University of Tartu
Institute of Computer Science
Computer Science Curriculum

Joosep Tavits

Implementing Temporal Resources

Master's Thesis (30 ECTS)

Supervisors:
Danel Ahman, PhD
Vesal Vojdani, PhD

Tartu 2025

Implementing Temporal Resources

Abstract:

Temporal resources – constrained not only by *how* but also by *when* they may be used – can be modeled using Fitch-style modal types. Following the approach of Ahman and Žajdela, this work develops a λ-calculus with a temporally aware type system and operational semantics. Building on *Millet*, a pure ML-like language created by Matija Pretnar, the $\lambda_{\mathsf{Mille}[\tau]}$ core calculus is formalized together with sound type-inference rules and a unification algorithm. An interpreted prototype language, *Temporal Millet*, is implemented as an extension of *Millet*. *Temporal Millet* enforces temporal safety through the type system and supports exploring realistic scenarios that require correct sequencing of time-sensitive operations. The results demonstrate both the feasibility and the challenges of temporal reasoning in effectful languages, identifying open problems such as temporal recursion and the unboxing of modally typed values within functions that perform no temporal computations. Overall, the work provides a foundation for future research toward a compiler that preserves temporal safety in compiled programs.

Keywords: Lambda calculus, temporal resources, algebraic effects, type inference

CERCS: P170 Computer science, numerical analysis, systems, control

Temporaalsete ressursside implementeerimine

Lühikokkuvõte:

Temporaalsed ressursid – mille kasutamist piirab mitte üksnes *kuidas*, vaid ka *millal* neid võib kasutada – on modelleeritavad Fitchi-stiilis modaalsete tüüpide abil. Järgides Ahmani ja Žajdela lähenemist, arendab käesolev töö ajateadliku tüübisüsteemi ja operatsioonisemantikaga λ-arvutuse. Tuginedes Matija Pretnari loodud puhta ML-laadsele keelele *Millet*, formaliseeritakse lambda-arvutus $\lambda_{\mathsf{Mille}[\tau]}$ koos korrektsete tüübituletusreeglite ja unifitseerimisalgoritmiga. Lisaks realiseeritakse *Millet*i laiendusena interpreteeritud prototüüpkeel *Temporal Millet*. *Temporal Millet* tagab tüübisüsteemi kaudu temporaalse ohutuse ning võimaldab uurida realistlikke stsenaariume, mis nõuavad ajakriitiliste operatsioonide korrektset järjestamist. Käesolev töö näitab nii temporaalsete arvutuste järgimise formaalset teostatavust kui ka sellega kaasnevaid väljakutseid efektsete keelte kontekstis, tuvastades lahendamata probleeme nagu temporaalne rekursioon ning modaalselt tüübitud väärtuste lahtipakkimine funktsioonides, mis ei teosta temporaalseid arvutusi. Kokkuvõttes loob töö aluse edasiseks uurimistööks kompilaatori suunas, mis säilitab kompileeritud programmides temporaalse ohutuse.

Võtmesõnad: Lambda-arvutus, temporaalsed ressursid, algebralised efektid, tüübituletus

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1. Introduction

Recent research has presented algebraic effects and handlers as a compelling alternative to monads as a basis for effectful computations in functional programming [1–3]. Today, many popular programming languages such as OCaml, Haskell, C and others implement handler libraries that offer novel features for practical programming [3, 4]. In addition to extensions of existing languages, new languages have been built with the purpose of testing new concepts related to programming with algebraic effects and handlers. One such example is Eff [5].

In addition to research on algebraic effects and handlers, this thesis is largely based on the recent work of Danel Ahman and Gašper Žajdela [6, 7]. In these publications, they propose that in addition to modeling *how* resources are used, algebraic effects and handlers, combined with modal types, could potentially be used to model *when* resources can be used through the use of temporal effects. These resources are classified as temporal resources, since their availability is constrained by temporal structure, and they are modeled using Fitch-style modal types. Ahman and Žajdela show that it is possible to enforce sound type-checking for such resources by defining a stateful time-aware operational semantics for temporal resources. The main motivation of their work and this thesis is to reduce the risk of programming errors when handling temporal operations. Instead of leaving the correctness of resource usage as the programmer's responsibility, they intend to provide a baseline for typing systems that detect erroneous resource usage at compile time. This provides the opportunity for safer programming in settings where certain operations are known to take a certain amount of time and proper sequencing and optimal resource usage are critical. The simplest example of such a setting could be anything that happens in an automated factory. Ahman and Žajdela discuss the example of a robotic arm that has multiple functions that are time-sensitive.

We can motivate the utility of such a typing system through additional examples that illustrate both its practical benefits and the desired programmer-facing abstractions. Consider the following resin-based 3D printing workflow, which sequentially prints a sword and a hammer:

```
let rawSwordPrint = printResinModel swordModel in
let rawHammerPrint = printResinModel hammerModel in
(* ... *)
let curedSwordPrint = uvCure rawSwordPrint in
(* ... *)
let curedHammerPrint = uvCure rawHammerPrint in
(polish curedSwordPrint, polish curedHammerPrint)
```

In resin-based 3D printing, printed models must be allowed to cool before undergoing ultraviolet (UV) post-curing. Premature curing may lead to deformation, incomplete hardening, or internal stress fractures. In the above example, the program is safe only if the invocation of `uvCure` for the sword occurs after sufficient cooling time has elapsed, here represented by the time spent printing the hammer. Otherwise, the programmer must introduce a delay or insert other time-consuming operations in place of the first comment to ensure temporal safety.

Similarly, the `uvCure` operation for the raw hammer print is valid only after the object has fully cooled. Therefore, the operation may be applied if UV curing the sword takes longer than the hammer's required cooling time. If this temporal dependency is violated, the programmer must again insert a delay or a suitable operation to ensure safe execution in place of the second comment.

Ideally, the type system should enforce these temporal constraints statically — the program should typecheck only if sufficient time elapses between operations, thereby ensuring that each stage consumes temporally valid inputs. Such guarantees would be particularly valuable in precision manufacturing pipelines, where incorrect sequencing may lead to costly defects.

The central objective of this thesis is to design and implement a prototype programming language capable of modeling such temporal resources via a combination of modal types and effectful computations. This prototype serves as a foundation for exploring the challenges of temporal reasoning in programs, allowing for the development and validation of examples in an interactive environment. The prototype is built on top of the Millet language [8] by Matija Pretnar, a minimal statically typed functional programming language designed to illustrate algebraic effects and handlers in an interactive web-based user interface. While the current implementation is not intended for industrial use, it is

structured to support future work, including the development of a compiler that translates programs into native code while preserving the safety guarantees ensured by the type system.

Our contributions are

- Extending Millet with temporal effects and operations, following the approach proposed by Ahman and Žajdela.
- Showing that the type inference and unification of the basic effectful lambda calculus of Millet are sound.
- Showing that the type inference and unification methods implemented in the extended prototype are sound and conform to the type system defined by Ahman and Žajdela with a few additions inherited from Millet.

The rest of this thesis is structured as follows. In Section 2, we give a brief overview of the theoretical basis of languages, their semantics and type systems introduced in earlier work. Section 3 explores the properties of the implementation and core calculus of the underlying language that is meant to serve as a starting point for experimenting with theoretical programming language concepts. In Section 4, we add temporal effects to the core calculus based on the work of Ahman and Žajdela and provide a brief overview of the extended core calculus. We then formalize type inference and unification in the described system and show that they conform to the typing rules presented in the extended core calculus. Section 5 gives a brief overview of implementation details and design choices. Additionally, we explore interesting sample programs and consult known limitations that we found. Finally, we present ideas for future work that did not fit into the scope of this thesis in Section 6 and conclude the thesis in Section 7.

The implementation of Temporal Millet, reviewed in Section 5, is available in Zenodo [9].

OpenAI's GPT-4.1 large language model [10] was employed to generate initial drafts of type inference rules, using a limited set of manually authored examples together with OCaml source code specifying the remaining cases. The model's output underwent comprehensive manual verification and refinement to ensure correctness and clarity, but served as an effective foundation for further development. GPT-4.1 was also applied to enhance the phrasing, readability, and overall quality of the writing.

2. Background

This chapter briefly introduces the theoretical foundations and language constructs that underpin the work presented in this thesis. We begin by reviewing the lambda calculus, a foundational model of computation and the basis for functional programming languages. Building on this, we explore evaluation strategies that are central to modeling effects in modern languages. We then turn to the type-theoretic aspects of language design, including algebraic data types and polymorphism via the *Hindley-Milner* type system.

2.1 Lambda Calculus and Evaluation Strategies

The lambda calculus, originally introduced by Alonzo Church in 1936 [11], forms the theoretical foundation of functional interfaces in programming languages. As a minimal model of computation, it captures the essence of function abstraction and application and serves as the basis for reasoning about programs. For a detailed overview of lambda calculus, including its syntax, semantics, and computational properties, the reader is referred to books on untyped and typed lambda calculus by Barendregt et al. [12, 13].

Practical programming languages enrich the original, untyped lambda calculus with features like static typing, structured data, and evaluation strategies. We are most interested in the *call-by-value* (CBV) evaluation strategy, which evaluates function arguments before application and aligns closely with most real-world functional languages such as OCaml, Standard ML, and F#, as it simplifies reasoning about effects and evaluation order [14]. The alternative is *call-by-name* (CBN) evaluation, which defers the evaluation of function arguments until their values are actually needed, potentially avoiding unnecessary computation but, on the contrary, making reasoning about effects and evaluation order more complex [14].

A more structured variant, the *fine-grain call-by-value* (FGCBV) lambda calculus, refines the CBV lambda calculus by making the distinction between values and computations explicit, enabling precise control over evaluation order and thus simplifying the modeling of computational effects [15]. This separation is generalized by Paul Blain Levy's *call-by-push-value* (CBPV) lambda calculus, which unifies CBV and CBN evaluation strategies [16].

2.2 Types and Data Structures

In addition to evaluation strategies, modern functional languages rely on *algebraic data types* (ADTs) to represent structured data and enable powerful control flow constructs.

