicity hides the substantial effort behind its design. In this section, we discuss informally the design of several constructs, interesting not only in their practical value but also in their non-trivial interactions with the contextual box type, often inducing necessary trade-offs. A formal definition is not always presented (principle 4), but when it is, the typing sequent I use will be as follows:

$$\mathcal{A}; \Sigma; \Theta; \Delta; \Gamma \vdash e : A$$

where $\mathcal{A}$ is the store for array values or *array store*; Σ is the store for references or *store*; Θ is the type variable context or *type-var context*; Δ is the metavariable context or the *meta context* (as in CMTT); and Γ is the intuitionistic typing context, *object context*, or sometimes just *context*.

3.3.1 Parametric Polymorphism

Motivation

Parametric Polymorphism is not a required feature *per se*. CMTT is monomorphic, and any polymorphic function or data type can be defined as a series of monomorphic functions or data types, one for each use with a different type. However, this is inconvenient. Additionally, Lys

[1]i.e. we construct a syntactic representation of the value by inspecting its internal structure.

```
datatype 'a list = Nil | Cons of ('a * 'a list);;

let rec map: forall 'a. forall 'b. 'a list -> ('a -> 'b) -> 'b list =
    'a. 'b. fun (xs: 'a list) -> fun (f: 'a -> 'b) ->
        match xs with
        | Nil -> Nil['b]
        | Cons (x, xs) ->
            Cons['b] (f x, map ['a] ['b] xs f);;
```

Figure 3.2: Polymorphic map function

is a language with runtime code generation, and generating polymorphic code could greatly reduce the number of such generations to be done at run-time.

Design

I choose to achieve parametric polymorphism in a System-F-like manner, which, in the Curry-Howard sense, corresponds to second-order universal quantification (i.e. over types). In Lys, a value `e`:τ can be parameterised with type variable α (written `'a` concretely) to give a value α`.e` of type $\forall\alpha.\ \tau$. Then, we can apply one such parameterised value with type application `e[T]`. Lys also has polymorphic algebraic data types, defined similarly as in OCaml. For example, fig. 3.2 presents the definition of a polymorphic list type and a polymorphic map function.

Staging polymorphic code requires a non-trivial type-theoretical extension to CMTT. Naively, the monomorphic box can be polymorphic in two ways: `'a. box (... |- ...)` or `box (...|- 'a. ...)`. The former allows the box to be polymorphic, but any time we want to unbox it, we have to specialise this box with respect to a specific type, so we cannot achieve polymorphic code generation. The latter does allow this, but analogously to boxed functions (subsection 2.3.2), leaves spurious redexes (type applications) behind. The solution, too, is analogous. A box is henceforth defined as **box** $(\Phi; \Psi \vdash e)$ where Φ is a type-var context, and Ψ is the original context which can depend on the type-var context. Explicit substitutions u **with** σ then becomes u **with** $[\phi](\sigma)$ where ϕ represents a simultaneous type substitution.

A polymorphic box is then typed as follows:

$$\frac{\Phi \cap \Theta = \emptyset \qquad \Theta, \Phi \vdash \Psi \text{ ctx} \qquad \Theta, \Phi \vdash \tau \text{ type} \qquad \mathcal{A}; \Sigma; \Theta, \Phi; \Delta; \Psi \vdash e : \tau}{\mathcal{A}; \Sigma; \Theta; \Delta; \Gamma \vdash \textbf{box}\ (\Phi; \Psi \vdash e) : [\Phi; \Psi \vdash \tau]}\ \Box I$$

With polymorphic boxes, we can then perform the staging of the map function as in fig. 3.3. We then get, as required:

```
map_staged [int] (Cons[int] (1, Cons [int] (2, Nil[int]))
(fun (x: int) -> lift[int] x)
```

$\rightsquigarrow^*$

```
box ('b; f: (int -> 'b) |- Cons['b] (f 1, Cons['b] (f 2, Nil['b])
```

Discussion

There are a few points to note.

```
let rec map_staged:
    forall 'a. 'a list ->
        ('a -> []'a) -> ['b; f:'a -> 'b |- 'b list] =
    'a. fun (xs: ('a) list) -> fun (lift_a: 'a -> []'a) ->
        match xs with
        | Nil -> box ('b; f: 'a -> 'b |- Nil['b])
        | Cons (x, xs) ->
            let box u = map_staged ['a] xs lift_a in
            let box x = lift_a x in
            box ('b; f: 'a -> 'b |-
                Cons['b] (f (x with ()), u with ['b](f)))
;;
```

Figure 3.3: Staged polymorphic map function

- The staged polymorphic map function takes one additional argument – the `lift` function which given a value of type `'a`, converts it to a piece of code representing that value, of type `[]'a`. As explained in section 3.3.2, this is because we do not allow polymorphic lifting.

- We could have avoided all of these type-related operations by completely ignoring type annotations during evaluation. The new polymorphic box would not be necessary at all in this case. However, since the focus of the project is on exploring the type theory and not on the performance of the implementation, I opt for the more elaborate, type-theory-driven approach.

- Interestingly, many datatypes such as lists can be embedded in System F style polymorphic lambda calculi in terms of the type of their destructors, as presented in II types [27]. This is, however, cumbersome for the programmer so I opted for algebraic data types instead.

- The approach I took for the polymorphic box is similar to a less general version of that in [20], which, with complex multi-level extensions, enables code pattern-matching too. Generalising Lys' type system as such is left for future work.

3.3.2 Lifting

Motivation

To generate next-stage code containing values at the current stage of type A (as in fig. 3.3), we often need to *lift* them into boxes of type $\Box A$, which can then be unboxed and spliced in the resulting code.

In the literature, lifting often has to be done manually. Logically, this is because in modal S4 logic, $A \to \Box A$ is not true for every A. In practice, the programmer has to convert values to their intensional, syntactic representation, by traversing the data structure we want to lift. Intuitively, we should expect that both the time complexity and the space complexity of the generated code should be $O(R)$ where R is the size of the semantic representation of that value at runtime.

```
let rec lift_int_inefficient: int -> []int = fun (n: int) ->
    if n < 0 then let box u = lift_int (-n)    in box (⊢ -(u with()))
    else if n = 0 then box (|- 0)
    else let box lifted = lift_int_inefficient (n-1) in
    box (|- 1 + lifted with ())

let rec lift_int_efficient: int -> []int = fun (n: int) ->
    if n < 0 then lift_int (-n)
    else if n = 0 then box (|- 0)
    else if n = 1 then box (|- 1)
    else
        if (n % 2 = 0) then
            let box u = lift_int_efficient (n/2) in
            box (|- 2 * (u with ()))
        else
            let box u = lift_int_efficient ((n-1)/2) in
            box (|- 2 * (u with ()) + 1);;

```

Figure 3.4: Two implementations of `lift_int`

This is not only tedious but also error-prone and inefficient. Consider lifting an integer. Naively, we might write a lifting function which converts the integer into its Peano representation, which takes both linear time and space, whereas the low-level bitstring representation is only of size $O(\log(n))$ where n is the integer to be lifted. With this insight, we alleviate this cost with an implementation equivalent to considering the integer as a binary number, as shown in fig. 3.4. We have achieved the required complexity, but this is obviously still *extremely* inefficient with respect to the value's underlying implementation! How, then, can we guarantee to match the implementation's complexity and constant factor? The answer is simple: we defer this lifting to the implementation.

Design

We allow the lifting of any value which we could have lifted manually, i.e. any value which does not contain a function (we discuss this choice below), and by extension, we disallow polymorphic lifting. Given a primitive value like an integer or a character, we can convert it to a syntactic value directly and avoid the overhead of interpreting the value structurally. Given structured data like a list or a tree, no matter how it is represented at run-time, we can imagine that it is simple to do this conversion automatically, given that we can convert any non-structured value within it. Given a box, we can lift it trivially because the box already uses a syntactic representation.

Therefore, I choose to include a lift construct for any ground (non-functional and non-polymorphic) value as follows:

$$\frac{\text{ground}(\tau) \qquad \mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash e : \tau}{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash \text{lift}[\tau]\ e : \Box\tau}\ \text{lift}$$

The syntax for specifying the type of the value we are lifting conforms to type application.

```
let pow: int -> [x: int |- int] = fun (n: int) ->
    let c: [x: int |- int] ref = ref (box (x:int |- 1)) in
    let cnt: int ref = ref 0 in
    while (!cnt) < n do
        let box u = (!c) in
        c := box (x: int |- (u with (x)) * x);
        cnt := !cnt + 1
    done;
    !c;;
pow 2;;
:- box (x: int |- 1 * x * x)
```

Figure 3.5: Imperatively staged pow

Discussion

What is the issue with lifting functions?

Firstly, it is clear that we cannot lift any arbitrary function to its intensional representation within Lys, for there is no way to inspect the code inside the function [46]. Allowing this would confer an automatic `lift` construct more power than just a convenient primitive.

We could implement function lifting as a language/compiler primitive. If the language was purely generative, we could simply represent boxed code homogeneously with non-boxed code. In other words, a compiled implementation would represent boxed code using its bytecode [49]; an interpreter would represent boxed code using an AST.

However, if we want to *inspect* the code using intensional features like code pattern-matching, we must require that boxed code contain structured information like in an AST.

As code inspection is a desirable future extension, Principle 2 requires that we assume a compiled implementation with a heterogeneous code representation, to avoid making it harder to make this extension later. However, unfortunately, allowing function lifting would render the implementation significantly more complex (requiring decompilation), which is not justified by its usefulness.

3.3.3 Imperative programming

Motivation

Adding imperative programming constructs, i.e. references, arrays and loops, is desirable in that it enables both *(imperative code) generation* (see fig. 3.6, adapted from [3]) and *imperative (code generation)* (fig. 3.5).

Design

As I discuss in the evaluation section, this has been a long-standing non-trivial issue for MetaML and its descendants, but no formal presentation was given for $\lambda^{\Box}$'s descendants. Surprisingly, this extension proved to be nicely compatible with Lys' contextual box, with one caveat: we disallow lifting reference locations or arrays directly to avoid the hard-coding of global pointers

```
let rec pow: int -> [x: int, y: int ref |- unit] =
    fun (n: int) ->
    if n = 0 then box (x: int, y: int ref|- y:= 1)
    else
        let box u = pow (n-1) in
        box (x: int, y: int ref |-
            u with (x, y);
            y := x * (!y)
        );;
pow 2;;
:- box (x: int, y: int ref |- y := 1; y:= x * !y; y:= x * !y)
```

Figure 3.6: Staged imperative pow

into code boxes, for this makes the code box contain an implicit 'hole' which is not specified in the context. Instead, we lift them syntactically. For example, if location $\ell \in \Sigma$ contains value 1, then `lift[int ref] (`ℓ`) = box (|- ref 1)` and not `box (|- `ℓ`)`. Likewise, if array label `a` $\in \mathcal{A}$ points to the array `[|1, 2, 3|]`, then `lift[int array] (a) = box (|- [|1, 2, 3|])`.

Discussion

Two trade-offs were made:

- In exchange for having arrays (and without the power of dependent types), we sadly lose type safety for array indexing. This is acceptable (principle 1) as this is the **only** case in Lys' type system, and can be well handled by a runtime exception.
- Lys implements imperative programming with an OCaml-like syntax, and for simplicity (principle 3) and as this is not the focus of the project, defers memory management to OCaml's garbage collector.

3.4 Lys language implementation

According to the spiral model, each piece of design should be followed by implementation and testing. In this section, I present a brief overview of my Lys implementation. With few exceptions, I divide the implementation naturally into modules, each with one type `M.t`, and implement the utility functions around it within the same module.

3.4.1 Repository overview

The Lys interpreter is written from scratch in OCaml and follows the classical directory structure of projects using the `dune` build tool. I present in table 3.1 the repository.

The interpreter pipeline is as shown in fig. 3.7. Thanks to the careful design, implementation is mostly straightforward.

I decided to make the interpreters substitution-based, for the original aim was to make the interpreter formally verifiable. As the type theory and the extensions are very complex and

Directory	Description	LoC
`bin/main.ml`	The main interpreter program	34
`bin/repl.ml`	The REPL	97
`bin/benchmark(2).ml`	Benchmark driver code	181
`bin/(rest)*`	Debugging programs	116
`lib/parsing/*`	Lexer and Parser	455
`lib/ast/*`	Definition of the parse tree, the abstract syntax tree, and utility functions	2704
`lib/substitutions/*`	Implementation of various substitutions	963
`lib/typing/*`	Type checker	1434
`lib/interpreter/*`	Implementations of Lys' evaluators	2163
`lib/utils/*`	Utility functions	94
`lib/benchmarks/*`	Custom benchmark specification library	507
`test/example_programs/*`	Example Lys programs	1580
`test/(rest)*`	Unit and regression tests	2794
`notebooks/*`	Jupyter Notebooks for processing benchmarking data	(*ignored*)

Table 3.1: Repository overview

absolute performance does not matter, I favoured the simplest implementation by principle 3. This is a trade-off consciously made, and faster implementations with lazy substitution or *grafting* as discussed in [39] or compilation [49] are left for future work.

To simplify the implementation for substitution, we use De Bruijn indices to represent binders. We replace each string variable occurrence with a number counting the binders between that occurrence and its corresponding binder. This way, the syntax tree we get is completely invariant to α-conversion.

Therefore, the remainder of the complexity lies mainly in fiddling with De Bruijn indices, which I detail in later sections.

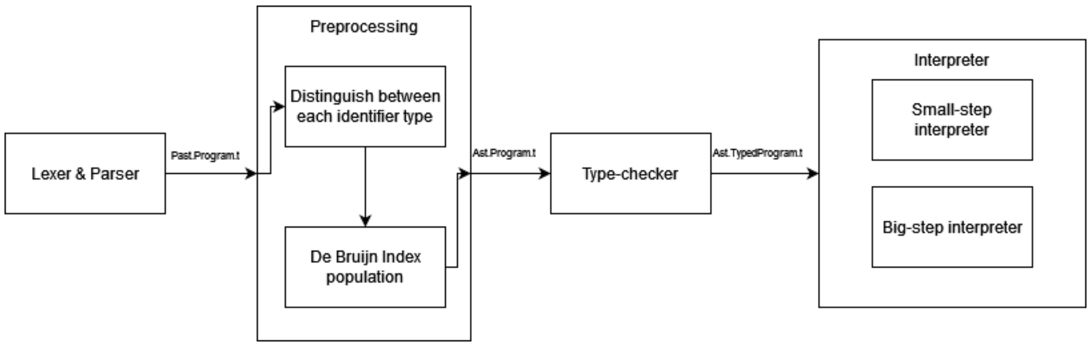


Figure 3.7: Interpreter pipeline

3.4.2 Lexer and Parser

Instead of writing a lexer and a parser by myself, I use `ocamllex` and `menhir` respectively to generate them. Both are standard OCaml tools and rely on concise specification files which make fast iteration on Lys' syntax possible:

- For every feature, I specify the new tokens in regular expression format for `ocamllex`.
- Then, I write the new production rules in terms of the new tokens in the `menhir` configuration file, which represents the language to be parsed as a context-free grammar. I specify the precedence of the relevant rules and associate each operator with appropriate associativity. Conveniently, `menhir` is an LR(1) parser generator, so I need not worry about issues such as left-recursive production rules.

3.4.3 Preprocessing

Given a parse tree of type `Past.Program.t`, we first perform a pass to distinguish between *object-level identifiers* (lambda-bound binders), *meta-level identifiers* (bound to an unboxed expression), and *type identifiers* (data type names), not distinguished by the parser. We also distinguish between object identifiers defined at the top level and those within expressions (cf. the structure of a program presented in section 3.2), which are treated differently: top-level identifiers are recorded in a context to be passed onto different places, whereas others use a substitution-based implementation.

Then, we populate the De Bruijn indices on object-level identifiers, meta-identifiers and *type variables* (polymorphic parameters, not to confuse with type identifiers). Therefore, this stage converts such a parse tree to a program of type `Ast.Program.t` by performing the above transformations and clean-ups. This is done *before* type-checking because type-checking parametric polymorphism and existential types requires type-substitution, which in turn needs De Bruijn indices.

3.4.4 Type-checker

Given an `Ast.Program.t`, the type-checking procedure simply recursively traverses the tree with the contexts specified in the typing sequent above (section 3.3) and implements the typing rules in CMTT, combined with those in System F, extended with the various features mentioned above. An example rule is in section 3.3.1. The complexity lies in the correct handling of De Bruijn indices on type variables, which need shifting in appropriate places.

3.4.5 Interpreters

Once type-checked, we feed the `Ast.TypedProgram.t` into our interpreters.

Lys provides two expression evaluators: one big-step, with performance close to the theoretical complexities, and timing facilities – on which we perform our evaluations; and one small-step reducer with a step-counter and a reduction step dumper **exactly** implementing the small-step operational semantics – used for testing and debugging purposes. Given a program, both interpreters should give **exactly the same outputs**, and this is empirically verified by the tests included under `test/test_interpreter`. As discussed above, type annotations are ignored except for type applications. Code between the two implementations is shared extensively using OCaml modules.

```
... and Pattern : sig
  type t = ...  [@@deriving sexp, show, equal, compare] (* ppx extensions
   generating utility functions*)
  val of_past : Past.Pattern.t -> t (* mutually recursive conversion
   function from Past.t *)
  ...
  val pretty_print : t -> string (* mutually recursive pretty pretting *)
end
and Expr : sig
  type t = ...  [@@deriving sexp, show, compare, equal]
  val of_past : Past.Expr.t -> t
  ...
  val pretty_print : ?alinea_size:int -> t -> string
end ...

```

Figure 3.8: Excerpt of mutually recursive module signatures in `Ast`

Given a program, the interpreter processes the definitions sequentially and evaluates the expressions with respect to the top-level contexts (object-level binders and type definitions) therein, using either of the expression evaluators. This structure enables the same implementation to be used for both evaluating Lys files and providing a REPL.

3.4.6 Software engineering techniques and practices

During the project, I exploited the powerful features of OCaml to enable maximal code reuse and extensibility. This includes

- Extensive use of unit tests and regression tests with the `OUnit2` library [40]
- The `Or_error` monad which enables uniform handling, easy propagation and composition of errors via tagging [7]. This not only facilitates interpreter debugging but also produces readable type errors for the Lys user.
- Module-oriented design of the code base: the `Ast` module is composed of mutually recursive modules `Expr`, `Typ`, etc. with each module have a unique `M.t`, enabling a natural interface-driven way to write mutually recursive utility functions (like populating De Bruijn indices or pretty-printing) on different AST components (fig. 3.8), and type-driven utility generation with `ppx` extensions.

3.5 Summary

In this chapter, I presented the design and implementation of Lys, an OCaml-like polymorphic functional multi-stage language based on CMTT. In particular, I first laid out four core design principles. Then, I presented Lys' key features and highlighted the complexity of three of them. This careful design at each stage of the spiral left a relatively straightforward implementation based on De Bruijn indices, which I then briefly presented. Finally, I mentioned some of the engineering techniques I employed to manage my substantial code base.

Chapter 4

Evaluation

We now demonstrate that Lys meets the success criteria set in section 2.5. Our implementation fulfils both the core (interpreter for a Turing-complete extension of CMTT) and the extensions (implementing OCaml-like features). We now evaluate it along the three axes specified in the proposal: **correctness** (section 4.1 have I correctly implemented Lys, meeting the success criteria?), **performance** (section 4.2 what performance gain can I expect to get by staging programs using Lys?) and **expressiveness** (section 4.3 how well can the programmer use Lys to stage a program, compared to using other paradigms?).

4.1 Correctness

The correctness of the implementation is a prerequisite to any evaluation. I provide empirical evidence for the correctness of Lys' implementation under the `test` directory of the repository, along two axes. First, I show that my type system, stemming from contextual modal logic, does inherit the correctness properties with the Curry Howard correspondence. Then, I show that both the core Lys interpreter and the extensions exhibit the expected behaviour.

4.1.1 The Lys type system

The standard way of empirically checking the correctness of the implementation of a type theory obtained with the Curry-Howard correspondence is to demonstrate the possibility of using this implementation as a *theorem prover* for the logic in question. In other words, for every valid proposition in that logic, we can provide programs which type-check in this implementation with the corresponding type. Thus, we evidence Lys' correctness by giving proofs of valid propositions in ICS4 in the style of Nanevski [39]. Note that the claims only hold in the CMTT fragment of the language, as multiple extensions make the system logically inconsistent (recursion, references, System F polymorphism, recursive types).

For example, *contextual weakening* is stated as the following Lys type (ignore the binder names x and y as they are logically meaningless and only present for type-theoretical reasons).

$$[x : C]A \to [x : C, y : D]A$$

We can provide a proof in Lys as follows, for any proposition A, C and D:

```
'a. 'c.'d. fun (z: [x:'c]'a) ->
    let box u = z in box (x: 'c, y: 'd |- u with (x));;
```

Which type-checks in Lys with the following polymorphic type (i.e. any type A, C, D would make the program type-check):

```
forall 'a. forall 'c. forall 'd. (([x:'c]'a) -> ([x: 'c, y: 'd]'a))
```

as required.

We provide similar proofs in Lys for ICS4 identities and demonstrate successful type-checking under `test/example_programs/cmtt_paper_proofs` in appendix B.1.1.

4.1.2 The Lys interpreter

I wrote **196** unit tests covering a wide range of edge cases and applied them to both the single- and multi-step implementations. This gives assurance in both the correctness and equivalence of their implementations. Moreover, we use our type system to enhance every unit test on the evaluators by instrumenting the single-step evaluator and checking for type preservation at every step.

I also include more meaningful example programs listed in table 4.1 – **over 1500** lines of Lys code under `/test/example_programs`, depending on various core or extension features – all type-checking and operationally demonstrating the required staging behaviour on both evaluators, i.e., generated well-typed code at run-time.

For example, with the single-step evaluator, we obtain the execution trace of the staged power function (section 2.3.3) applied to integer 2, and compare it to the expected result from the operational semantics. As shown in appendix B.1.2, we get **exactly the expected 16-step trace** reducing to the expected result, `box (x: int |- x * (x * 1))`.

These programs demonstrate Lys' practical applications. This includes important examples like embedding domain-specific languages (DSL) into Lys by staging their interpreters (section 4.2.2), implementing optimised efficient data structure libraries (stream fusion, section 4.3.3), as well as other smaller applications (regular expression matching, convolution as an application to signal processing in section 4.3.2 etc.).

4.2 Performance

We now turn to **the potential performance gains we can expect by staging programs in Lys**. We attempt to provide a *snapshot* of possible improvements by quantitatively analysing example staged programs, within the framework for partial evaluation (PE) proposed by [21], as programs which have been *manually partially evaluated* by our *particular* MSP system implementation.

After a brief summary of our experimental setup, we dive into the details of the performance analysis of an example, the staged WHILE language interpreter, while introducing the formal framework. Then, we present the experiments and discuss the results. Finally, we critique our method and motivate evaluation for expressiveness.

Program	Description	Dependencies	LoC
`hello_world.lys`	Staged power function	Core	12
`imperative_pow.lys`	Staged power function (imperative)	Core, Imperative	26
`poly_map.lys`	Polymorphic list and staged utility functions	Core, Polymorphism	29
`regexp.lys`	Staged regular expression matcher (from [38])	Core, Strings	160
`stream_fusion.lys`	Staged stream fusion [25]	Core, Polymorphism, Existentials, Imperative programming	294
`lambda_tagless.lys`	Staged tagless-final STLC	Core, Polymorphism	50
`while_better.lys`	Staged while language as a compiler (inspired from [44])	Core, Strings	288
`flowchart_better.lys`	Staged flowchart language as a compiler	Core, Strings	246

Table 4.1: Incomplete list of staged Lys programs

(Core corresponds to the Core deliverable (incl ADTs) defined in section 2.5)

4.2.1 Experimental setup

We choose to use the execution time as our metric for measuring speed-ups. Due to the non-determinism incurred especially by the garbage collector (GC), we cannot guarantee the execution times collected to exhibit a normal distribution (appendix B.2.1). For this reason, the benchmarks are executed using the `Core_bench` library [18] on top of which I write a thin wrapper. Designed to minimise the effects due to the GC, `Core_bench` runs benchmarks in sample batches of different sizes, stabilises the GC after each run, and performs linear regression to obtain an estimate of the run time. Furthermore, we enable bootstrapping to estimate confidence intervals for each benchmark.