ADTs combine the capabilities of product types (tuples, records), sum types (tagged unions or variants), and recursive types to express complex data structures in a compositional and type-safe way. These types are typically used in conjunction with pattern matching, which allows programs to deconstruct and analyze data based on its shape and constructor, offering a concise and readable alternative to explicit control flow constructs like conditionals or manual case analysis [17, 18]. ADTs play a central role in the expressiveness and safety of statically typed functional languages.

To further support reusable abstractions over data, functional languages often employ polymorphic type systems. In particular, languages like ML [19] and OCaml are built upon the *Hindley-Milner* type system [19, 20], which supports *let-polymorphism* and automatic type inference without requiring explicit type annotations. This system ensures that well-typed programs have *principal types*, the most general types assignable to expressions, and that these can be inferred algorithmically. As a result, developers benefit from both the flexibility of polymorphism and the guarantees of static typing with minimal syntactic overhead. *Hindley-Milner* remains the foundation for type inference in a wide range of modern statically typed languages, including OCaml and Haskell (with extensions).

These ingredients — FGCBV structure and evaluation strategy, ADTs, and *Hindley-Milner* style type inference — form the core of Millet [8], the language on which this thesis builds. We proceed by introducing Millet and formalizing its core calculus in the next chapter.

3. Millet

In this chapter, we closely examine the properties of Millet. We begin by reviewing notable properties of the *surface language*, the high-level syntax presented to programmers, which is desugared into the smaller core calculus $\lambda_{\mathsf{Millet}}$ with a formally defined semantics. Finally, we formalize the type inference of $\lambda_{\mathsf{Millet}}$ and show that it is sound.

Millet is a pure ML-like language designed as a baseline for building new programming languages. It was originally created to facilitate experimentation with theoretical programming language concepts without requiring the complete construction of a language from scratch. Millet does not enforce a strict FGCBV program structure at the surface level, in order to provide a smoother developer experience. However, it employs FGCBV as its core calculus.

3.1 Surface Language

The surface language of Millet supports five principal top-level constructs: *type definitions*, *top-level variable definitions*, *top-level function definitions*, *top-level recursive function definitions*, and the `run` block. The example program in Figure 1 below illustrates each:

All top-level constructs, except `run`, introduce globally scoped bindings that are accessible throughout the program, analogous to user-defined components of a standard library for the code in the `run` block. In the example, the `type` declaration defines a binary sum type `either` with constructors `Left` and `Right` injecting values of types `'l` and `'r`, respectively. The binding `let n = 5` defines a global integer constant. The function `swap` has type (α, β) `either` $\to (\beta, \alpha)$ `either` and exchanges the constructors of its argument. The recursive function `foldl`, of type $(\beta \to \alpha \to \beta) \to \beta \to \alpha$ `list` $\to \beta$, implements a standard left fold over lists. The `run` block serves as the program's entry point. The program it contains constructs a list `lst` of type `(int, string) either list`, folds over it to swap constructors, and reverses the result to restore the original element order, yielding `(Right 5) :: (Left "test") :: []`. Programs written in the surface language do not lend themselves to formal reasoning as directly as those expressed in the strict FGCBV core calculus.

The transformation from Millet's CBV surface language to its FGCBV core calculus is achieved by desugaring top-level runnable computations from CBV to FGCBV-compliant code using Algorithm 1. The algorithm takes a top-level runnable computation and

```
type ('l, 'r) either = Left of 'l | Right of 'r

let n = 5

let swap = function
  | Left l  -> Right l
  | Right r -> Left r

let rec foldl f acc = function
  | []      -> acc
  | y :: ys -> foldl f (f acc y) ys

run
    let lst = (Left 5) :: (Right "test") :: [] in
    let swapped_lst = foldl (fun acc e -> swap e :: acc) [] lst in
    reverse swapped_lst
```

Figure 1. Example program in Millet's surface language.

recursively desugars it. The (indirectly) recursive call returns pairs containing patterns and their corresponding desugared binding computations as a set of general bindings for all nested computation terms, and the core computation that yields the final result. Then it constructs a sequence of nested *let*-bindings as AST nodes, wrapping the core computation in reverse order. Each *let*-node structurally binds a computation to its corresponding pattern using Millet's abstract syntax constructor, represented as `Let` in Algorithm 1. This results in a properly structured FGCBV-compliant representation, where each intermediate binding appears explicitly as a node in the AST. Finally, the fully nested *let*-structure is returned as the desugared computation.

Note that the subprocedure DesugarPlainComputation recursively desugars the program contained within the computation and invokes DesugarComputation on each nested computation it encounters. Therefore, its implementation details are not very relevant regarding the transformation from CBV to FGCBV, and for this reason, we omit its implementation details here.

Algorithm 1 Desugar a computation into fine-grain call-by-value form

1: **function** DesugarComputation(*state, term*)
2: $(binds,\ comp) \leftarrow$ DesugarPlainComputation(*state*, *term*)
3: $result \leftarrow comp$
4: **for all** $(pattern, binding\_comp) \in binds$ **in reverse order do**
5: $result \leftarrow$ `Let`$(binding\_comp,\ (pattern,\ result))$
6: **return** *result*

To demonstrate the work done by Algorithm 1, take for example the following surface program in Millet:

```
let id x = x
let square n = n * n


run
    id (square 5)
```

This code snippet is not in FGCBV form because `square 5` is a computation; namely, the application of the function `square` to the value `5`. In the FGCBV discipline used by Millet, function application requires the argument to be a value. Since `square 5` is a computation, it must be evaluated and its result bound before it can be passed as an argument to `id`. Note that the top-level functions `square` and `id` are already in FGCBV form and do not require transformation during desugaring. As a result, their desugaring is omitted.

Algorithm 1 achieves FGCBV form by recursively desugaring the input computation using the DesugarPlainComputation subprocedure, which identifies `square 5` as a computation and recursively calls DesugarComputation to generate a binding for it and returns both the binding and the resulting application `id v`. The outer DesugarComputation function then wraps the computation with the generated *let*-binding, producing the final FGCBV-desugared term:

```
let id x = x
let square n = n * n


run
    let v = square 5 in (id v)
```

Here, `v` is a fresh variable introduced to hold the result of evaluating the computation `square 5`. This transformation ensures that all function applications in the resulting code are applied strictly to values, in accordance with FGCBV constraints.

Additionally, Millet adopts a form of *let-polymorphism* in the style of the *Hindley-Milner* type system. In Millet, polymorphic generalization is allowed at the top level. As a result, functions defined at the top level (e.g., `let id x = x`) below are assigned fully polymorphic types such as $\forall\alpha.\,\alpha \to \alpha$, allowing them to be instantiated with multiple types in different contexts. Therefore, the following program typechecks successfully and returns (5, "str") as expected when evaluated.

```
let id x = x


run
    let a = (id 5) in (let b = id "str" in (a, b))
```

However, Millet does not permit generalization of functions defined inside computations or as values of lambda expressions. For example, the expression `let id x = x` inside a `run` block below results in a monomorphic binding for `g`, and subsequent uses of `g` at differing types produce a type error. We can illustrate this behavior with the minimal program that produces the error *Typing error: Cannot unify types int = string.*

```
run
    let id x = x in (let a = id 5 in (let b = id "str" in (a, b)))
```

This restriction is consistent with the value restriction observed in languages like Standard ML and OCaml, where only syntactic values may be generalized [21, 22]. Consequently, Millet's type system is capable of ensuring soundness in the presence of effects while retaining the expressive power of implicit polymorphism at the value level.

3.2 Core Calculus

In order to reason about the behaviour of Millet, we now describe its core calculus, $\lambda_{\mathsf{Millet}}$. As one might expect, the core calculus of a baseline language is largely standard. Nevertheless, it is relatively feature-rich. In particular, the inclusion of pairing and variant types allows it to model structured data and control flow constructs such as tuples, conditionals, and pattern matching within a minimal and compositional framework.

3.2.1 Syntax

To formalize the syntax of $\lambda_{\mathsf{Millet}}$, we begin by introducing its term structure. In Figure 2 we formalize the terms of $\lambda_{\mathsf{Millet}}$ in Backus–Naur Form (BNF). They are structured according to the FGCBV discipline [16], in which values and computations are syntactically separated. As mentioned before, this separation serves both semantic and practical purposes, ensuring that evaluation never proceeds inside values, and that operations which use an expression, such as function application or pattern matching, are performed only when that expression has already been evaluated to a value. Although functions are themselves values and may contain computations, these computations remain dormant until the function is applied, so there is no conflict with the requirement that bound expressions be values.

This syntactic stratification is particularly well-suited for languages that support algebraic effects and handlers, as it provides a clear framework for defining evaluation order, modular reasoning about effects, and a compositional typing discipline. Values include units, constants, variables, variant constructors, pairings, lambda abstractions, and recursive lambda abstractions, all of which are inert; that is, they do not reduce any further on their own. Computations, on the other hand, encompass constructs such as function application, returning values, sequential composition, and pattern matching for pairings and variants, which describe active evaluation steps.

Such fine-grain separation allows $\lambda_{\mathsf{Millet}}$ to reflect a structured evaluation model where computation steps are explicit, and forms the basis for defining its small-step operational semantics, effect handling, and temporal constructs presented later on.

Values $V, W ::= x$		variable
	$\mid ()$	unit
	$\mid f(V_1, \ldots, V_n)$	constant (primitive)
	$\mid \texttt{inl}\ V$	left injection
	$\mid \texttt{inr}\ V$	right injection
	$\mid (V, W)$	pairing
	$\mid \texttt{fun}\ x \mapsto M$	function
	$\mid \texttt{rec fun}\ f\ x \mapsto M$	recursive function

Computations $M, N ::= V\ W$		function application
	$\mid \texttt{return}\ V$	return value
	$\mid \texttt{let}\ x = M\ \texttt{in}\ N$	sequential composition
	$\mid \texttt{match}\ V\ \texttt{with}\ \{(x, y) \mapsto M\}$	pattern matching on pair
	$\mid \texttt{match}\ V\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M$	
	$\mid \texttt{inr}\ y \mapsto N\}$	pattern matching on variant

Figure 2. Values and computations of $\lambda_{\mathsf{Millet}}$.

3.2.2 Type system

In addition to terms, we also give the BNF grammar of types and contexts of $\lambda_{\mathsf{Millet}}$ in Figure 3. While they are relatively standard, we briefly review each construct for completeness. Types contain:

- Type variables α, necessary for describing abstract input and return types.
- The singleton type unit, often used where no meaningful value is returned.
- Base types b stand for primitive types such as booleans, integers and others.
- Sum type $X + Y$ represents disjoint unions or variants, enabling case analysis.
- Product type $X \times Y$ captures pairings of values.
- Function type $X \to Y$ describes a mapping from inputs of type X to outputs of type Y. The function body is considered a computation.

Well-typed values of and computations of $\lambda_{\mathsf{Millet}}$ are specified using typing judgments $\Gamma \vdash_{\mathrm{v}} V : X$ and $\Gamma \vdash_{\mathrm{c}} M : X$, respectively, where contexts Γ are sequences of variable bindings $x_1 : X_1, \ldots, x_n : X_n$, with $\emptyset$ denoting the empty context.

$$\begin{array}{ll} \textbf{Types} & X, Y, Z ::= \alpha \mid \mathrm{unit} \mid b \mid X + Y \mid X \times Y \mid X \to Y \\ \textbf{Contexts} & \Gamma ::= \emptyset \mid \Gamma, x : X \end{array}$$

Figure 3. Types and contexts of $\lambda_{\mathsf{Millet}}$.

A term is said to be well-typed if it can be assigned a type according to the typing rules of the language under a given typing context [17]. We present the typing rules of $\lambda_{\mathsf{Millet}}$ in Figure 4. Note that, as is standard in type-theoretic systems, we assume that all typing judgments are interpreted relative to a fixed global signature Σ, which provides the definitions of primitive types, constants, and operations. This signature remains constant throughout derivations and is therefore omitted from the judgment syntax. The signature Σ consists of:

- Base types, such as bool, int and others.
- Constants with type signatures of the form $f : (b_1, \ldots, b_n) \to b$, where b_i and b are base types.
- Primitive operations, which may be represented similarly to constants, such as $(+) : (\mathsf{int}, \mathsf{int}) \to \mathsf{int}$ for example.

The typing rules are largely standard for languages based on FGCBV. Nevertheless, to build intuition, we illustrate the meaning of them case by case:

- Var formalizes variable lookup in the typing context. It states that if the typing context Γ contains a binding $x : X$, then the variable x is well-typed as a value of type X under that context. In other words, the judgment $\Gamma \vdash_{\mathrm{v}} x : X$ is derivable whenever $(x : X) \in \Gamma$.
- Unit handles the trivial case of the unit value. Since unit contains only a single value $()$, it is always well-typed with type unit.

$$\textsc{Var}\ \frac{(x : X) \in \Gamma}{\Gamma \vdash_{\mathsf{v}} x : X} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash_{\mathsf{v}} () : \mathrm{unit}} \qquad \textsc{Const}\ \frac{(\Gamma \vdash_{\mathsf{v}} V_i : b_i)_{1 \le i \le n}}{\Gamma \vdash_{\mathsf{v}} f(V_1, \ldots, V_n) : b} \qquad \textsc{Inl}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X}{\Gamma \vdash_{\mathsf{v}} \mathtt{inl}\ V : X + Y}$$

$$\textsc{Inr}\ \frac{\Gamma \vdash_{\mathsf{v}} V : Y}{\Gamma \vdash_{\mathsf{v}} \mathtt{inr}\ V : X + Y} \qquad \textsc{Pair}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X \qquad \Gamma \vdash_{\mathsf{v}} W : Y}{\Gamma \vdash_{\mathsf{v}} (V, W) : X \times Y} \qquad \textsc{Fun}\ \frac{\Gamma, x : X \vdash_{\mathsf{c}} M : Y}{\Gamma \vdash_{\mathsf{v}} \mathtt{fun}\ x \mapsto M : X \to Y}$$

$$\textsc{RecFun}\ \frac{\Gamma, f : X \to Y, x : X \vdash_{\mathsf{c}} M : Y}{\Gamma \vdash_{\mathsf{v}} \mathtt{rec\ fun}\ f\ x \mapsto M : X \to Y} \qquad \textsc{Apply}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X \to Y \qquad \Gamma \vdash_{\mathsf{v}} W : X}{\Gamma \vdash_{\mathsf{c}} V\ W : Y}$$

$$\textsc{Return}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X}{\Gamma \vdash_{\mathsf{c}} \mathtt{return}\ V : X} \qquad \textsc{Let}\ \frac{\Gamma \vdash_{\mathsf{c}} M : X \qquad \Gamma, x : X \vdash_{\mathsf{c}} N : Y}{\Gamma \vdash_{\mathsf{c}} \mathtt{let}\ x = M\ \mathtt{in}\ N : Y}$$

$$\textsc{MatchPair}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X \times Y \qquad \Gamma, x : X, y : Y \vdash_{\mathsf{c}} M : Z}{\Gamma \vdash_{\mathsf{c}} \mathtt{match}\ V\ \mathtt{with}\ \{(x, y) \mapsto M\} : Z}$$

$$\textsc{MatchSum}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X + Y \qquad \Gamma, x : X \vdash_{\mathsf{c}} M : Z \qquad \Gamma, y : Y \vdash_{\mathsf{c}} N : Z}{\Gamma \vdash_{\mathsf{c}} \mathtt{match}\ V\ \mathtt{with}\ \{\mathtt{inl}\ x \mapsto M \mid \mathtt{inr}\ y \mapsto N\} : Z}$$

Figure 4. Typing rules of $\lambda_{\mathsf{Millet}}$.

- Const defines the typing of primitive constants. If each argument V_i to a constant function f is well-typed with the appropriate base type b_i, specified by the signature Σ, then the application $f(V_1, \ldots, V_n)$ is well-typed with the return base type b.
- Inl and Inr inject values into sum types. If V has type X, then $\mathtt{inl}\ V$ is well-typed with type $X + Y$ for any type Y. Similarly, if V has type Y, then $\mathtt{inr}\ V$ is well-typed with type $X + Y$ for any type X.
- Pair states that if two values V and W are well-typed with types X and Y respectively, then the pair (V, W) is well-typed with the product type $X \times Y$.
- Fun defines the typing of function abstractions. If the computation M is well-typed with type Y under an extended context that contains the binding $x : X$, then the function $\mathtt{fun}\ x \mapsto M$ is well-typed as a value of function type $X \to Y$.

- RecFun defines recursive functions. It extends the context with both bindings for the function name $f : X \to Y$ and its argument $x : X$. If the body M is well-typed with type Y, then the recursive function `rec fun` $f\ x \mapsto M$ has type $X \to Y$.
- Apply defines function application. If V is a function of type $X \to Y$ and W is a value of type X, then the application $V\ W$ is a computation of type Y.
- Return lifts a value into the computation layer, trivially returning it without performing any effects. If V is a well-typed value of type X, then `return` V is a well-typed computation of type X.
- Let performs sequential composition in computations. If M is a computation of type X and N is a computation of type Y under a context extended with the binding $x : X$, then the expression `let` $x = M$ `in` N is a computation of type Y.
- MatchPair performs pattern matching on a pair. If V is a value of type $X \times Y$ and N is a computation of type Z under a context extended with the bindings $x : X$ and $y : Y$, then the match expression is well-typed as a computation of type Z.
- MatchSum enables case analysis on sum types. If V has type $X + Y$, and both branches M and N are well-typed computations of type Z under contexts extended with the bindings $x : X$ and $y : Y$, respectively, then the full match expression is well-typed as a computation of type Z.

3.3 Semantics

In addition to typing rules, we formalize the small-step operational semantics for $\lambda_{\mathsf{Millet}}$ given by a relation $M \rightsquigarrow M'$ defined in Figure 5, specifying how computations evaluate one step at a time using reduction rules. Reduction applies only to computations, as values are considered terminal forms and do not reduce further.

We write $W[V/x]$ and $M[V/x]$ for the *capture-avoiding substitution* of value V for variable x in a value W or computation M, respectively. Following standard practice [17], this operation replaces all free occurrences of x while avoiding variable capture through renaming. To simplify reasoning, we adopt *Barendregt's convention* [12], which assumes that bound variables are always chosen distinct from free variables in any given context. As a result, we omit explicit variable renaming.

The reduction rules follow a standard call-by-value evaluation strategy: *let*-bindings reduce computations to returning values before creating the binding, function application substitutes a value into the function body, and pattern matching on pairs or sum types destructs fully evaluated values before continuing. This step-by-step formulation defines a formal operational semantics where congruence in *let*-bindings drives computation, implicitly performing function application and pattern matching only once their arguments are fully evaluated. This design enforces a strict separation between values and computations, in line with the FGCBV structure of $\lambda_{\mathsf{Millet}}$.

$$\textsc{Sem-App}\quad \frac{}{(\texttt{fun}\ x \mapsto M)\ V \rightsquigarrow M[V/x]}$$

$$\textsc{Sem-App-Rec}\quad \frac{}{(\texttt{rec fun}\ f\ x \mapsto M)\ V \rightsquigarrow M[V/x][(\texttt{rec fun}\ f\ x \mapsto M)/f]}$$

$$\textsc{Sem-Let-Return}\quad \frac{}{\texttt{let}\ x = (\texttt{return}\ V)\ \texttt{in}\ N \rightsquigarrow N[V/x]} \qquad \textsc{Sem-Let-Cong}\quad \frac{M \rightsquigarrow M'}{\texttt{let}\ x = M\ \texttt{in}\ N \rightsquigarrow \texttt{let}\ x = M'\ \texttt{in}\ N}$$

$$\textsc{Sem-MatchPair}\quad \frac{}{\texttt{match}\ (V, W)\ \texttt{with}\ \{(x, y) \mapsto M\} \rightsquigarrow M[V/x, W/y]}$$

$$\textsc{Sem-MatchInl}\quad \frac{}{\texttt{match}\ (\texttt{inl}\ V)\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\} \rightsquigarrow M[V/x]}$$

$$\textsc{Sem-MatchInr}\quad \frac{}{\texttt{match}\ (\texttt{inr}\ V)\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\} \rightsquigarrow N[V/y]}$$

Figure 5. Small-step operational semantics of $\lambda_{\mathsf{Millet}}$.