The details of the platform are as follows:

- **CPU**: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
- **Memory**: 16.0 GB
- **OS**: Ubuntu 18.04 on WSL for Windows 10 19045.2846
- **OCaml version**: 4.14.0

4.2.2 A first example: staging the WHILE language

In this subsection, we walk through the process of staging and analysing the performance for the staged WHILE language interpreter. The aim is to **demonstrate what Lys can do in terms of true performance improvement**.

```
datatype expr = Int of int | Var of string | Add of (expr * expr)
    | Leq of (expr * expr) | Mul of (expr * expr);;

datatype com = Assign of (string * expr) | Seq of (com * com)
    | Cond of (expr * com * com) | While of (expr * com) | DoNothing;;

```

Figure 4.1: AST definition of the WHILE language

The WHILE language

Imagine that we need to embed a domain-specific language, WHILE, into our language Lys. The specification of the abstract syntax tree is shown in fig. 4.1.

We can then specify a program as the type `string list * string list * com`, corresponding to the inputs, outputs and program body respectively. A trivial example is the imperative `fib` program, shown in

After all this setup, it is very straightforward to write an interpreter for this language, with signature `interpret: (string list * string list * com) -> int list -> int list` (the first argument is the program the second contains the list of input values).

However, this interpreter is rather slow. How can we make it faster?

Scenario 1: fixed static input, variable dynamic input

Assume that the input program changes much less often than its inputs. Then, to make it faster we naturally think of staging the interpreter. By dividing the execution into stages, we *provably* cannot achieve a super-linear speed-up (analogously to PE [21]). However, we can eliminate the constant factor corresponding to the interpretation overhead by moving it to an earlier stage, so that the generated later-stage code only contains the necessary computation.

This makes the staged interpreter, with signature `interpret_staged: (string list * string list * com) -> [args: int list] int list`, a two-stage program which first compiles WHILE ASTs to a boxed Lys expression and then runs it. It achieves non-negligible albeit linear speed-ups, the extent of which we study. Compilation comes at a cost, but we can assume that the program is executed a sufficient number of times so that this cost is amortised away.

To give a taste of what 'compiling' means, I show in fig. 4.2 how the compiled version of the small program `INPUT i; OUTPUT i; j := 1; i := j + 1` would look like. I include the compiled version of the fib program in appendix B.2.3.

Formally, in the style of Jones [21], let p be the WHILE interpreter with static input s (the program) and dynamic input d (the arguments). We write $t_p(s,d)$ as the time used to compute $p\ s\ d$ in Lys. When staged with respect to the program, we get p_s, the compiled program, the execution time of which on input d is denoted by $t_{p_s}(d)$.

```
box (args: (int list)|-
    ...
    let zipped_env: env = ((zip Cons[string] ("i", Nil[string])) args) in
    let new_env: (env) =
        let new_store: (env) = ((update zipped_env) ("j", 1)) in
        ((update new_store)
            ("i", (((lookup new_store) "j") + ((lookup new_store) "i"))))
    in Cons[int] (((lookup new_env) "i"), Nil[int])
)

```

Figure 4.2: A WHILE program compiled to Lys

Then the *speedup function* $su_s(d)$ of the non-staged interpreter with respect to the staged interpreter is defined by

$$su_s(d) = \frac{t_p(s,d)}{t_{p_s}(d)}$$

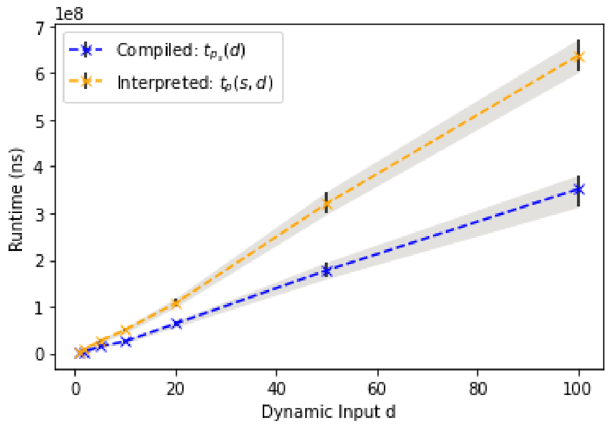
and we define the *speedup bound* of program p specialised with static input s as the limit of this value when d goes to infinity (where the *limit* of a set $A = \{a_0, a_1, a_2...\}$ is defined somewhat unconventionally as *the minimum value ℓ such that $\ell > a_i$ for all but finitely many i's.*):

$$sb(s) = \lim_{|d|\to\infty} su_s(d)$$

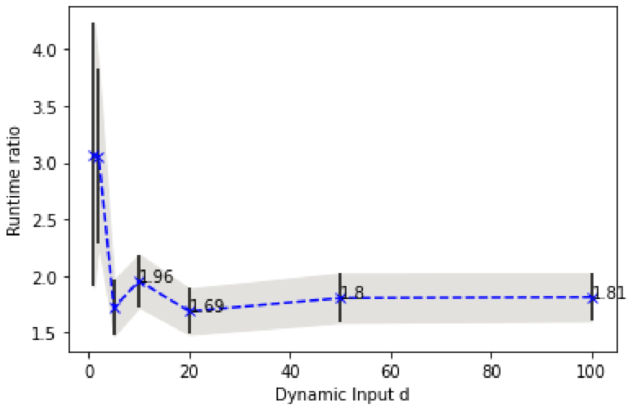
which is guaranteed to be finite because we can only achieve linear speed-ups.

Within this framework, we now study the speed-up $su_s(d)$ with respect to d with a fixed s (the `fib` function) to estimate the speed-up bound $sb(s)$ and present the results on fig. 4.3 (a) and (b).

Figure 4.3a compares the execution time of the compiled fib program and its interpreted counterpart. We can clearly observe the $O(n)$ complexity inherent to the Fibonacci program, as well as a linear speed-up of the compiled program. By dividing the two values, we obtain a curve estimating the speed-up ratio $su_s(d)$ (fig. 4.3b), which indicates a speed-up bound of around 1.8, i.e., the staged version is slightly less than twice as fast as the non-staged version.



(a) Execution time of the compiled program $t_{p_s}(d)$ and the interpreted program $t_p(s,d)$ as a function of the input d. Shared areas and dark bars denote the confidence interval.



(b) Estimate of the speed-up ratio as a function of the input d^2

Figure 4.3: Scenario 1 experiments on staging the WHILE interpreter with the Fibonacci program as the static input.

Scenario 2: variable static input and dynamic input

But what if the assumption that d changes much more often than s is wrong? Might staging still be advantageous in a single run?

Formally, we want to find the smallest d satisfying the following inequality:

$$t_{\text{compile}}(p, s) + t_{p_s}(d) \leq t_p(s, d)$$

where $t_{\text{compile}}(p, s)$ is the time to specialise p with respect to s.

In fig. 4.4, I use linear regression[1] to approximate the linear execution time curves and find a graphical solution: the smallest (integer) d_{thres} satisfying the equation is 4. In other words, similarly to just-in-time (JIT) compilation, running the two-stage process in one go is still advantageous provided that the input size d is big enough.

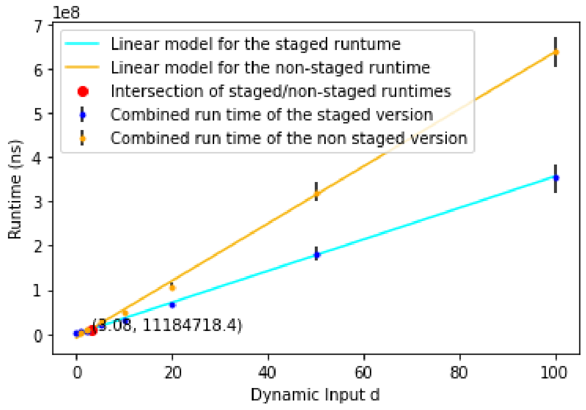


Figure 4.4: Scenario 2 experiment on staging the WHILE interpreter: time taken to run the 1-stage program VS the 2-stage program in one go

Summary

In summary, we have shown how staging the WHILE interpreter produces a compiler from WHILE to Lys, which can be advantageous in two scenarios: when the program is fixed and the input changes, it can bring a non-negligible speed-up, and when the program varies as well as the input, it can still be advantageous, provided the input is big enough for that particular program.

4.2.3 Further Experiments and Results

To demonstrate the power of Lys as an MSP language, I analyse other two-stage Lys programs, taken from the corpus of programs shown in table 4.1, and discuss the results.

p	s	Sample d	Runs
Pow	10	8192	10000
Conv	200-element int list	200-element int list	1000
Regexp	`0(a*b)*`	Accepted 100-character string	1000
While	Fibonacci	100	200
Flowchart	Fibonacci	100	100

Table 4.2: Table of benchmarks

[1]using MSE loss, which makes the assumption of Gaussian noise. This assumption is that `Core_bench` reduces the effects of GC enough to turn the remaining noise into Gaussian.

[2]Uncertainty propagated using standard methods, assuming both estimates are drawn from independent Gaussian distributions. See appendix B.2.2

[3]This evaluation does not apply, because this implementation of convolution requires both of its inputs to be of the same length, making it impossible to vary d without varying s.

[4] Obtained with linear regression because the input program (`fib`) is $O(d)$

p	Sample $t_p(s, d)$ (ns)	Sample $t_{p_s}(d)$ (ns)	$sb(s)$ estimate[2]	Sample $t_{compile} + t_{p_s}$ (ns)	d_{thres} estimate
Pow	$1.8e6 \pm 1.1e5$	$2.7e4 \pm 1.2e3$	67 ± 5.0	$2.4e6 \pm 1.5e5$	∞
Conv	$5.9e8 \pm 2.7e7$	$2.6e8 \pm 1.7e7$	2.3 ± 0.18	$5.4e8 \pm 2.7e7$	N/A[3]
Regexp	$5.5e8 \pm 2.4e7$	$5.4e6 \pm 4.8e5$	$1e2 \pm 1e1$	$5.9e6 \pm 5.3e5$	0
While	$6.4e8 \pm 3.4e7$	$3.5e8 \pm 3.6e7$	1.8 ± 0.21	$3.6e8 \pm 3.7e7$	4[4]
Flowchart	$2.3e9 \pm 2.8e8$	$9.9e8 \pm 6.3e7$	2.3 ± 0.32	$1.0e8 \pm 6.6e7$	6[4]

Table 4.3: Experiment results (with 95% confidence).

Overview

Table 4.2 presents the programs on which I performed the analyses. For each program p, I present the s value I chose to specialise the program on, a *sample* d value, as a snapshot of the actual data obtained, and the number of runs performed.

Table 4.3 presents the results obtained from experiments relevant to scenarios 1 (estimating the speed-up bound) and 2 (exploring the benefits of running both stages in one go by estimating the d_{thres}: the smallest value of d big enough to amortise away the execution cost of compilation).

Results and Discussion

By exploring scenario 1, we notice that as expected, all five benchmarks exhibit a linear speed-up, quite substantial in some cases.

Scenario 2 results show that staging is often desirable even if both inputs tend to change. For the majority of cases, staging is worthwhile because the d threshold is either small enough (for the two interpreters) or 0 (regexp). We explain this by the property of Lambda calculi that the number of evaluation steps depends on the evaluation order. Indeed, for the regexp case, according to our single-step evaluator, the non-staged version takes 2304 steps whereas the staged version takes only 1188 steps combined, for exactly the same inputs. In some cases, like the power function, staging does not confer an advantage regardless of the size of d ($d_{thres} = \infty$). Indeed, this happens because the cost of `pow` does not grow when d grows.

4.2.4 Caveats

So far, we have presented a snapshot of Lys magic and shown that **staging programs in Lys enables substantial linear speed-ups** while maintaining type safety. This, however, is an optimistic view: we ignored three very important limitations which make Lys less powerful than it seems.