3.4 Type Inference

As noted in Section 3.1, Millet implements type inference and unification in the style of the *Hindley-Milner* type system. In this section, we formalize the inference rules used in $\lambda_{\mathsf{Millet}}$

and the unification of the constraints produced during type inference. The presentation closely follows the style of Matija Pretnar's paper on inferring algebraic effects [23].

3.4.1 Inference Rules

The declarative type system of $\lambda_{\mathsf{Millet}}$, presented in Section 3.2.2, already supports type variables α, allowing us to deal with unannotated terms and support polymorphism.

We infer types and constraints using syntax-directed type inference judgments for values and computations of the form:

$$\Gamma; \Xi \vDash_{F} V \Rightarrow X \mid C \quad \text{and} \quad \Gamma; \Xi \vDash_{F} M \Rightarrow X \mid C.$$

Here, C is a set of constraints between types of the form $X \doteq Y$, and F is the set of fresh type variables introduced during type inference. Ξ denotes the context of polymorphic assumptions, which are introduced by top-level polymorphic let expressions, as seen in Section 3.1. The polymorphic assumptions in Ξ assign type schemes to variables. A type scheme is an expression of the form $\forall F.X$, where F is a possibly empty set of type variables and X is a type. If Ξ is empty, we omit it for brevity.

The type schemes used in $\lambda_{\mathsf{Millet}}$ are thus structurally similar to the polymorphic types of ML. As in [23], Ξ is distinct from the non-polymorphic context Γ. Although Γ could be seen as a restricted instance of Ξ, we maintain this separation in order to relate inference judgments to the typing judgments defined earlier, which refer only to Γ. Additionally, note that the type schemes in Ξ are not unaffected by unresolved constraints, similarly to polymorphic let-bindings in general Hindley-Milner style type inference.

We adopt the same assumptions regarding parameter sets as in Pretnar's work [23]. Although F is formally a set, we treat it as a sequence of distinct elements to emphasize that no parameter is repeated. For instance, a judgment of the form F_1, F_2, α implicitly assumes that F_1 and F_2 are disjoint and that neither contains α. Following the same convention, we also assume that any bound parameters in F are implicitly renamed (i.e., freshly instantiated) each time polymorphic variable usage occurs. These assumptions simplify reasoning about polymorphic variable usage and are consistent with the approach taken in [23].

Additionally, similarly to the declarative typing judgments, we assume that all inference judgments are interpreted relative to a fixed global signature Σ, which provides the

definitions of primitive types, constants, and operations. This signature remains constant throughout derivations and is therefore omitted from the judgment syntax.

The type inference rules are presented in Figure 6.

$$\textsc{Infer-Unit}\ \frac{}{\Gamma;\Xi \vDash_{\emptyset} () \Rightarrow \text{unit} \mid \emptyset}$$

$$\textsc{Infer-Const}\ \frac{\Gamma;\Xi \vDash_{\emptyset} V_i \Rightarrow b_i \mid \emptyset \quad \text{for all } i}{\Gamma;\Xi \vDash_{\emptyset} f(V_1,\ldots,V_n) \Rightarrow b \mid \emptyset}$$

$$\textsc{Infer-Var}\ \frac{(x : X) \in \Gamma}{\Gamma;\Xi \vDash_{\emptyset} x \Rightarrow X \mid \emptyset}$$

$$\textsc{Infer-PolyVar}\ \frac{(x : \forall F.X) \in \Xi}{\Gamma;\Xi \vDash_{F} x \Rightarrow X \mid \emptyset}$$

$$\textsc{Infer-Inl}\ \frac{\Gamma;\Xi \vDash_{F} V \Rightarrow X \mid C}{\Gamma;\Xi \vDash_{F,\alpha} \texttt{inl}\ V \Rightarrow X + \alpha \mid C}$$

$$\textsc{Infer-Inr}\ \frac{\Gamma;\Xi \vDash_{F} V \Rightarrow Y \mid C}{\Gamma;\Xi \vDash_{F,\alpha} \texttt{inr}\ V \Rightarrow \alpha + Y \mid C}$$

$$\textsc{Infer-Pair}\ \frac{\Gamma;\Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma;\Xi \vDash_{F_2} W \Rightarrow Y \mid C_2}{\Gamma;\Xi \vDash_{F_1,F_2} (V,W) \Rightarrow X \times Y \mid C_1 \cup C_2}$$

$$\textsc{Infer-Fun}\ \frac{\Gamma, x : \alpha;\Xi \vDash_{F} M \Rightarrow X \mid C}{\Gamma;\Xi \vDash_{F,\alpha} \texttt{fun}\ x \mapsto M \Rightarrow \alpha \to X \mid C}$$

$$\textsc{Infer-RecFun}\ \frac{\Gamma, f : \alpha_1, x : \alpha_2;\Xi \vDash_{F} M \Rightarrow X \mid C}{\Gamma;\Xi \vDash_{F,\alpha_1,\alpha_2} \texttt{rec fun}\ f\ x \mapsto M \Rightarrow \alpha_1 \mid C \cup \{\alpha_1 \doteq \alpha_2 \to X\}}$$

$$\textsc{Infer-App}\ \frac{\Gamma;\Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma;\Xi \vDash_{F_2} W \Rightarrow Y \mid C_2}{\Gamma;\Xi \vDash_{F_1,F_2,\alpha} V\ W \Rightarrow \alpha \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\}}$$

$$\textsc{Infer-Return}\ \frac{\Gamma;\Xi \vDash_{F} V \Rightarrow X \mid C}{\Gamma;\Xi \vDash_{F} \texttt{return}\ V \Rightarrow X \mid C}$$

$$\textsc{Infer-Let}\ \frac{\Gamma;\Xi \vDash_{F_1} M \Rightarrow X \mid C_1 \qquad \Gamma, x : \alpha;\Xi \vDash_{F_2} N \Rightarrow Y \mid C_2}{\Gamma;\Xi \vDash_{F_1,F_2,\alpha} \texttt{let}\ x = M\ \texttt{in}\ N \Rightarrow Y \mid C_1 \cup C_2 \cup \{X \doteq \alpha\}}$$

$$\textsc{Infer-MatchPair}\ \frac{\Gamma;\Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma, x : \alpha_1, y : \alpha_2;\Xi \vDash_{F_2} M \Rightarrow Y \mid C_2}{\Gamma;\Xi \vDash_{F_1,F_2,\alpha_1,\alpha_2} \texttt{match}\ V\ \texttt{with}\ \{(x,y) \mapsto M\} \Rightarrow Y \mid C_1 \cup C_2 \cup \{X \doteq \alpha_1 \times \alpha_2\}}$$

$$\textsc{Infer-MatchSum}\ \frac{\Gamma;\Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma, x : \alpha_1;\Xi \vDash_{F_2} M \Rightarrow Y \mid C_2 \qquad \Gamma, y : \alpha_2;\Xi \vDash_{F_3} N \Rightarrow Z \mid C_3}{\begin{array}{c}\Gamma;\Xi \vDash_{F_1,F_2,F_3,\alpha_1,\alpha_2,\alpha_3} \texttt{match}\ V\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\} \\ \Rightarrow \alpha_3 \mid C_1 \cup C_2 \cup C_3 \cup \{(X \doteq \alpha_1 + \alpha_2), (Y \doteq \alpha_3), (Z \doteq \alpha_3)\}\end{array}}$$

Figure 6. Inference rules for type and constraint generation in $\lambda_{\mathsf{Millet}}$.

We review each rule presented in Figure 6 case-by-case:

- Infer-Unit defines type inference for the unit value (). It requires no premises. In any typing context $\Gamma; \Xi$, we infer the type `unit` for the term (). No fresh variables are introduced, and no constraints are generated in this case.

- Infer-Const defines type inference for function constant applications. If each argument V_i to a function constant f, can be inferred a corresponding base type b_i, specified by the signature Σ, then the we infer the base type b for the application $f(V_1, \ldots, V_n)$. No fresh variables are introduced, and no constraints are generated in this case.

- Infer-Var defines type inference for a monomorphic variable x from the non-polymorphic context Γ. If $(x : X) \in \Gamma$, then the type X is inferred for x. No fresh variables are introduced, and no constraints are generated in this case.

- Infer-PolyVar defines type inference for a polymorphic variable x from the polymorphic context Ξ. If $(x : \forall F.X) \in \Xi$, then the type X is inferred for x and all type variables in F, bound by the quantifier, are considered fresh for this usage. No constraints are generated in this case. Recall that we assume that any bound parameters in F are implicitly renamed each time this rule is applied.

- Infer-Inl defines type inference for the left injection into a sum type. If the term V can be inferred to have type X under contexts $\Gamma; \Xi$, then the term `inl` V is inferred to have type $X + \alpha$, where α is a fresh type variable representing the right-hand summand. The set of fresh variables includes those obtained from inferring the type of V together with α. The set of constraints is the same as that inferred for V.

- Infer-Inr defines type inference for the right injection into a sum type. If the term V can be inferred to have type Y under contexts $\Gamma; \Xi$, then the term `inr` V is inferred to have type $\alpha + Y$, where α is a fresh type variable representing the left-hand summand. The set of fresh variables includes those obtained from inferring the type of V together with α. The set of constraints is the same as that inferred for V.

- Infer-Pair defines type inference for product types. If V can be inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, and W can be inferred to have type Y, introducing the set of fresh variables F_2 and the

set of constraints C_2, then the pair (V, W) is inferred to have type $X \times Y$. The set of fresh variables is the combined set F_1, F_2, obtained from inferring the types of V and W, respectively, and the set of constraints is $C_1 \cup C_2$, also obtained from inferring the types of V and W, respectively.

- Infer-Fun defines type inference for non-recursive functions. If the function body M can be inferred to have type X under the contexts $\Gamma, x : \alpha; \Xi$, where α is a fresh type variable representing the argument type of the function, then the term `fun` $x \mapsto M$ is inferred to have type $\alpha \to X$. The set of fresh variables consists of those obtained from inferring the type of M together with α, and the set of constraints remains equal to the one obtained from inferring the type of M.

- Infer-RecFun defines type inference for recursive functions. If the function body M can be inferred to have type X under the contexts $\Gamma, f : \alpha_1, x : \alpha_2; \Xi$, where α_1 and α_2 are fresh type variables representing the function's type and the argument type respectively, then the term `rec fun` $f\ x \mapsto M$ is inferred to have type α_1. The set of fresh variables consists of those obtained from inferring the type of M together with α_1 and α_2. The set of constraints is the union of the constraints obtained from inferring the type of M and the constraint $\alpha_1 \doteq \alpha_2 \to X$, ensuring that the function type matches its argument and return types.

- Infer-App defines type inference for function application. If V can be inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, and W can be inferred to have type Y, introducing the set of fresh variables F_2 and the set of constraints C_2, then the application $V\ W$ is inferred to have type α, where α is a fresh type variable representing the result type of the application. The set of fresh variables is F_1, F_2, α, and the set of constraints is $C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\}$, enforcing that the function type matches its argument type and result type.

- Infer-Return defines type inference for returning a value from a computation. If V can be inferred to have type X under contexts $\Gamma; \Xi$, introducing the set of fresh variables F and the set of constraints C, then the term `return` V is also inferred to have type X. The set of fresh variables and the set of constraints remain the same as those obtained from inferring V.

- Infer-Let defines type inference for let-bindings. If M can be inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, and N can

be inferred to have type Y under contexts extended with $x : \alpha$, introducing the set of fresh variables F_2 and the set of constraints C_2, then the term $\texttt{let}\ x = M\ \texttt{in}\ N$ is inferred to have type Y. The set of fresh variables is F_1, F_2, α, and the set of constraints is $C_1 \cup C_2 \cup \{X \doteq \alpha\}$, ensuring that the bound variable x has the same type as the value computed by M.

- Infer-MatchPair defines type inference for pattern matching on a pair. If V can be inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, and M can be inferred to have type Y under contexts extended with $x : \alpha_1$ and $y : \alpha_2$, introducing the set of fresh variables F_2 and the set of constraints C_2, then the term $\texttt{match}\ V\ \texttt{with}\ \{(x, y) \mapsto M\}$ is inferred to have type Y. The set of fresh variables is $F_1, F_2, \alpha_1, \alpha_2$, and the set of constraints is $C_1 \cup C_2 \cup \{X \doteq \alpha_1 \times \alpha_2\}$, ensuring that V has a product type matching the two bound variables.

- Infer-MatchSum defines type inference for pattern matching on a sum type. If V can be inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, M can be inferred to have type Y under contexts extended with $x : \alpha_1$, introducing the set of fresh variables F_2 and the set of constraints C_2, and N can be inferred to have type Z under contexts extended with $y : \alpha_2$, introducing the set of fresh variables F_3 and the set of constraints C_3, then the term $\texttt{match}\ V\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\}$ is inferred to have type α_3, where α_3 is a fresh type variable representing the result type of the match. The set of fresh variables is $F_1, F_2, F_3, \alpha_1, \alpha_2, \alpha_3$, and the set of constraints is $C_1 \cup C_2 \cup C_3 \cup \{X \doteq \alpha_1 + \alpha_2,\ Y \doteq \alpha_3,\ Z \doteq \alpha_3\}$, ensuring that V has a sum type matching the two bound variables and that both branches yield the same result type.

Note that the inference rules do not directly compute *substitutions*. Instead, they generate a type, a set of fresh variables, and a set of constraints. These constraints are solved later by applying the unification algorithm presented in the next subsection. This two-phase structure — constraint generation followed by unification — follows the *Hindley–Milner* tradition, keeping the inference rules simple while isolating the complexity of solving constraints to a dedicated step.

3.4.2 Unification

This subsection focuses on constraint unification in $\lambda_{\mathsf{Millet}}$. We begin by defining supporting structures, followed by a formalization of the unification algorithm. We then review the properties of the algorithm and its output, and finally, we give a concrete example of one case of unification.

Definition 3.1 (Substitution)**.** *A* substitution of type variables *is a finite mapping σ from type variables to types. We write* $\mathrm{dom}(\sigma)$ *for the domain of σ, i.e., the set of type variables that σ replaces. If α is a type variable and X is a type, then $\sigma = [\alpha \mapsto X]$ denotes the singleton substitution that replaces α with X.*

Substitutions can be applied to types and composed.

Definition 3.2 (Substitution Application to Types)**.** *Let σ be a substitution of type variables. The application of a substitution σ to a type X, written $\sigma(X)$, is defined recursively as follows:*

$$\sigma(\alpha) = \begin{cases} X & \text{if } \alpha \mapsto X \in \sigma \\ \alpha & \text{otherwise} \end{cases}$$

$$\sigma(b) = b, \text{where } b \text{ is a base type (e.g., } \mathsf{int}, \mathsf{bool})$$

$$\sigma(\mathit{unit}) = \mathit{unit}$$

$$\sigma(X \to Y) = \sigma(X) \to \sigma(Y)$$

$$\sigma(X + Y) = \sigma(X) + \sigma(Y)$$

$$\sigma(X \times Y) = \sigma(X) \times \sigma(Y)$$

Definition 3.3 (Substitution Composition)**.** *Let σ_1 and σ_2 be substitutions. Their composition $\sigma_1 \circ \sigma_2$ is the substitution defined by:*

$$(\sigma_1 \circ \sigma_2)(\alpha) = \sigma_1(\sigma_2(\alpha))$$

for all type variables α. That is, we first apply σ_2, then apply σ_1 to the result.

We now provide a formal description of the unification algorithm of $\lambda_{\mathsf{Millet}}$ that takes a constraint set C as input and returns a substitution σ if the constraint set is unifiable.

Algorithm 2 Recursive unification algorithm of $\lambda_{\mathsf{Millet}}$

```
1:  function UNIFY(C)
2:      if C = ∅ then
3:          return ∅
4:      else
5:          Let (X ≐ Y) ∈ C, and let C' = C \ {(X ≐ Y)}
6:          if X = Y then
7:              return UNIFY(C')
8:          else if X = X1 → X2 and Y = Y1 → Y2 then
9:              return UNIFY({(X1 ≐ Y1), (X2 ≐ Y2)} ∪ C')
10:         else if X = X1 × X2 and Y = Y1 × Y2 then
11:             return UNIFY({(X1 ≐ Y1), (X2 ≐ Y2)} ∪ C')
12:         else if X = X1 + X2 and Y = Y1 + Y2 then
13:             return UNIFY({(X1 ≐ Y1), (X2 ≐ Y2)} ∪ C')
14:         else if X = α and α ∉ fv(Y) then
15:             Let C'' = C' with α replaced by Y in all types
16:             Let σ' = UNIFY(C'')
17:             return σ' ∪ {α ↦ σ'(Y)}
18:         else if Y = α and α ∉ fv(X) then
19:             Let C'' = C' with α replaced by X in all types
20:             Let σ' = UNIFY(C'')
21:             return σ' ∪ {α ↦ σ'(X)}
22:         else
23:             fail("Cannot unify  X  and  Y")
```

It is an adaptation of Robinson's unification algorithm [17], extended to account for product and sum types. These extensions are handled through structural decomposition, in a manner analogous to the treatment of function types.

In contrast to the original formulation of Robinson's unification algorithm, which constructs the result by composing substitutions at the end, this version incrementally extends a substitution map and eagerly applies substitutions to the remaining constraint set. This avoids the need to maintain a deferred composition and ensures that all types in constraints remain normalized after each step. These modifications preserve the

fundamental properties of Robinson's unification algorithm. Therefore, Algorithm 2 is guaranteed to terminate.

We now define *constraint satisfaction*, *unifiers*, *principal unifiers*, and *principal types* for values and computations.

Definition 3.4 (Constraint Satisfaction)**.** *Let C be a set of type equality constraints of the form $X \doteq Y$. A substitution σ* satisfies *C, written $\sigma \vDash C$, if for every constraint $(X \doteq Y) \in C$, we have:*

$$\sigma(X) = \sigma(Y)$$

where equality is understood as syntactic equality of types after applying σ.

Definition 3.5 (Unifier)**.** *Let C be a constraint set. A substitution σ is called a* unifier *for C if $\sigma \vDash C$.*

Definition 3.6 (Principal Unifier)**.** *Let C be a constraint set. A substitution σ is called a* principal unifier *for C if:*

- *σ is a unifier for C, and*
- *for every substitution σ' such that σ' is also a unifier for C, there exists a substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$.*

In other words, σ is the most general solution to C; any other unifier is an instance of it.

Algorithm 2 produces a *principal unifier* when the input constraint set is unifiable, and fails otherwise. The principal unifier can then be applied to the type generated during inference to obtain a *principal type*.

Definition 3.7 (Principal Type of a Term)**.** *Let V be a value in the language. A type X is said to be a* principal type *of V if:*

- *V has type X in the declarative type system, and*
- *for every type Y such that V also has type Y, there exists a substitution σ such that $Y = \sigma(X)$.*

That is, X is the most general type assignable to V. All other valid types are its instances. Analogously for computations M.

In both the value and computation fragments of the language, principal types are computed in two phases:

1. Generate a type X and a set of constraints C for the value or computation using the corresponding inference rules.
2. Compute a principal unifier σ of the constraints C.

The resulting principal type is $\sigma(X)$. We now work through a concrete example of type inference and unification producing a principal type.

Example 3.1 (Inference and unification in action)**.** *Consider the application of a left injection function to an integer:*

$$(\texttt{fun}\ x \mapsto \texttt{inl}\ x)\ 42$$

Let α_1, α_2, and α_3 be fresh type variables.