Not all *programs* are equal

Firstly, partial-evaluation-capable systems like Lys cannot exhibit speed-ups on *any* program. Even amongst programs p with two inputs s and d, the speed-up we can achieve by specialising p with respect to s depends on how s influences the run-time of the program. In fact, some programs simply cannot give any speed-up.

```
datatype expr = Int of int | Var of string
              | Add of (expr * expr) | Leq of (expr * expr);;
datatype command = Goto of int | Assign of (string * expr)
                 | If of (expr * int * int) | Return of expr;;

```

Figure 4.5: Flowchart AST in Lys

For example, consider the $O(s + d)$ program that takes `int` arguments s and d and returns d^{s+d}. Then, $sb(s) = \lim_{d\to\infty} \frac{ks+k'd}{\text{constant}+k'd} = 1$ so we cannot expect to achieve any speed-up.

In fact, generalising the proof in [21], if p's run-times are additive, i.e. $t_p \in \mathcal{O}(f(|s|)g(|d|) + h(|d|))$ such that $g \in o(h)$, then we cannot achieve any speed-up because the factor in $|s|$ becomes a constant.

Not all *programmers* are equal

Secondly, Lys is a programming language. How much gain we get from it *strongly correlates with the skills of the programmer.*

Firstly, staging certain programs can be challenging to get right or require non-trivial manual program transformations.

For example, staging the interpreter for the Flowchart language was non-trivial. Flowchart [16] is a simple imperative language, differing from other such languages in that it has unstructured control flow (fig. 4.5). Staging its interpreter is thus challenging because unstructured jumps create arbitrary cycles in the program's flow graph. Naive staging traversing it depth-first gets into an infinite loop. One solution is to compile *by basic block* (a linear code sequence without jumps except at the entry and exit points) and create a top-level loop at run-time that handles the jumps. This, I argue is challenging to do automatically.

Moreover, usually, naive staging does not give optimal performance. In these cases, the extent to which the staged program is optimised depends on the programmer.

One example would be staging a stream processing library to achieve stream fusion. Stream fusion [8] is an automatic *deforestation* method for streams, i.e., optimises away the intermediate data structures generated when composing stream functions like `map` and `fold`, often involving compositional generation of code to alleviate the overhead. In Lys, I replicate fragments of the staged stream fusion library following the insights of Kiselyov et al. [25], which include using the pull-based representation of streams (with a hidden state and a step function), using continuation-passing style (CPS) to exploit knowledge about the static structure of the stream (with a Nil continuation and a Cons continuation), as well as Lys-specific tricks (threading through free variables of unknown types using polymorphism). The resulting implementation is highly complex (fig. 4.6c) but generates highly efficient code (fig. 4.7).

I argue (alongside Kiselyov et al. [25]) that it is precisely this sort of optimisation that is hard to do automatically, and that distinguishes MSP systems like Lys from conventional partial evaluators. Unfortunately, this also makes the previous section not only an evaluation of Lys, but also of the Author's talent as a programmer.

```
datatype ('a, 's) stream_shape = Nil | Cons of ('a * 's);;
datatype 'a stream =
    St of (exists 's. ('s * ('s -> ('a, 's) stream_shape)));;

```

(a) Unstaged pull stream

```
datatype 'a st_stream = St_staged of
    (exists 's. ([]'s * ([s: 's |- ('a, 's) stream_shape])));;

```

(b) Naively staged pull stream

```
datatype 'a st_stream_2 = St_staged_2 of (
    exists 's.
        ([]'s *
            ['fold_loop_type, 'fold_z_type, 'o;
                cons_cont: [fold_loop: 'fold_loop_type,
                    fold_z: 'fold_z_type, a: 'a, t:'s |- 'o]
                |-
                [fold_loop: 'fold_loop_type, fold_z: 'fold_z_type,
                state:'s, nil_cont: 'o |- 'o]]));;

```

(c) More efficient staged pull stream

Figure 4.6: Datatype definitions for staged stream fusion in Lys

```
fold_staged [int][int] (box (acc: int, x: int |- acc + x)) (box (|- 0)) (
    (map_staged[int][int] (box (x_a: int |- x_a * x_a)) (
        of_arr_staged [int] (box (|- [|1, 2, 3, 4, 5|])))));;

=====>*

box (|-
let rec loop: (int -> ((int * (int array)) -> int)) =
(fun (z: int) -> (fun (s: (int * (int array))) ->
    match s with
    (i, arr)->
        if ((i < (len(arr)))) then
            let el: int = (arr.(i))
            in let a2: int = (el * el)
            in ((loop (z + a2)) ((i + 1), arr))
        else z))
    in
    ((loop 0) (0, [|1, 2, 3, 4, 5|]))
)
```

Figure 4.7: Example stream pipeline and code generated by the efficient implementation of stream fusion, accomplishing *exactly the same staging* as Kiselyov et al. in section 5.1 of [25]

Not all *paradigms* are equal

Has our evaluation thence lost its meaning? Happily, this is not the case: the programmer is constrained by the expressiveness of the language, which is the focus of the next section.

4.3 Expressiveness

MSP paradigms are both powered and bottlenecked by their expressiveness. This is why improving the expressiveness of various staged calculi has been the main research direction since the seminal papers by Davies [9] and Davies and Pfenning [10].

In this section, we explore the expressiveness that Lys arms the programmer with compared to other relevant paradigms by implementing programs therefrom taken.

4.3.1 Overview

Properties of the code construct or quasiquote	Lys (CMTT)	$\nu^{\square}$	MetaOCaml
Well-typed	Yes*	Yes	Yes*
Well-scoped	Yes	Yes	Yes (except for imperative effects [3])
Hygiene	Yes	Yes	Yes
Explicit free variables	Yes	Yes	No
Symbolic evaluation	No	No	Yes
Cross-stage portability	Yes	Yes	No
Cross-stage persistence	Yes (boxes only)	Yes (boxes only)	Yes (with difficulty [46])
Type-safe code execution	Yes	Yes	NI([45])
Impure/stateful code generation	Yes	No**	Yes
Polymorphic code generation	Yes	No	Yes
Context polymorphism	NI([35, 36])	Yes	Yes
Intensional analysis	NI([20])	Yes (limited)	No

Table 4.4: Taxonomy of the code construct between Lys and two related paradigms.

Here, **Yes** means it is possible *and* implemented; **No** means it is not supported, and **NI** means it is possible but not implemented. *: Well-typed modulo the unsafe language constructs like arrays. **: The author believes it should be possible with a similar extension to CMTT.

At the core of Lys' code generation is the `box` primitive representing a piece of code, which shapes how Lys achieves staging. As we see in section 4.3.3, this reveals to be limiting.

In table 4.4, we compare Lys' code construct with two other paradigms: $\nu^{\square}$, a direct ancestor of CMTT from [37], and MetaOCaml [47], a well-known MSP language. The listed properties are those deemed to be useful in the literature [46, 41], some of which I define below:

- *Hygiene* [26]: no capturing of binders.
- *Symbolic evaluation*: manipulating code with free variables bound on the next stage.
- *Cross-stage portability* [46]: safely running different stages on different machines.
- *Cross-stage persistence (CSP)* [46]: using a value defined at an earlier stage in later stages.
- *Context polymorphism*: polymorphism over the free-variable context.
- *Intensional analysis*: inspection of pieces of code via pattern-matching.

These properties might not be compatible with each other and involve trade-offs. In the next sections, we qualitatively compare Lys's expressiveness with the two paradigms by experimenting with implementing various non-trivial programs from these paradigms in Lys.

4.3.2 Lys and $\nu^{\square}$

In this section, we demonstrate empirically that Lys can implement any staged *generative* (i.e. without intensional analysis) program one can implement in $\nu^{\square}$.

$\nu^{\square}$, briefly.

The modal calculus $\nu^{\square}$ by Nanevski is very close to CMTT, hence to Lys. Informally, the only big difference is the representation of free variables. Instead of using *local binders* in a *context*, $\nu^{\square}$ uses *global names* $X, Y \in \mathcal{N}$ (coupled with a name generation procedure to avoid duplication), represented by the $\nu X : T : e$ binding. A piece of code with free variables is then represented as **box** $(e) : \square_C T$, where e is of type T depending on a free name in the *support set* C. Finally, explicit substitutions are represented by an explicit mapping of names to expressions $\langle X \to e, Y \to ...\rangle e$.

This calculus can be extended with *support polymorphism*, defined as polymorphism over the support set, i.e. with respect to the free variables in a piece of code.

A limited form of intensional analysis with *code pattern-matching* is also possible, but we do not discuss it further.

Is Lys more expressive than $\nu^{\square}$?

Non-support-polymorphic, generative (i.e. no code pattern-matching) $\nu^{\square}$ programs are easily expressible in Lys. Informally, boxes in $\nu^{\square}$ can be simply translated by contextual boxes depending on free variables of exactly the same types: e.g. $\square_{\{X\}} B$ where X is declared of type A translates to $[x : A]B$, and the rest follows straightforwardly. To demonstrate this, we show a line-by-line translation of `pow` in fig. 4.8.

I hypothesise that support-polymorphic $\nu^{\square}$ programs can also be (inelegantly) expressed because they can be compiled into non-support-polymorphic programs by including all the names that might appear in the support set and closing them with dummy values. This is how we translate the regular expression program [38, p. 88] (appendix B.3.1): with the name S of type `string`, the auxiliary function (in CPS form) of type `regexp` $\to$

```
let pow: int -> □(int -> int) =
fun (n:int) ->
  let name X:int
  let pow': int -> □_X int =
    if m = 0 then
      box 1
    else
    let box u = pow' (m-1) in
      box (X * u)
  in
  let box v = pow' n in
  box (fun (x:int) -> <X -> x> v)
```

```
let pow: int -> [](int -> int) =
fun (n: int) ->
(* Define name X as free-var x *)
  let pow': int -> [x: int]int =
    if m = 0 then
      box (x: int |- 1)
    else
    let box u = pow' (m-1) in
      box (x: int |- x * u with (x))
  in
  let box v = pow' n in
  box (fun (x: int) -> v with (x));;
```

Figure 4.8: Translating `pow` from $\nu^{\Box}$ (on the left) [38, p. 81] to Lys (on the right)

$\forall p.(\Box_{S,p}\texttt{bool}) \to (\Box_{S,p}\texttt{bool})$ is translated to `regexp` $\to$ $[s : \texttt{string}, \texttt{free\_variables}]\texttt{bool} \to [s : \texttt{string}, \texttt{free\_variables}]\texttt{bool}$ in Lys, where `free_variables` correspond to all free names that can occur in the code box within this function. Nevertheless, there exists a more elegant embedding with context polymorphism, based on recent work by Murase et al. [36], which also resolves some of Lys' key limitations.

Finally, Lys does not support code pattern-matching, although an extension in the style of Jang et al. [20] should enable this feature.

4.3.3 Lys and MetaOCaml

Finally, we compare Lys with MetaOCaml, a diametrically opposed paradigm based on linear temporal type theories [9], which **exposes the strengths and the limits of Lys**. I do not attempt to give a comprehensive comparison, for the relationship between modal type theories and linear temporal type theories is an active research topic. However, I stress that recent works [35, 36] enabling polymorphic contexts can be partially motivated by the limitations I expose.

MetaOCaml, briefly.

MetaOCaml, based on $\lambda^{\bigcirc}$, hence linear temporal logic (LTL), achieves type-safe staging by dividing the program into different time steps, or *stages*. Stages are indicated using quotes `.<e>.` and splices `.~(e)`. Quoting code *delays* its execution to the next stage; splicing *advances* it to the previous stage. The current stage is the difference between the number of surrounding quotes and splices. The execution order is then exactly the stage order: execution only happens at stage 0. Importantly, **each stage has its own independent context**, and each stage-n variable can only be bound to a stage-n binder[5]. Figure 4.9 shows the staged power function, where constructs are highlighted by stage and variables coloured by their corresponding bindings.

With the implicit stage-specific context, MetaOCaml binds its free variables directly when the

[5]Or a stage-m binder with $m \leq n$ thanks cross-stage persistence, which we do not yet discuss to avoid confusion.

```
1  (*pow: int -> (int -> int) code*)
2  let pow n =
3      (*aux: int -> int code -> int code*)
4      let rec aux n x =
5          if n = 0 then .<1>.
6          else .<.~(x) * .~( aux (n-1) x )>.;;
7      in
8      .<fun x ->
9          .~(aux n .<x>.)>.;;
10
```

Figure 4.9: Staged power function in MetaOCaml

Yellow = stage 0; Green = stage 1; variables and corresponding bindings in the same colour

next-stage code is created. For example, stage-1 free variable `x` on line 9 is bound to the stage-1 binder on line 8. This is in contrast to Lys boxes, where free variables are explicitly carried in their types and are only instantiated when used.

This is the property that enables **symbolic evaluation**. It is expressed as functions of type `A code -> B code`, which constructs a `B code` within which the input `A code` is spliced into various places. Because the input has all its free variables statically bound when it was constructed, all free variables the output `B code` inherits will also be bound in that same context. In other words, the **output implicitly depends on the context of the input**. For example, given `n`, `aux n .<x>.` (line 9) reduces to an `int code` containing the variable `x`, which is bound to the same stage-1 binder on line 8 as the input.

Lys' Strengths

It is well known that MetaOCaml used complex and unsafe extensions to implement features inherent to Lys because of the implicit contexts for each stage. Lys supports safe code execution, imperative code generation and cross-stage persistence (CSP) out of the box. In contrast, MetaOCaml struggles to implement the first two: `run` is unsafe due to the difficulty of determining whether quoted code is closed, and stateful computation creates *scope extrusion*, as code containing free variables can escape from its scope with references. These issues were solved with complex type systems [3, 45, 24] but are not implemented due to their lack of usability [22]. Lastly, CSP is implemented but requires a complex procedure where persisted values are serialised and pointed to by a special label [22] and disables cross-stage portability because of the cross-stage dependencies.

Instead, we focus on pushing Lys to its limits by experimenting with expressing non-trivial staged MetaOCaml programs.

Pushing Lys to its limits

Lys, because of its explicit contexts, struggles to express symbolic evaluation, ubiquitous in MetaOCaml.

We first attempt to translate `A code -> B code` functions into Lys by turning every `code` into a contextual box. We must explicitly specify the same context for both the input and output

```
(* ('a code -> 'b code) -> ('a -> 'b) code *)
let back f = .<fun x -> .~(f .<x>.)>.

(* ('a -> 'b) code -> ('a code -> 'b code) *)
let forth f x = .<.~f .~x>.
```

Figure 4.10: Witnesses of the type isomorphism between `A code -> B code` and `(A -> B) code` [44]

```
let f_inefficient (x: (τ1 × τ2) code) =
    (* tuple deconstructed at stage 1*)
    ... .<match .~x with (v1, v2) -> ... v1 ... v2 ...>.

let f_efficient (x: τ1 code × τ2 code) =
    (* tuple deconstructed at stage 0*)
    match x with (c1, c2) -> ... .< ... .~c1 ... .~c2 ...>.
```

Figure 4.11: Moving the deconstruction of data structures fro stage 1 to stage 0 in MetaOCaml

boxes, lending to the following type: `[Ψ]A -> [Ψ]B`. Moreover, this function must work *for any arbitrary* context Ψ. This, however, cannot be expressed in Lys.

To find an alternative translation, we turn to type theory. MetaOCaml inherits, from LTL, the isomorphism `(A -> B) code` ≅ `A code -> B code`[6], where `code` corresponds to the ◯ modality, meaning 'true at the next stage'. The witnesses are shown in fig. 4.10. Lys, based on modal S4 logic, can only express the left-to-right implication (K axiom), corresponding to type `[x: A |- B] -> ([Ψ]A -> [Ψ]B)`. Conveniently, this implication holds for any Ψ on the right-hand side, as required. This makes the left-hand side, `[x: A |- B]`, an adequate translation. Indeed, this makes intuitive sense: regardless of what context the `A code` and `B code` both depend on, all we are doing is having a context-agnostic *template* guiding where we should splice the `A` value in the resulting `B` code.

This worked quite well: we staged the WHILE language in [44] without any problem.

However, in MetaOCaml, the full power of symbolic evaluation comes when `A` contains statically known structures, allowing us to replace the fully dynamic argument `A code` with some type `A'`, with all the static structures pulled out. For example, assume that `A` is statically known to be a tuple, e.g. `A` $= \tau_1 \times \tau_2$, with dynamic components. We can then pull this structure to stage 0 which gives `A'`$= \tau_1$ `code` $\times\ \tau_2$ `code`, and use this more efficient representation as our new argument type. This way, we can deconstruct the tuple at stage 0 instead of generating code doing that at the next stage (fig. 4.11).

This is a key feature of MetaOCaml, as it enables substantial savings combined with continuation-passing style (CPS): instead of having fully dynamic composite values of type `A code` (which has to be constructed and deconstructed dynamically), we can use CPS to make it always appear as an argument to a (continuation) function, and with the above trick, we

[6]In modal logic, the left-to-right implication is the K axiom, and the converse the K^{-1} axiom, proper to the linear time assumption.

replace `A code` with a more efficient structure `A'` with the static structures pulled out. We thus obtain $\forall\omega.$(`A'` $\to \omega$ `code`) $\to \omega$ `code`, which enables the data structure to be deconstructed at stage 0.

Unfortunately, expressing this in Lys is rather awkward. Naively translating `A' -> B code` using the previous insight gives the type $[x : \texttt{A} \vdash \texttt{B}]$. Notice that we use the non-annotated `A` as the type of the free variable. This is clearly inefficient because we will have to unwrap the data structure in our generated code at stage 1.

Alternatively, we can flatten the `A'` data structure, extract all the dynamic components, and explicitly specify the corresponding types in the result's context – this is analogous to specifying a template which indicates, for each of `A'`'s dynamic component, which *holes* it should be spliced into. If we take our previous example, the result would look like $[x_1 : \tau_1, x_2 : \tau_2 \vdash \texttt{B}]$.

This works as expected, but now, our type makes a strong assumption on the structure of the argument type `A'`. In most cases, this is acceptable, but when it comes to trying to *quantify* over the efficient representation `A'`, we discover this formulation to be unacceptable, and we have to fall back to the previous inefficient translation.

For example, consider the datatype `'a t = 'a -> B code`, where we wish to make different static assumptions for different `'a`. There is no way of translating this with the second formulation for it would explicitly fix the static assumptions for one particular instance of `'a` in the type. This is a problem encountered when translating the tagless-final embedding of STLC [4] into Lys[7], partially resolved by opting for the first formulation.

Another example is existential quantification: $\exists$`'a.'a -> B code`. It is possible to translate it using the second formulation, but this breaks the data abstraction and exposes the internals of a particular implementation of `'a`, which is not acceptable. This is exactly the issue encountered during the adaptation of stream fusion to Lys step by step following [25], when trying to use CPS to deconstruct the hidden state tuple at stage 0, and is why the result shown in fig. 4.7 still has a spurious match statement in the generated code.

Both cases justify the importance of enabling quantification (universal, existential or higher-kinded) over contexts, which motivates Murase's recent work [36].

4.4 Summary

In this chapter, we have evaluated Lys's correctness, performance and expressiveness. With a corpus of examples, we provided empirical evidence for the correctness of the type system and the evaluator. We then showed that Lys, like partial evaluation, enables substantial albeit linear speed-ups, and presented a critique of our evaluation, arguing that it is dependent on the program, the programmer, and the paradigm. Motivated by the last point, we carried out a comprehensive evaluation of Lys' expressiveness compared to $\nu^\Box$ and MetaOCaml. We demonstrated that Lys can implement any staged generative $\nu^\Box$ program. We also uncovered Lys' limitations compared to MetaOCaml, motivating the latest CMTT extensions.

[7]Tagless-final is a way of embedding languages without using tagged datatypes. Details in appendix B.3.2

Chapter 5

Conclusion

This project was a success: Lys exceeded all the core success criteria, and several challenging extensions were implemented, making Lys a **practical MSP language**.

As set out in the introduction, I explored various aspects of CMTT's power as a practical MSP paradigm by demonstrating its applications in signal processing (staged convolution), accelerating common algorithms (staged regular expression matcher), embedding DSLs (staged while, staged flowchart, staged tagless-final lambda interpreter) and optimised data structures (staged stream fusion). Some of them are novel adaptations from alternative paradigms.

I also showed that Lys can provide linear, albeit substantial, speed-ups, depending on the programmer, and demonstrated both its strengths and limitations in expressiveness compared to other paradigms. **These provide motivation for state-of-the-art CMTT extensions** [20, 36].

5.1 Lessons learnt

This project was carried out with some unexpected events leading to major working plan overhauls.

Firstly, I underestimated the complexity hidden behind CMTT's simplicity. Substantial time was consumed in reading about this long-standing field of research. As I added different constructs, the complexity piled up and I observed unexpected interactions (or their puzzling absence) between different primitives and language constructs, making certain originally mentioned extensions unfeasible. This demonstrates **how difficult it is to put pristine type theories like CMTT into practical use**.

Secondly, more time should have been allocated to the write-up. The theoretical complexity of this project contributed to the challenge of explaining the subject matter and demonstrating the power of the language, without providing an excessive amount of theoretical background.

In hindsight, it was a satisfying journey to work from barebone CMTT, attempting the design problems the latest papers expose.

5.2 Future work

Lys can be extended in various directions:

- Polymorphic, multi-level boxes with code pattern-matching [20]
- Context polymorphism [35, 36], making Lys have expressiveness close to $\lambda^{\bigcirc}$
- A Lisp-like notation where unboxing can be done without the binder [10]
- Algebraic effects typed with contextual modality [50]
- A Hindley-Milner type system [33], to enable type inference
- A MetaOCaml-style [23] runtime code generating compiler, but with Squid-like code pattern-matching [41]

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN: 0321227255.

[2] B. W. Boehm. “A spiral model of software development and enhancement”. In: *Computer* 21.5 (May 1988), pp. 61–72. ISSN: 1558-0814. DOI: 10.1109/2.59.

[3] C. CALCAGNO, E. MOGGI, and T. SHEARD. “Closed types for a safe imperative MetaML”. In: *Journal of Functional Programming* 13.3 (2003), pp. 545–571. DOI: 10.1017/S0956796802004598.

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. “Finally Tagless, Partially Evaluated”. In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 222–238. ISBN: 978-3-540-76637-7.

[5] James Chapman et al. “System F in Agda, for Fun and Profit”. In: *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings*. Porto, Portugal: Springer-Verlag, 2019, pp. 255–297. ISBN: 978-3-030-33635-6. DOI: 10.1007/978-3-030-33636-3_10. URL: https://doi.org/10.1007/978-3-030-33636-3_10.

[6] Robert Chatley, Alastair Donaldson, and Alan Mycroft. “The Next 7000 Programming Languages”. In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 250–282. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_15. URL: https://doi.org/10.1007/978-3-319-91908-9_15.

[7] *Core library*. URL: https://opensource.janestreet.com/core/.

[8] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. “Stream Fusion: From Lists to Streams to Nothing at All”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 315–326. ISBN: 9781595938152. DOI: 10.1145/1291151.1291199. URL: https://doi.org/10.1145/1291151.1291199.

[9] R. Davies. “A temporal-logic approach to binding-time analysis”. In: *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 1996, pp. 184–195. DOI: 10.1109/LICS.1996.561317.

[10] Rowan Davies and Frank Pfenning. “A Modal Analysis of Staged Computation”. In: *J. ACM* 48.3 (May 2001), pp. 555–604. ISSN: 0004-5411. DOI: 10.1145/382780.382785.

[11] Georg Fantner. *A brief introduction to error analysis and propagation - EPFL*. URL: https://www.epfl.ch/labs/lben/wp-content/uploads/2018/07/Error-Propagation_2013.pdf.

[12] Nate Foster. *CS 4110 – Programming Languages and Logics: Lecture 27: Recursive Types.* URL: `https://www.cs.cornell.edu/courses/cs4110/2012fa/lectures/lecture27.pdf`.

[13] Yoshihiko Futamura. "Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler". In: *Higher Order Symbol. Comput.* 12.4 (Dec. 1999), pp. 381–391. ISSN: 1388-3690. DOI: `10.1023/A:1010095604496`. URL: `https://doi.org/10.1023/A:1010095604496`.

[14] Murdoch Gabbay and Aleksandar Nanevski. "Denotation of syntax and metaprogramming in contextual modal type theory (CMTT)". In: *CoRR* abs/1202.0904 (2012). arXiv: `1202.0904`. URL: `http://arxiv.org/abs/1202.0904`.