- *For* $\texttt{fun}\ x \mapsto \texttt{inl}\ x$*, we apply* Infer-Inl *and* Infer-Fun*, respectively:*

$$\texttt{inl}\ x \Rightarrow \alpha_1 + \alpha_2$$

$$\texttt{fun}\ x \mapsto \texttt{inl}\ x \Rightarrow \alpha_1 \to \alpha_1 + \alpha_2$$

- *For the constant* 42*, we apply* Infer-Const*:*

$$42 \Rightarrow \mathsf{int}$$

- *For the application* $(\texttt{fun}\ x \mapsto \texttt{inl}\ x)\ 42$*, we apply* Infer-App*:*

$$\textit{generate result type } \alpha_3,\ \textit{produce constraint } \{\alpha_1 \to \alpha_1 + \alpha_2 \doteq \mathsf{int} \to \alpha_3\}$$

Unifying $\alpha_1 \to \alpha_1 + \alpha_2 \doteq \mathsf{int} \to \alpha_3$ *yields the following constraints:*

$$\begin{aligned} &(1) \quad \alpha_1 \doteq \mathsf{int} \\ &(2) \quad \alpha_1 + \alpha_2 \doteq \alpha_3. \end{aligned}$$

Unifying $\alpha_1 \doteq \mathsf{int}$*, we find that* α_1 *does not occur in* int*, so we substitute* $\alpha_1 \mapsto \mathsf{int}$ *and return:*

$$\sigma = \{\alpha_1 \mapsto \mathsf{int}\}.$$

Unifying $\mathsf{int} + \alpha_2 \doteq \alpha_3$*, we find that* α_3 *does not occur in* $\mathsf{int} + \alpha_2$*, so we substitute* $\alpha_3 \mapsto \mathsf{int} + \alpha_2$ *and return:*

$$\sigma = \{\alpha_1 \mapsto \mathsf{int}, \alpha_3 \mapsto \mathsf{int} + \alpha_2\},$$

which is the final substitution. Therefore, the principal type of the expression (`fun` $x \mapsto$ `inl` x) 42 *is:*

$$\mathsf{int} + \alpha_2$$

This reflects the fact that the injection function returns a left-tagged sum where the right type remains unconstrained.

3.5 Soundness of Type Inference

In this section, we present and prove an auxiliary Theorem 3.1 and Lemma 3.2 with the purpose of showing that polymorphic variable usage can be removed from an inference tree without altering its structure. Note that from here on out, we refer to the process of removing polymorphic variable usage from a type inference tree as *linearization*. Linearization is necessary for interpreting type inference results in the declarative type system. Linearization results are then supplemented with Theorem 3.3 and its proof, stating that $\lambda_{\mathsf{Millet}}$'s type inference is sound under the condition that polymorphic variables are not used. Finally, we conclude the chapter by stating the general soundness Theorem 3.4 and proving it using the aforementioned auxiliary results.

Theorem 3.1 (Polymorphic variable usage is linearizable)**.** *For all inference trees for the judgment* $\Gamma; \Xi \vDash_{\mathsf{v}\ F} V \Rightarrow X \mid C$*, there exist* Γ', F', V' *such that* $\Gamma', \Gamma \vDash_{\mathsf{v}\ F'} V' \Rightarrow X \mid C$*, where* Γ' *denotes a context derived from* Ξ*. Analogously, for all inference trees for the judgment* $\Gamma; \Xi \vDash_{\mathsf{c}\ F} M \Rightarrow X \mid C$*, there exist* Γ', F', M' *such that* $\Gamma', \Gamma \vDash_{\mathsf{c}\ F'} M' \Rightarrow X \mid C$*, where* Γ' *denotes a context derived from* Ξ*.*

Proof. We define two mutually recursive functions

$$\begin{aligned} \textsc{vlinearize} : \quad & (\Gamma; \Xi \vDash_{\mathsf{v}\ F} V \Rightarrow X \mid C) \times \mathbb{N} \to \\ & (\Gamma' : \mathsf{Ctx}) \times (F' : \mathcal{P}(\mathsf{Vars})) \times (V' : \mathsf{Val}) \times (\Gamma', \Gamma \vDash_{\mathsf{v}\ F'} V' \Rightarrow X \mid C) \times \mathbb{N} \end{aligned}$$

$$\begin{aligned} \textsc{clinearize} : \quad & (\Gamma; \Xi \vDash_{\mathsf{c}\ F} M \Rightarrow X \mid C) \times \mathbb{N} \to \\ & (\Gamma' : \mathsf{Ctx}) \times (F' : \mathcal{P}(\mathsf{Vars})) \times (M' : \mathsf{Comp}) \times (\Gamma', \Gamma \vDash_{\mathsf{c}\ F'} M' \Rightarrow X \mid C) \times \mathbb{N} \end{aligned}$$

for linearizing inference judgments on value types and computation types, respectively. Both functions perform a depth-first traversal of inference trees, replacing each use of a polymorphic variable from Ξ with a uniquely renamed copy. After transformation, Ξ is eliminated, and its contributions are integrated into the context Γ'. This way, all freshly

introduced variables derived from the initial polymorphic context Ξ are guaranteed to be used at most once, and thus polymorphic variable usage in the final inference tree is *linearized*.

Note that linearization preserves both the constraint set and the type of a term, as it leaves type variables unchanged. The transformation only renames concrete polymorphic variables, introducing fresh copies indexed by i. This is justified by the assumption that any bound parameters in F are implicitly renamed with each use of a polymorphic variable.

We proceed with an overview of the linearization functions. Let the auxiliary function $\textsc{fresh}(i, x : \forall F.X)$ generate a fresh polymorphic variable x_i with type X using the index i. The functions proceed as follows:

- For leaf rules or rules that do not involve polymorphic variables (e.g., Infer-Unit, Infer-Const), return them unchanged.
- In the most interesting case — the Infer-PolyVar rule,

$$\text{Let } (x_i : X) = \textsc{fresh}(i, x : \forall F.X),$$

$$\text{Let } \Gamma' = x_i : X,$$

then

$$\textsc{vlinearize}\left(\left(\frac{(x : \forall F.X) \in \Xi}{\Gamma; \Xi \vDash_{\forall}\ _F\ x \Rightarrow X \mid \emptyset}\ \textsc{Infer-PolyVar}\right), i\right) =$$

$$\left(\Gamma', \emptyset, x_i, \left(\frac{(x_i : X) \in \Gamma', \Gamma}{\Gamma', \Gamma \vDash_{\forall}\ _{\emptyset}\ x_i \Rightarrow X \mid \emptyset}\ \textsc{Infer-Var}\right), i + 1\right).$$

Observe that the set of fresh type variables F is returned as empty in Infer-Var. Consequently, the set of fresh type variables F' obtained after linearization no longer contains the free variables introduced in the initial Infer-PolyVar cases. Therefore, applying the polymorphic top-level Let rule to the resulting linearized type inference tree, where all variables in the term's F-index are universally quantified into the type scheme context Ξ, would yield a result differing from that obtained by applying the rule to the original type inference tree, as the variables introduced in the Infer-PolyVar cases are no longer present. This discrepancy is not relevant for the results

presented in this thesis, because $\lambda_{\mathsf{Mille}[\tau]}$ does not currently include polymorphic top-level Let rules. Nevertheless, the F-index produced by linearization continues to serve as a valid basis for ensuring the freshness of variables introduced during type inference.

- For rules with multiple premises (e.g., Infer-App, Infer-Pair), we recursively traverse the first subderivation to obtain an updated derivation and index, then apply the updated index to the second subderivation. For example:

  Let d_1 be an inference tree for $\Gamma; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1$.
  Let d_2 be an inference tree for $\Gamma; \Xi \vDash_{F_2} W \Rightarrow Y \mid C_2$.
  Let $(\Gamma_1', F_1', V', d_1', i_1) = \textsc{vlinearize}(d_1, i)$.
  Let $(\Gamma_2', F_2', W', d_2', i_2) = \textsc{vlinearize}(d_2, i_1)$.
  Let $\Gamma' = \Gamma_1', \Gamma_2'$.
  Let $F' = F_1', F_2', \alpha$.

  $$\text{Then } \textsc{clinearize}\left(\left(\dfrac{\dfrac{d_1 \qquad d_2}{\Gamma; \Xi \vDash_{F_1, F_2, \alpha} V\ W \Rightarrow \alpha \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\}}\ \textsc{Infer-App}}{}\right), i\right) =$$

  $$\left(\Gamma', F', V'\ W', \left(\dfrac{\dfrac{d_1' \qquad d_2'}{\Gamma', \Gamma \vDash_{F'} V'\ W' \Rightarrow \alpha \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\}}\ \textsc{Infer-App}}{}\right), i_2\right).$$

- For rules that involve variable bindings, (e.g Infer-Fun, Infer-Let), we notice that freshly introduced bound variables are not affected by linearization, as they are always non-polymorphic and therefore not affected by Ξ. For example:

  Let $\Gamma', \Gamma, x : \alpha \vDash_{F'} M' \Rightarrow X \mid C$ be the root of the linearized inference tree corresponding to $\Gamma, x : \alpha; \Xi \vDash_{F} M \Rightarrow X \mid C$, and i' the updated index that was returned from the linearization call of that tree, which was passed i.

  $$\text{Then } \textsc{vlinearize}\left(\left(\dfrac{\dfrac{\Gamma, x : \alpha; \Xi \vDash_{F} M \Rightarrow X \mid C}{\Gamma; \Xi \vDash_{F, \alpha} \texttt{fun}\ x \mapsto M \Rightarrow \alpha \to X \mid C}\ \textsc{Infer-Fun}}{}\right), i\right) =$$

  $$\left(\Gamma', (F', \alpha), \texttt{fun}\ x \mapsto M', \left(\dfrac{\dfrac{\Gamma', \Gamma, x : \alpha \vDash_{F'} M' \Rightarrow X \mid C}{\Gamma', \Gamma \vDash_{F', \alpha} \texttt{fun}\ x \mapsto M' \Rightarrow \alpha \to X \mid C}\ \textsc{Infer-Fun}}{}\right), i'\right).$$

We omit the rest of the cases, as they are structurally identical to either Infer-App or Infer-Fun or are trivial extensions of one, e.g. Infer-MatchSum being analogous to Infer-App, but having 3 premises instead of 2. It follows that the linearization of polymorphic variable usage in the inference trees of $\lambda_{\mathsf{Millet}}$ can be achieved by applying the functions vlinearize and clinearize to relevant nodes. Therefore, polymorphic variable usage is linearizable. □

We can observe that the structure of an inference tree remains identical to its linearized form, modulo renaming.