[15] Jacques Garrigue. "Programming with Polymorphic Variants". In: 1998.

[16] John Hatcliff. "An Introduction to Online and Offline Partial Evaluation Using a Simple Flowchart Language". In: *Partial Evaluation.* Ed. by John Hatcliff, Torben Æ Mogensen, and Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 20–82. ISBN: 978-3-540-47018-2.

[17] K. Hinsen. "Staged Computation: The Technique You Did Not Know You Were Using". In: *Computing in Science amp; Engineering* 22.04 (July 2020), pp. 99–103. ISSN: 1558-366X. DOI: `10.1109/MCSE.2020.2985508`.

[18] Roshan James. *Core_bench: Better micro-benchmarks through linear regression.* URL: `https://blog.janestreet.com/core_bench-micro-benchmarking-for-ocaml/`.

[19] Meteja Jamnik. *Logic and proof.* URL: `https://www.cl.cam.ac.uk/teaching/2223/LogicProof/`.

[20] Junyoung Jang et al. "Moebius: Metaprogramming using Contextual Types - The stage where System F can pattern match on itself (Long Version)". In: *CoRR* abs/2111.08099 (2021). arXiv: `2111.08099`. URL: `https://arxiv.org/abs/2111.08099`.

[21] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* USA: Prentice-Hall, Inc., 1993. ISBN: 0130202495.

[22] Oleg Kiselyov. "Reconciling Abstraction with High Performance: A MetaOCaml approach". In: *Found. Trends Program. Lang.* 5.1 (2018), pp. 1–101. DOI: `10.1561/2500000038`.

[23] Oleg Kiselyov. "The Design and Implementation of BER MetaOCaml". In: *Functional and Logic Programming.* Ed. by Michael Codish and Eijiro Sumii. Cham: Springer International Publishing, 2014, pp. 86–102. ISBN: 978-3-319-07151-0.

[24] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. "Refined Environment Classifiers". In: *Programming Languages and Systems.* Ed. by Atsushi Igarashi. Cham: Springer International Publishing, 2016, pp. 271–291. ISBN: 978-3-319-47958-3.

[25] Oleg Kiselyov et al. "Stream Fusion, to Completeness". In: *CoRR* abs/1612.06668 (2016). arXiv: `1612.06668`. URL: `http://arxiv.org/abs/1612.06668`.

[26] Eugene Kohlbecker et al. "Hygienic Macro Expansion". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* LFP '86. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 151–161. ISBN: 0897912004. DOI: `10.1145/319838.319859`. URL: `https://doi.org/10.1145/319838.319859`.

[27] Neelakantan Krishnaswami. *Types.* URL: `https://www.cl.cam.ac.uk/teaching/2223/Types/`.

[28] P. J. Landin. “The next 700 Programming Languages”. In: *Commun. ACM* 9.3 (Mar. 1966), pp. 157–166. ISSN: 0001-0782. DOI: `10.1145/365230.365257`. URL: `https://doi.org/10.1145/365230.365257`.

[29] Xavier Leroy et al. “The OCaml system: Documentation and user’s manual”. In: *INRIA* 3 (), p. 42.

[30] Jeffrey K. Liker. *Toyota Way: 14 Management Principles from the World’s Greatest Manufacturer*. en. 1st Edition. New York: McGraw-Hill Education, 2004. ISBN: 9780071392310. URL: `https://www.accessengineeringlibrary.com/content/book/9780071392310`.

[31] Simon Marlow et al. “Haskell 2010 language report”. In: *Available online http://www.haskell. org/(May 2011)* (2010).

[32] John McCarthy. “LISP: A Programming System for Symbolic Manipulations”. In: *Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery*. ACM ’59. Cambridge, Massachusetts: Association for Computing Machinery, 1959, pp. 1–4. ISBN: 9781450373647. DOI: `10.1145/612201.612243`. URL: `https://doi.org/10.1145/612201.612243`.

[33] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(78)90014-4`. URL: `https://www.sciencedirect.com/science/article/pii/0022000078900144`.

[34] Eugenio Moggi et al. “An Idealized MetaML: Simpler, and More Expressive”. In: *Programming Languages and Systems*. Ed. by S. Doaitse Swierstra. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49099-9.

[35] Yuito Murase and Yuichi Nishiwaki. “Polymorphic Context for Contextual Modality”. In: *CoRR* abs/1801.09225 (2018). arXiv: `1801.09225`. URL: `http://arxiv.org/abs/1801.09225`.

[36] Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. “Contextual Modal Type Theory with Polymorphic Contexts”. In: *Programming Languages and Systems*. Ed. by Thomas Wies. Cham: Springer Nature Switzerland, 2023, pp. 281–308. ISBN: 978-3-031-30044-8.

[37] ALEKSANDAR NANEVSKI and FRANK PFENNING. “Staged computation with names and necessity”. In: *Journal of Functional Programming* 15.6 (2005), pp. 893–939. DOI: `10.1017/S095679680500568X`.

[38] Aleksandar Nanevski and Frank Pfenning. “Functional Programming with Names and Necessity”. AAI3143944. PhD thesis. USA, 2004. ISBN: 0496019651.

[39] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. “Contextual Modal Type Theory”. In: *ACM Trans. Comput. Logic* 9.3 (June 2008). ISSN: 1529-3785. DOI: `10.1145/1352582.1352591`.

[40] *Ounit2 2.2.6 · ocaml package*. URL: `https://ocaml.org/p/ounit2/2.2.6/doc/index.html`.

[41] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. “Squid: Type-Safe, Hygienic, and Reusable Quasiquotes”. In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. SCALA 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 56–66. ISBN: 9781450355292. DOI: `10.1145/3136000.3136005`.

[42] Brigitte Pientka. “Beluga: Programming with Dependent Types, Contextual Data, and Contexts”. In: *Functional and Logic Programming*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–12. ISBN: 978-3-642-12251-4.

[43] Matthias Puech. "A Contextual Account of Staged Computations". preprint on webpage at `http://cedric.cnam.fr/~puechm/draft_contextual.pdf`. 2016.

[44] Tim Sheard. "Using MetaML: a Staged Programming Language". In: vol. 1608. Apr. 1999. ISBN: 978-3-540-66241-9. DOI: `10.1007/10704973_5`.

[45] Walid Taha and Michael Florentin Nielsen. "Environment Classifiers". In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '03. New Orleans, Louisiana, USA: Association for Computing Machinery, 2003, pp. 26–37. ISBN: 1581136285. DOI: `10.1145/604131.604134`. URL: `https://doi.org/10.1145/604131.604134`.

[46] Walid Taha and Tim Sheard. "Multi-Stage Programming with Explicit Annotations". In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* PEPM '97. Amsterdam, The Netherlands: Association for Computing Machinery, 1997, pp. 203–217. ISBN: 0897919173. DOI: `10.1145/258993.259019`.

[47] Walid Taha et al. "MetaOCaml: A compiled, type-safe multi-stage programming language." In: (Jan. 2004).

[48] J.B. Wells. "Typability and type checking in System F are equivalent and undecidable". In: *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111–156. ISSN: 0168-0072. DOI: `https://doi.org/10.1016/S0168-0072(98)00047-5`. URL: `https://www.sciencedirect.com/science/article/pii/S0168007298000475`.

[49] Philip Wickline, Peter Lee, and Frank Pfenning. "Run-Time Code Generation and Modal-ML". In: *SIGPLAN Not.* 33.5 (May 1998), pp. 224–235. ISSN: 0362-1340. DOI: `10.1145/277652.277727`. URL: `https://doi.org/10.1145/277652.277727`.

[50] Nikita Zyuzin and Aleksandar Nanevski. *Contextual Modal Types for Algebraic Effects and Handlers.* 2021. arXiv: `2103.02976 [cs.PL]`.

Appendix A

More implementation

A.1 Algebraic datatypes

Design

Algebraic data types (ADT) is Lys' way of representing data structures, as user-defined combinations of composite and recursive types.

In Lys' syntax, such types are included as OCaml-like recursive tagged variant types, only declarable at the top level as follows:

```
datatype intlist = Nil | Cons of (int * intlist);;
```

where `Nil` and `Cons` are *constructors* of `intlist` values.

We then also include a first-level pattern-matching construct `match e with Nil -> ... | Cons (x, xs) -> ...` as in OCaml acting like a *destructor* of ADTs. This construct is then also extended to support matching n-ary products types, sum types, Strings (both matching the whole string, as in `match e with "some string" -> ...` and matching the string as the concatenation of a character and the rest of the string: `match e with c ++ s -> ...`).

Note that this construct does not support multi-level pattern matching yet as in `match e with Cons (Cons (x, xs), xxs) -> ...`. This trade-off was made by principle 1 because any multi-level pattern match can be expressed as a series of first-level pattern matches, and thus, we do not lose expressiveness.

Note on a more formal presentation

Formally, recursive types are usually introduced as the least fixpoint of a function from types to types [12]. To avoid confusion, we first consider monomorphic recursive types.

We use the following syntax:

$$\begin{aligned} A &::= ... \mid \mu\alpha.B \mid \alpha \\ e &::= ... \mid \text{fold } (e) \mid \text{unfold } (e) \end{aligned}$$

where α is a type variable and μ corresponds to a fix-point combinator for types.

To understand what this means, we can define capture-avoiding type substitution on types and terms in the standard way, with syntax $\{A/\alpha\}B$ and $\{A/\alpha\}e$ respectively. Then, what we mean by μ being a fix-point combinator is that for any expression $\mu\alpha.T$, we have that:

$$\mu\alpha.\ T \cong \{\mu\alpha.\ T/\alpha\}T$$

In other words, the left-hand side and right-hand side are isomorphic, by which we mean that we have witnesses *fold* and *unfold* corresponding to the introduction and elimination of μ types respectively:

$$\frac{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash e : \{\mu\alpha.T/\}T}{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash \text{fold } e : \mu\alpha.T}\ \mu I \qquad \frac{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash e : \mu\alpha.T}{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash \text{unfold } e : \{\mu\alpha.T/\}T}\ \mu E$$

Then, we can represent a list as the recursive type $\mu\alpha.\text{unit} + (\text{int} \times \alpha)$, `Nil` be represented as fold () and `Cons (x, xs)` as fold (x, xs), and a match statement becomes unfolding and case-splitting on the sum type.

This representation of recursive types is named *iso-recursive*, for the fix-point combinator $\mu\alpha.T$ is only isomorphic to the substituted type. There also exists an equi-recursive formulation, where both sides of the isomorphism are defined to be equal.

Lys has iso-recursive types because ADTs are represented as tagged variant types, and the *fold* constructs are explicitly inserted as the datatype's constructors: tagging a value (x, xs) with Cons is indeed a form of explicit folding.

Then to achieve polymorphic datatypes, we simply generalise this fixpoint operator to support kinded bindings *à la* System-$F_{\omega\mu}$ [5], i.e. support fixpoints of not only plain types but also *type constructors* (of the form `'a t` in Lys).

A.2 Existentials

Existential types form the standard way of both providing data abstraction and enabling alternative implementation for data structures such as streams [25].

Lys provides a standard implementation of existential types $\exists\alpha.\ T$, as presented in [27].

$$\frac{\Theta \vdash A \text{ type} \qquad \Theta, \alpha \vdash B \text{ type} \qquad \mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash e : \{A/\alpha\}B}{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash \textbf{pack } (\exists\alpha.B, A, e) : \exists\alpha.B}\ \exists I$$

$$\frac{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash e : \exists\alpha.B \qquad \mathcal{A};\Sigma;\Theta,\alpha;\Delta;\Gamma, x : B \vdash e' : C \qquad \Theta \vdash C \text{ type}}{\mathcal{A};\Sigma;\Theta;\Delta;\Gamma \vdash \textbf{let pack } (\alpha, x) = e \textbf{ in } e' : C}\ \exists E$$

The type $\exists\alpha.\ T$ specifies an *interface type* T which depends on *some* hidden type denoted by type variable α. Lys then enables the programmer to *pack* a value implementing the interface $\exists\alpha.B$ with hidden type A. Unpacking is done with `let pack`, which provides a binder x which corresponds to the implementation, and type variable α to denote the hidden type. The body of such an expression must be of a type independent of the hidden implementation, i.e. independent of the type variable α, to avoid the leakage of the hidden type.