Lemma 3.2 (Linearization of polymorphic variable usage preserves structure)**.** *A value V differs from its linearized form V' only in variable names. Their syntactic structure remains identical. Analogously, a computation M differs from its linearized form M' only in variable names. Their syntactic structure remains identical.*

Proof. This follows directly from the definition of the auxiliary function fresh used in the proof of Theorem 3.1, which only renames variables without modifying the syntactic form of values. □

Notice that in the function cases, we invoke both substitution application on types, defined in Definition 3.2, and substitution application on computations, written σ_C, given by the following definition.

The soundness proof requires applying substitutions to contexts.

Definition 3.8 (Substitution application to Contexts)**.** *Let σ be a substitution of type variables. The application of σ to a context $\Gamma = \{x_1 : X_1, \ldots, x_n : X_n\}$, written σ_Γ, is defined as $\sigma_\Gamma(\Gamma) = \{x_1 : \sigma(X_1), \ldots, x_n : \sigma(X_n)\}$.*

Note that we assume that applying an empty substitution to types, values, computations, and contexts does not modify them. With these definitions in mind, we present the soundness results.

Theorem 3.3 (Soundness of $\lambda_{\mathsf{Millet}}$ type inference without polymorphism)**.** *Let $\Gamma; \Xi \vdash_F V \Rightarrow X \mid C$ be an inference derivation where $\Xi = \emptyset$. Suppose that $\sigma = \textsc{unify}(C)$ is a principal unifier for the constraint set C. Then it follows that:*

$$\sigma_\Gamma(\Gamma) \vdash V : \sigma(X)$$

in the declarative type system.

Analogously, if $\Gamma; \Xi \vDash_F M \Rightarrow X \mid C$ *and* $\sigma = \textsc{unify}(C)$*, then:*

$$\sigma_\Gamma(\Gamma) \vDash M : \sigma(X)$$

in the declarative type system.

In short, the provided program will successfully typecheck in the declarative type system and have the principal type $\sigma(X)$ *if type inference and unification succeed and polymorphic variables are not used.*

Proof. We proceed by induction on the structure of the inference derivation. For each rule, we construct a corresponding declarative typing derivation under the unified context by applying structural induction on the inference derivation, each time invoking the induction hypothesis on subderivations, applying the principal unifier σ, and reconstructing the corresponding declarative derivation under $\sigma_\Gamma(\Gamma)$. Ξ is omitted, as it is assumed to be empty and therefore has no effect. We review each case separately.

- Infer-Unit: Base case. Since no constraints are generated, $\sigma = \emptyset$, we simply apply the empty substitution and directly derive

$$\frac{\textsc{Infer-Unit}}{\Gamma \vDash_\emptyset () \Rightarrow \text{unit} \mid \emptyset} \Longrightarrow \frac{}{\sigma_\Gamma(\Gamma) \vDash () : \text{unit}} \Longrightarrow \frac{\textsc{Unit}}{\Gamma \vDash () : \text{unit}}$$

  in the declarative type system.

- Infer-Const: Base case. Analogous to Infer-Unit with substitution applications where relevant. Uses Const rule in the declarative type system.

- Infer-Var: Base case. Analogous to Infer-Unit with substitution applications where relevant. Uses Var rule in the declarative type system.

- Infer-PolyVar: Base case. Since Ξ is assumed to be empty, polymorphic variables are not used and this case can never happen.

- Infer-Inl and Infer-Inr: Both analogous to Infer-Fun. Using Inl and Inr rules in the declarative type system respectively.

- Infer-Pair: Analogous to Infer-App modulo the added constraint. Uses Pair rule in the declarative type system.

- INFER-FUN: Suppose we have a derivation:

$$\textsc{Infer-Fun}\ \frac{\Gamma, x : \alpha \vDash_{\mathsf{c}\ F} M \Rightarrow X \mid C}{\Gamma \vDash_{\mathsf{v}\ F,\alpha} \texttt{fun}\ x \mapsto M \Rightarrow \alpha \to X \mid C}$$

  Let $\sigma = \textsc{unify}(C)$. By IH, there exists a substitution σ such that $\sigma_\Gamma(\Gamma, x : \alpha) \vDash_{\mathsf{c}} M : \sigma(X)$ in the declarative type system. Since σ unifies the constraint set, α is fresh and no constraints have been introduced to it in C, it is not in σ and remains unaltered. We omit explicit substitution application steps and conclude:

$$\textsc{Fun}\ \frac{\sigma_\Gamma(\Gamma, x : \alpha) \vDash_{\mathsf{c}} M : \sigma(X)}{\sigma_\Gamma(\Gamma) \vDash_{\mathsf{v}} \texttt{fun}\ x \mapsto M : \sigma(\alpha) \to \sigma(X)}$$

  in the declarative type system.

- INFER-RECFUN: Suppose we have a derivation:

$$\textsc{Infer-RecFun}\ \frac{\Gamma, f : \alpha_1, x : \alpha_2 \vDash_{\mathsf{c}\ F} M \Rightarrow X \mid C}{\Gamma \vDash_{\mathsf{v}\ F,\alpha_1,\alpha_2} \texttt{rec fun}\ f\ x \mapsto M \Rightarrow \alpha_1 \mid C \cup \{\alpha_1 \doteq \alpha_2 \to X\}}$$

  Let $\sigma = \textsc{unify}(C \cup \{\alpha_1 = \alpha_2 \to X\})$. By IH, there exists a substitution σ' such that $\sigma'_\Gamma(\Gamma, f : \alpha_1, x : \alpha_2) \vDash_{\mathsf{c}} M : \sigma'(X)$ in the declarative type system. Since σ unifies the combined constraint set and $C \subset (C \cup \{\alpha_1 = \alpha_2 \to X\})$, there exists a substitution σ'' such that $\sigma = \sigma'' \circ \sigma'$. Therefore, we can apply σ'' to the subderivation to obtain $\sigma_\Gamma(\Gamma, f : \alpha_1, x : \alpha_2) \vDash_{\mathsf{c}} M : \sigma(X)$. From the constraint $\alpha_1 \doteq \alpha_2 \to X$, it follows that $\sigma(\alpha_1) = \sigma(\alpha_2) \to \sigma(Y)$. Hence, using the declarative typing rule RECFUN, we can conclude:

$$\textsc{RecFun}\ \frac{\sigma_\Gamma(\Gamma, f : \alpha_1, x : \alpha_2) \vDash_{\mathsf{c}} M : \sigma(X)}{\sigma_\Gamma(\Gamma) \vDash_{\mathsf{v}} \texttt{rec fun}\ f\ x \mapsto M : \sigma(\alpha_2) \to \sigma(Y)}$$

  in the declarative type system.

- INFER-APP: Suppose we have a derivation:

$$\textsc{Infer-App}\ \frac{\Gamma \vDash_{\mathsf{v}\ F_1} V \Rightarrow X \mid C_1 \qquad \Gamma \vDash_{\mathsf{v}\ F_2} W \Rightarrow Y \mid C_2}{\Gamma \vDash_{\mathsf{c}\ F_1,F_2,\alpha} V\ W \Rightarrow \alpha \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\}}$$

Let $\sigma = \textsc{unify}(C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\})$.

By IH, there exist substitutions $\sigma_1 = \textsc{unify}(C_1)$ and $\sigma_2 = \textsc{unify}(C_2)$ such that

$$\sigma_{1\,\Gamma}(\Gamma) \vDash_{\forall} V : \sigma_1(X) \quad \text{and} \quad \sigma_{2\,\Gamma}(\Gamma) \vDash_{\forall} W : \sigma_2(Y)$$

in the declarative type system. Since σ unifies the combined constraint set, and $C_1 \subseteq C$, there exists a substitution σ_1' such that $\sigma = \sigma_1' \circ \sigma_1$. Similarly for σ_2. Therefore, we can apply σ_1' and σ_2' to the above derivations, respectively, to obtain:

$$\sigma_\Gamma(\Gamma) \vDash_{\forall} V : \sigma(X) \quad \text{and} \quad \sigma_\Gamma(\Gamma) \vDash_{\forall} W : \sigma(Y)$$

and from the constraint $X \doteq Y \to \alpha$, it follows that $\sigma(X) = \sigma(Y) \to \sigma(\alpha)$. Hence, using the declarative typing rule Apply, we can conclude:

$$\frac{\sigma_\Gamma(\Gamma) \vDash_{\forall} V : \sigma(Y) \to \sigma(\alpha) \qquad \sigma_\Gamma(\Gamma) \vDash_{\forall} W : \sigma(Y)}{\sigma_\Gamma(\Gamma) \vDash_{\mathsf{c}} V\ W : \sigma(\alpha)}\ \textsc{Apply}$$

in the declarative type system.

- Infer-Return: Analogous to Infer-Fun. Uses Return rule in the declarative type system.
- Infer-Let: Analogous to Infer-App. Uses Let rule in the declarative type system.
- Infer-MatchPair and Infer-MatchSum: Analogous to Infer-App. Using MatchPair and MatchSum rules in the declarative type system respectively.

Therefore, the type inference of $\lambda_{\mathsf{Millet}}$ is sound when polymorphic variables are not used. □

Finally, we present the general soundness theorem and proof for $\lambda_{\mathsf{Millet}}$.