This is very close to OCaml modules, where packs are modules with one hidden type `t` and unpacking corresponds to `let open M in ...`.

Appendix B

More Evaluation

B.1 Correctness

B.1.1 Type theoretical correctness: table of proofs in ICS4

Valid proposition in ICS4	Interpretation	File
$[x : C]A \to [y_1 : C, y_2 : D]A$	Context weakening	`m1.lys`
$[x_1 : C, x_2 : C]A \to [y : C]A$	Merging two assumptions	`m2.lys`
$[x : A]A$	Identity	`m3.lys`
$[x : A]B \to [y_1 : A][y_2 : B]C \to [z : A]C$	Applying Modus ponens in the box	`m4.lys`
$[]A \to A$	Reflexivity of boxes (axiom T)	`m5.lys`
$[x : C]A \to [y_1 : D][y_2 : C]A$	Contextual variant of axiom 4	`m6.lys`
$[x : C](A \to B) \to [y : D]A \to [z_1 : C, z_2 : D]B$	Contextual variant of axiom K	`m7.lys`
$[x : A](A \to B) \to [y : B]C \to [z : A]C$	Applying Modus ponens on assumptions in the context	`m8.lys`

Table B.1: Valid propositions in ICS4 and their Lys proofs

B.1.2 Staged Pow execution trace

We execute the following program

```
let rec (pow: int -> [x:int]int) = fun (n:int) ->
if n = 0 then box (x:int |- 1)
else
    let box u = pow (n-1) in
        box (x:int |-
            x * (u with (x))
        )
in
pow 2;;
```

and generate the execution trace with the single-step reducer by executing the command:

```
dune exec lys -- -i ssv -pp [file_containing_pow] > traces/pow_trace
```

and thus obtain the execution trace (abbreviated) as follows:

```
------------------------------
Reduction steps (#): 16
val: [; x: int |- int] = box (x: int|- (x{0} * (x{0} * 1)) )
------------------STEPS----------------
let rec pow: (int -> [; x: int |- int]) =
    (fun (n: int) -> (
 if ((n{0} = 0)) then box (x: int|- 1) else
        let box u =
            (pow{1} (n{0} - 1))
         in
            box (x: int|- (x{0} * (u{0} with (x{0}))))
            )))
 in
    (pow{0} 2)
=====>>>>>
((fun (n: int) -> (
if ((n{0} = 0)) then box (x: int|- 1)
 else
    let box u =
        (let rec pow: (int -> [; x: int |- int]) =
            (fun (n: int) -> (if ((n{0} = 0)) then
                box (x: int|-
                    1
                )
             else
                let box u =
                    (pow{1} (n{0} - 1))
                 in
                    box (x: int|-
                        (x{0} * (u{0} with (x{0})))
                    )))
         in
            pow{0} (n{0} - 1))
     in
        box (x: int|- (x{0} * (u{0} with (x{0}))) )))
  2)
=====>>>>>
(if ((2 = 0)) then box (x: int|- 1)
 else let box u =
        (let rec pow: (int -> [; x: int |- int]) =
            (fun (n: int) -> (if ((n{0} = 0)) then
                box (x: int|- 1)
             else let box u =
                    (pow{1} (n{0} - 1))
                 in
                    box (x: int|- (x{0} * (u{0} with (x{0}))) )))
         in
            pow{0} (2 - 1))
     in
        box (x: int|- (x{0} * (u{0} with (x{0}))) ))
```

```
=====>>>>>
(if (false) then box (x: int|- 1)
 else let box u =
        (let rec pow: (int -> [; x: int |- int]) =
            (fun (n: int) -> (if ((n{0} = 0)) then
                box (x: int|- 1)
             else
                let box u =
                    (pow{1} (n{0} - 1))
                 in
                    box (x: int|- (x{0} * (u{0} with (x{0})))))
         in
            pow{0} (2 - 1))
     in
        box (x: int|-
            (x{0} * (u{0} with (x{0})))
        ))
=====>>>>>
let box u =
    (let rec pow: (int -> [; x: int |- int]) =
        (fun (n: int) -> (if ((n{0} = 0)) then
            box (x: int|-
                1
            )
         else
            let box u =
                (pow{1} (n{0} - 1))
             in
                box (x: int|-
                    (x{0} * (u{0} with (x{0})))
                )))
     in
        pow{0} (2 - 1))
 in
    box (x: int|-
        (x{0} * (u{0} with (x{0})))
    )
=====>>>>>*
[ABRIEVIATED]
=====>>>>>
let box u =
    let box u =
        box (x: int|- 1)
     in box (x: int|- (x{0} * (u{0} with (x{0}))))
 in box (x: int|- (x{0} * (u{0} with (x{0}))) )
=====>>>>>
let box u =
    box (x: int|- (x{0} * 1))
 in
    box (x: int|- (x{0} * (u{0} with (x{0}))))
=====>>>>>
box (x: int|- (x{0} * (x{0} * 1)))
```

B.2 Performance

B.2.1 Initial experiments showing the distribution of runtimes

Clearly, as shown here, the distribution of runtimes without trying to alleviate the impact of GC is not Gaussian at all.

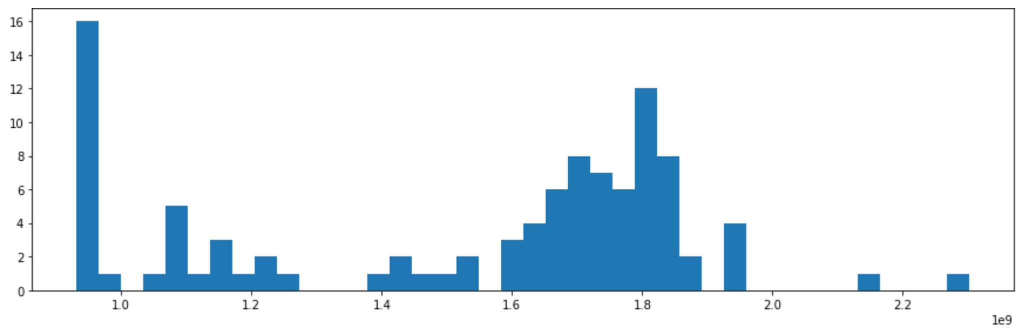


Figure B.1: Empirical distribution of execution times for the staged while interpreter specialised with `fib` applied to 100

B.2.2 Uncertainty propagation for quotients

We have obtained two estimates with confidence intervals $[t_1 - i_1; t_1 + j_1]$ and $[t_2 - i_2; t_2 + j_2]$. We now want to estimate the error bounds for the quotient, t_1/t_2.

Assuming both estimates are drawn of two independent Gaussian distributions X_1 and X_2, we can first compute the mean and standard deviation of these distributions as follows:

- The means will be taken as our estimates, respectively: $\hat{\mu}_1 = t_1$, $\hat{\mu}_2 = t_2$.
- The 95% confidence interval corresponds to a deviation of $\pm 1.96\sigma$ from the mean. Therefore, we can use i's and j's to estimate the respective standard deviations (by assuming a symmetric distribution and taking the largest of i and j as a safe approximation for 1.96σ). So we get $\hat{\sigma}_1 = \max(i_1, j_1)/1.96$ and $\hat{\sigma}_2 = \max(i_2, j_2)/1.96$.

Then, to get the new confidence interval $[t_1/t_2 - i_{new}; t_1/t_2 + j_{new}]$, it suffices to do the standard (independent Gaussian) error propagation for quotients [11]:

$$i_{new} = j_{new} = 1.96\hat{\sigma}_{new} = 1.96|\frac{\hat{\mu}_1}{\hat{\mu}_2}|\sqrt{(\frac{\hat{\sigma}_1}{\hat{\mu}_1})^2 + (\frac{\hat{\sigma}_2}{\hat{\mu}_2})^2}$$

B.2.3 Fibonacci in Flowchart

Here is the program computing the Fibonacci numbers in WHILE:

```
IN i;
OUT t1;
t1 = 1, t2 = 1;
if (i<=1){
    do_nothing;
} else {
    while (i >= 2) {
            t1 = t2 + t1; t = t1; t1 = t2; t2 = t;
            i = i - 1;
        }
    }

```

B.3 Expressiveness

B.3.1 Translating $\nu^{\square}$ to Lys

Convolution – a vanilla $\nu^{\square}$ program

We present the staged convolution function translated from [38, p. 82].

```
let rec conv_staged: intlist -> [ys: intlist] intlist -> [ys: intlist] intlist =
    fun (xs: intlist) -> fun (cont: [ys: intlist]intlist) ->
        match xs with
        | Nil -> let box u = cont in box (ys: intlist|- u with (ys))
        | Cons (x, xs) ->
            let f:[ys: intlist]intlist -> [ys: intlist]intlist = conv_staged xs in
            let box lifted_x = lift[int] x in
            let box u = cont in
            f (box (ys: intlist |-
                match ys with
                | Cons (hd, tl) -> Cons ((lifted_x with ()) * hd, u with (tl))
            ));;
```

Regexp – a support polymorphic $\nu^{\square}$ program

We present the staged regexp translated from [38, p. 88].

```
let rec acc2:(regexp -> [cont: [str:string, prev_str: string, loop: string -> bool]bool][str: string, prev_str: string, loop: string -> bool]bool) =
fun (exp: regexp) ->
    match exp with
    | Empty ->
        box (cont: [str:string, prev_str: string, loop: string -> bool]bool |-
            cont
        )
    | Plus (e1, e2) ->
        let box res1 = acc2 e1 in
        let box res2 = acc2 e2 in
        box (cont: [str:string, prev_str: string, loop: string -> bool]bool |-
            let box unboxed_res1 = (res1 with (cont)) in
            let box unboxed_res2 = (res2 with (cont)) in
            box (str: string, prev_str: string, loop: string -> bool |-
                (unboxed_res1 with (str, prev_str, loop)) || (unboxed_res2 with (str, prev_str, loop))
            )
    )
  | Times (e1, e2) ->
    let box res1_ = acc2 e1 in
    let box res2_ = acc2 e2 in
    box (cont: [str:string, prev_str: string, loop: string -> bool]bool |-
      res1_ with (res2_ with (cont))
    )
  | Const (c) ->
    let box lifted_unboxed_c = lift[char] c in
    box (cont: [str:string, prev_str: string, loop: string -> bool]bool |-
      let box unboxed_cont = cont in
      box (str: string, prev_str: string, loop: string -> bool |-
        match str with
        | "" -> false
        | x++xs ->
          (x = (lifted_unboxed_c with ())) && (unboxed_cont with (xs, prev_str, loop))
      )
    )
  | Star (e) ->
    let box res = acc2 e in
    box (cont: [str: string, prev_str: string, loop: string -> bool]bool |-
      let box unboxed_cont = cont in
      box (str: string, prev_str: string, loop: string -> bool |-
        (*Construct cont*)
        let new_cont: [str: string, prev_str: string, loop: string -> bool]bool = box (str: string, prev_str: string, loop: string -> bool |-
          if str = prev_str then false
          else loop str
        )
        in
        let box unboxed_acc2_e_new_cont = res with (new_cont) in
        let rec star_loop: string -> bool =
          fun (s: string) ->
            (unboxed_cont with (s, prev_str, loop)) ||
              (unboxed_acc2_e_new_cont with (s, s, star_loop))
        in
        star_loop str
      )
    )
;;



let accept2:(regexp->[str:string]bool) =
fun (exp:regexp) ->
  let null:[str:string, prev_str: string, loop: string -> bool]bool =
    box (str: string, prev_str: string, loop: string -> bool |-
      str = ""
    )
  in
```

```
    let box place_holder = box (|-fun (x:string) -> false) in
    let box unboxed_acc2_exp = acc2 exp in
    let box unboxed_acc_exp_null = unboxed_acc2_exp with (null) in
    box (str: string |-
        unboxed_acc_exp_null with (str, "", place_holder with ())
    )
;;
```

B.3.2 MetaOCaml

tagless-final Lambda

Tagless-final style is an alternative to using algebraic datatypes in order to embed domain-specific languages in functional host languages. It is a very elegant, extensible style, whereby instead of using tagged datatypes to present the AST, we use the *final* representation, i.e. where each DSL construct is represented as a destructor function (instead of an AST node constructor) and is hardwired to one particular representation in the host language. For example, if we choose to embed a calculator language in tagless-final style, then instead of representing '+' as the variant `Plus of expr * expr`, we represent it as `plus: repr -> repr -> repr`, for *some* `repr`.

The power of this representation is that by using the same interface, we can specify different interpreters of the same language. If we choose `repr = int`, and define `plus` as expected, applying `plus` will directly evaluate to the desired result in the host language, so we get an evaluator. If we choose `repr = string` and define `plus = fun x y -> x ^"+"^ y` then we get a pretty printer. Importantly, if we choose `repr = int code` (in MetaOCaml), we can get a compiler.

The author recommends further reading with Carette et al.'s paper [4].

I encountered the aforementioned problem when doing a tagless-final embedding of STLC in Lys using De Bruijn indices. The way this embedding works in MetaOCaml is by having binders represented as De Bruijn indices, and by having a representation of the form `('a, 'h) repr = 'h -> 'a code`, where `'h` represents a list of bindings, i.e. a *context*, which is unwrapped statically by the final embedding of De Bruijn indices.

This, when translated into Lys, does not work: we are forced to get `datatype ('a, 'h) repr = Ctx of [h: 'h |- 'a];;` (because assumptions vary for each instantiation of `'h`), which clearly has `'h` as an unannotated dynamic type, so the unwrapping has to be done dynamically.

An implementation is included under `example_programs/tagless_final/lambda_tagless.lys`.

Appendix C

Project Proposal

May 12, 2023

Project Originators: Alan Mycroft and the author

Project Supervisors: Jeremy Yallop and Alan Mycroft

Project Overseers: Alan Blackwell and Srinivasan Keshav

Introduction

Multi-stage programming or *MSP* is a metaprogramming paradigm that allows the programmer to divide the execution of a program into stages, or to "stage" it. Each stage generates code for the next phase and often specialises it with the information available at the current stage. The final stage runs the code and produces the final output [17]. This is done by having a mechanism to control the order of evaluation of the expressions [44] – operators similar to quasiquotes in Lisp (except statically typed and lexically scoped), with which one can delay or advance computations.

MSP is, in a way, analogous to program compilation & execution and partial evaluation, as presented in [34]. Intuitively, this mechanism resembles having explicit control on the order of (beta) reduction in lambda calculus so as to control the number of such reductions needed to reach β normal form. In practical terms, it is close to automatic inline expansion and partial evaluation optimisations in compilers, but is instead expressed as code annotations by the programmer.

As a metaprogramming paradigm, MSP involves the manipulation of code fragments. Usually, languages as such either have a mechanism dealing with open code (code fragments containing free variables) or closed code (code fragments where every variable is bound). They gave rise to respectively to two theoretical calculi: Davies' $\lambda^{\bigcirc}$ based on linear temporal logic dealing with open code [9] and Davies & Pfenning's $\lambda^{\Box}$ based on modal logic dealing with closed code [10].

Modern, practical approaches to MSP often tend to result from the unification the two calculi. Significant work had been done on refining $\lambda^{\bigcirc}$ to enable static typing and lexical scoping, and hence type safe execution of generated code. This is the approach taken by MetaML [46] and its derived languages like MetaOCaml [47], but it gives rise to complex type systems with rather unclear logical foundations.

Instead, we focus on the alternative approaches presented by Nanevski [37] which instead relaxes the constraints of the closed code type constructor in $\lambda^{\Box}$ with context on which variables are free in the code fragment in question. Nanevski's work on contextual modal type theory (CMTT) [39] further generalises the approach and presents a strong logical foundation for MSP. This work was then extended in various ways [20, 50, 35]. While languages (Beluga [42]) based on CMTT have been developed, they were mainly focused on the Curry-Howard isomorphism and the theorem-proving aspects of the calculus.

Project

In this project, I shall aim to create an interpreter for an ML-like pure functional, statically typed language called Lys implementing the calculus presented in Nanevski & Pfenning 2005

[37] or Nanevski 2008 [39] (also based on the denotational semantics presented in [14]) specially for the purpose of *multi-stage programming*, following a thorough evaluation of the feasibility of implementing the latter (CMTT, Nanevski 2008), during the preparation phase. This preparation is necessary because there is no clear operational semantics for CMTT, but later papers have presented denotational semantics and operational semantics for extended/generalised versions of the original CMTT.

The implementation will be done in a functional language like Haskell or OCaml, and this decision shall be made after the preparation phase.

Extensions can then include implementing the various extensions by the different papers following either approach, as will be detailed hereunder, and the choice of extensions should not depend on the choice of base calculus.

Main steps

1. Preparation: I shall conduct a thorough review of literature in the older and newer paradigms developed by Nanevski and evaluate the feasibility of creating a language based on CMTT. I shall also decide on whether to implement the language in Haskell or OCaml.
2. Implementation: I shall implement the various components of the language, in the following order, such that I can generate the test cases for the next component.
    (a) A lexer & parser (either coded or *generated*)
    (b) A type checker (this is the most technically challenging part)
    (c) An interpreter of the AST corresponding to the semantics
3. Evaluation: this phase will involve implementing a corpus of programs varying in complexity in order to evaluate correctness of the implementation as well as relative expressiveness and performance of the language as compared to other MSP languages.
4. Extend & Evaluate: implement various extensions and evaluate them.

Success criteria

- Design and implement a statically typed pure functional language based on the calculi of either Nanevski 2005 ($\nu^{\Box}$) or Nanevski 2008 (simple CMTT).
- Extend the original language with additional constructs to make it more usable. This should include at least one of:
    - Primitives,
    - Recursion/Fix point,
    - Lists, Sum types, Product types.

- Evaluate in terms of correctness, expressiveness and performance across different iterations of the language and with other MSP languages.

Evaluation strategy

Apart from testing the correctness guarantees of the type system, evaluation will be largely carried out comparatively with other existing MSP languages like Squid [41] and MetaOCaml [47].

- Correctness of the implementation: I shall evaluate qualitatively the correctness of the static and dynamic semantics by presenting a corpus of programs with various extensions I will make to the calculus, and compare it against the various examples given in related papers [38, 37, 39]. Note that the examples given by Nanevski's older approach are still to an extent applicable to CMTT, for the dynamic semantics are very similar.

- Expressiveness: This will involve the implementation of complex programs exploring the limits of expressiveness. This can also include comparing expressiveness of Lys to that of its MSP peers like MetaML/MetaOCaml and Squid/Scala by implementing the examples given in the MetaOCaml-related and Scala-related paper, like stream fusion [25] or staging domain specific languages interpreters to compilers [22].

- Performance: I shall evaluate Lys quantitatively with benchmarks (like the factor of linear speed up, or an estimation of the amortised costs of specialisation as compared to not specialising) defined by the partial evaluation community [21], with the non-staged version, as well as with other paradigms like quote/unquote in MetaML/MetaOCaml and speculative rewrite rules in Scala.

Possible extensions

Initial extensions

I am planning to implement the majority of the following extensions, depending on time constraints.

- Improve usability
  - Syntactic sugar to make it easier to program
  - Extend with algebraic data types (originally only sum and product types, and lists are built in).
  - Modules system (file based)
- A translation from staged code to non-staged OCaml
- JIT compilation of staged code (enabling additional evaluation in terms of compile time and how this differs from running it directly).

More challenging extensions

I aim to implement one of the following extensions.

- Make multi-level and System F style parameter polymorphism (Murase 2018 [35], Moebius [20])
- Pattern matching on code (Moebius [20])
- Some limited degree of type inference (might not be possible)
- Error handling (ECMTT [50])
- **TRY** Embedding MetaML in the language (there is a draft by Puech 2016 [43])
- Imperative programming (either OCaml ref or Haskell monad) – as pointed out by J. Yallop and A. Mycroft, further evaluation of the feasibility and degree of interest of this particular extension is needed.

Starting point

I had no previous experience in writing interpreters or compilers, or implementing programming languages. I had done the IA Foundations of Computer Science, IB Semantics, Compiler Construction and Concepts of Programming Languages courses, and will do the II Denotational Semantics, Type Theory and Category Theory courses and units, which I will heavily draw from. I know some functional programming and some OCaml with IA FoCS albeit without any industry experience, and no Haskell. Prior to the start of the project, I have read related literature and have played with various metaprogramming languages/libraries like Squid and MetaOCaml, but have not written any code pertaining to the implementation of MSP mechanisms.

Timetable

Week 1-2: Oct 13 - Oct 26

Work:

Review literature for CMTT and evaluate the feasibility of implementing a language based on it. Fallback to $\nu^\Box$ if not feasible.

Compare Haskell and OCaml as languages to implement this language.

Deadline (Oct 14): Project proposal

Milestone: Deliver a LateX document to supervisors comparing and evaluating the feasibility of implementation of both calculi, and the choice of the language to be used.

Week 3-4: Oct 27 - Nov 9

Work:

Practice using the chosen language by writing small programs.

Design the syntax and semantics for Lys.

Milestone: Syntax and semantics for Lys.

Week 5-6: Nov 10 - Nov 23

Work:

Implement the lexer and the parser or generate the lexer & parser.

If I do the latter, start reading through existing implementations of type checkers.

Milestone: Generate the AST from a text file.

Week 7-8: Nov 24 - Dec 7, (Dec 2 End of Michaelmas)

Work:

Implement the type checker (static semantics).

Note: if it turns out the type checker is more challenging, we can allow for 1 more week (half a work-package).

Milestone: Type checker implemented

Week 9-10: Dec 8 - Dec 21

Work:

Implement the interpreter.

Milestone: Able to run programs on the interpreter.

Week 11-12: Dec 22 - Jan 4

Work:

Write the test suite of programs for evaluation (which ones exactly to be decided according to the state of the interpreter, could include e.g. staged interpreters for DSLs or stream fusion programs.)

Read through benchmarks for partial evaluation.

Carry evaluation out.

Milestone: Prepare a corpus of programs suitable for evaluation and LaTeX or Markdown document providing a write up for evaluation carried out.

Week 13-14: Jan 5 - Jan 18, (Jan 17 Beginning of Lent)

Work:

Finish core.

Slack time & Revision.

Milestone: Pass success criteria

Week 15-16: Jan 19 - Feb 1

Work:

Write progress report.

Evaluate the feasibility of the complicated extensions.

Work on simple extensions.

Milestone: Finished writing Progress Report + Implemented some extensions + LaTeX document on feasibility of the complicated ones.

Week 17-18: Feb 2 - Feb 15

Work:

Make presentation for the progress review.

Work on more complex extensions according to the feasibility analysis.

Deadline (Feb 3): Finish progress report & Presentation.

Milestone: Progress report and presentation.

Week 19-20: Feb 16 - Mar 1

Work:

Write introduction of dissertation.

Write preparation chapter of dissertation, adapting the notes I made on language choice and calculus choice.

Milestone: Introduction and Preparation chapters of dissertation.

Week 21-22: Mar 2 - Mar 15

Work:

Work on extensions.

Start writing the implementation chapter.

Slack time & Revision for Papers 8 and 9.

Milestone: None

Week 23-24: Mar 16 - Mar 29, (Mar 17 End of Lent)

Work:

Work on extensions

Finish implementation chapter.

Milestone: Implementation chapter.

Week 25-26: Mar 30 - Apr 12

Work:

Write evaluation chapter.

Finish dissertation and send to supervisor and DoS.

Final extensions.

Slack time.

Milestone: First draft of dissertation

Week 27-28: Apr 13 - Apr 26, (Apr 25 Beginning Easter)

Work:

Slack time & Papers 8/9 Revision.

Milestone: Revision and iterate on dissertation with respect to the feedback.

Week 29-30+: Apr 27 - May 10 + May 11-12

Work:

Final corrections and submit report.

Papers 8/9 Revision.

Deadline (May 12th): Final report

Milestone: Submit code and final report.

Resource Declaration

I will use my personal laptop (Gigabyte AERO 15 2020 – i7-10750H 2.60GHz, 16GB RAM) to carry out the development phase. GitHub, OneDrive and Google Drive will be used to in addition to the MCS to perform regular backups of my repository. Software packages needed will include the open-source compiler for the OCaml or the Haskell language, and other open source software and IDEs. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.