Theorem 3.4 (The type inference of $\lambda_{\mathsf{Millet}}$ is sound)**.** *Let* $\Gamma; \Xi \vDash_{F} V \Rightarrow X \mid C$ *be an inference derivation, with a possibly non-empty* Ξ*. Suppose that* $\sigma = \textsc{unify}(C)$ *is a principal unifier for the constraint set* C*. Then it follows that:*

$$\exists\, \Gamma', V' \,.\, \sigma_\Gamma(\Gamma', \Gamma) \vDash_{\forall} V' : \sigma(X)$$

in the declarative type system, where Γ' *results from linearizing* Ξ *and* V' *differs from* V *only in the names of the linearized polymorphic variables from* Ξ*.*

Analogously, if $\Gamma; \Xi \vDash_{\overline{F}} M \Rightarrow X \mid C$ *and* $\sigma = \textsc{unify}(C)$*, then:*

$$\exists\, \Gamma', M' \,.\, \sigma_\Gamma(\Gamma', \Gamma) \vDash_{\overline{\in}} M' : \sigma(X)$$

in the declarative type system, where Γ' *results from linearizing* Ξ *and* M' *differs from* M *only in the names of the linearized polymorphic variables from* Ξ*.*

Proof. By Theorem 3.1, any inference tree can be linearized, and Lemma 3.2 shows that this transformation preserves structure. In addition, Theorem 3.3 establishes the soundness of $\lambda_{\mathsf{Millet}}$'s type inference in the absence of polymorphic variable usage.

Considering these results, it is clear that any inference tree for which a principal unifier can be constructed can be typed in the declarative type system. If the polymorphic variables are present, the inference tree can be linearized. Therefore, the type inference of $\lambda_{\mathsf{Millet}}$ is sound. □

Note that proving the completeness of the type inference of $\lambda_{\mathsf{Millet}}$ is out of the scope of this thesis, as in the scope of this thesis, we only explore the soundness of the type inference of the temporal version $\lambda_{\mathsf{Mille}[\tau]}$ that is introduced in Section 4.4. Therefore, we are also only interested in the details of the soundness of the underlying implementation of $\lambda_{\mathsf{Millet}}$.

4. Temporal Millet

In this section, we present $\lambda_{\mathsf{Mille}[\tau]}$, the core calculus underlying Temporal Millet, the programming language developed in this thesis as an extension of Millet. Reviewing the surface language of Temporal Millet is deferred to Section 5. $\lambda_{\mathsf{Mille}[\tau]}$ is closely based on Ahman and Žajdela's $\lambda_{[\tau]}$-calculus [7], enriched with constructs from $\lambda_{\mathsf{Millet}}$. Our contributions include the formulation of type inference rules, the design of a unification algorithm, and a proof of the soundness of type inference.

We begin by formalizing the extended core calculus and highlighting the key features it inherits from both $\lambda_{[\tau]}$ and $\lambda_{\mathsf{Millet}}$ in Section 4.1. Following the extended grammar and type system, we define the small-step operational semantics of $\lambda_{\mathsf{Mille}[\tau]}$ in Section 4.2. Then we give an overview of the extended set of inference rules in Section 4.3.1 and the unification algorithm of $\lambda_{\mathsf{Mille}[\tau]}$ in Section 4.3.2. Finally, we establish the soundness of the resulting inference procedure in Section 4.4.

4.1 Core Calculus

$\lambda_{\mathsf{Mille}[\tau]}$ extends $\lambda_{\mathsf{Millet}}$, the core calculus of Millet with temporal modal types, temporally aware algebraic effects and handlers, and an effect system almost entirely based on the $\lambda_{[\tau]}$-calculus introduced by Ahman and Žajdela [7]. It adheres to the fine-grain call-by-value (FGCBV) discipline by distinguishing between values and computations. First, we present the BNF grammar of $\lambda_{\mathsf{Mille}[\tau]}$ in Figure 7.

The set of computations is enriched with four constructs inherited from $\lambda_{[\tau]}$:

- **Algebraic operation calls**, written $\mathsf{op}\ V\ (x\ .\ M)$ for $\mathsf{op} \in \Omega$ that can be redefined by the programmer using effect handlers. For operations, Ω denotes a fixed set of operations with their respective signatures, V denotes the operation's argument, x denotes a variable that holds its result and is bound in M, which denotes its continuation.

- **Effect handling**, written $\texttt{handle}\ M\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N$, enables user-defined interpretations of operations in Ω. The metavariable H represents handler clauses, adopted from the $\lambda_{[\tau]}$-calculus. For detailed overview of effect handling, the reader is referred to [2]. A handler H is written in destructured form as $(x\ .\ k\ .\ M_{\mathsf{op}})_{\mathsf{op}\in\Omega}$, where each operation $\mathsf{op} \in \Omega$ is associated with a corresponding clause. Within each clause:

Values $V, W ::= x$		variable
	$\mid ()$	unit
	$\mid f(V_1, \ldots, V_n)$	constant
	$\mid \texttt{inl}\ V$	left injection
	$\mid \texttt{inr}\ V$	right injection
	$\mid (V, W)$	pairing
	$\mid \texttt{fun}\ x \mapsto M$	function
	$\mid \texttt{rec fun}\ f\ x \mapsto M$	recursive function

Computations $M, N ::= V\ W$		function application
	$\mid \texttt{return}\ V$	return value
	$\mid \texttt{let}\ x = M\ \texttt{in}\ N$	sequential composition
	$\mid \texttt{match}\ V\ \texttt{with}\ \{(x, y) \mapsto M\}$	pattern matching on pair
	$\mid \texttt{match}\ V\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M$	
	$\mid \texttt{inr}\ y \mapsto N\}$	pattern matching on variant
	$\mid \mathrm{op}\ V\ (x\ .\ M)$	algebraic operation call
	$\mid \texttt{delay}\ \tau\ M$	temporal delay
	$\mid \texttt{handle}\ M\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N$	effect handling
	$\mid \texttt{box}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M$	boxing
	$\mid \texttt{unbox}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M$	unboxing

Effect handlers $H ::= (x\ .\ k\ .\ M_{\mathrm{op}})_{\mathrm{op} \in \Omega}$

Figure 7. Values, computations, and effect handlers of $\lambda_{\mathsf{Mille}[\tau]}$.

  - x binds the operation's argument;
  - k binds a delayed continuation;
  - M_{op} specifies how the operation is interpreted by the handler.

- **Temporal delays**, written `delay` τ M, delay starting the computation M for τ temporal grades. The delay construct is similar to operations, but is treated more primitively and can not be redefined using handlers.
- **Boxing** temporal resources, written `box` τ V `as` x `in` M, encapsulates a value V of type X for τ temporal grades of computation. The duration τ may arise not only from explicit delay constructs but also from other operations whose combined execution takes equals τ temporal grades of computation. During this interval, V is inaccessible: the variable x is assigned the modal type $[\tau]X$ in the typing context, ensuring that it cannot be used until it is subsequently unboxed.
- **Unboxing** temporal resources, written `unbox` τ V `as` x `in` M, eliminates the temporal modality and allows use of the delayed value x within M under the assumption that τ temporal grades have passed since the resource was boxed.

The type system of $\lambda_{\mathsf{Mille}[\tau]}$ is almost entirely based on $\lambda_{[\tau]}$ [7]. It is designed to model temporally sensitive computations via temporal grades, modal type constructors, and effect annotations. In Figure 8, we present the syntax of types and typing contexts, which serve as the foundation for the typing judgments introduced later. Notable modifications are the addition of abstract temporal grades t and sum types inherited from $\lambda_{\mathsf{Millet}}$.

A temporal grade τ is drawn from the natural numbers $\mathbb{N}$ and represents a discrete abstraction of temporal cost. It is used to measure the temporal cost of a computation. Temporal grades may also be abstract, denoted t, meaning that their concrete value is not yet determined. To denote the addition of abstract temporal grades, we also introduce a binary operation $\oplus$. We do this to keep a strict notational difference between the addition of two abstract temporal grades and the addition of two temporal grades that are known to be concrete natural numbers. The addition of two abstract temporal grades τ and τ' is written $\tau \oplus \tau'$.

Temporal cost may arise from explicit delay constructs, but also from the execution of algebraic operations or constructs that incur temporal cost. Temporal grades appear

in both modal types $[\tau]X$ and computation types $X \,!\, \tau$, allowing precise tracking of temporal behavior in programs. Although the underlying theory permits general monoidal structures, we fix concrete temporal grades to natural numbers in this section for simplicity and concreteness as in [7].

The types of $\lambda_{\mathsf{Mille}[\tau]}$ are mostly the same as those of $\lambda_{\mathsf{Millet}}$, simply extended with temporal grade annotations. The types of $\lambda_{\mathsf{Mille}[\tau]}$ are categorized into three syntactic classes:

- **Ground types**, denoted A, B, C, represent basic data types, with $[\tau]A$ denoting a basic value of type A that becomes available only after τ temporal grades have passed. Ground types are used by operations in Ω.
- **Value types**, denoted X, Y, Z, extend ground types to include function types and temporal modalities over general value types. The function type $X \to Y \,!\, \tau$ describes a function from X to Y that incurs a temporal grade τ upon application.
- **Computation types** are of the form $X \,!\, \tau$ and describe computations that yield a value of type X after τ temporal grades of computation.

Typing contexts Γ maintain both term variable bindings and temporal assumptions. This is the general style for representing Fitch-style modal types [24]. A judgment of the form $\Gamma, \langle\tau\rangle$ asserts that the current point in the program's execution occurs at least τ temporal grades after a previous reference point. For example $\Gamma = x : [7]X, \langle 5\rangle, y : Y, \langle 3\rangle, z : Z$ denotes that a boxed value x was brought into scope, then 5 temporal grades of computation passed, followed by y being brought into scope, then another 3 temporal grades of computation passed and z was brought into scope. Such contexts enables precise reasoning about the temporal availability of variables and effectful computation [7].

Temporal grades	$\tau ::= n \in \mathbb{N} \mid t \mid \tau \oplus \tau$
Ground types	$A, B, C ::= \alpha \mid \text{unit} \mid b \mid A + B \mid A \times B \mid [\tau]A$
Value types	$X, Y, Z ::= A \mid X + Y \mid X \times Y \mid X \to Y \,!\, \tau \mid [\tau]X$
Computation types	$X \,!\, \tau$
Contexts	$\Gamma ::= \emptyset \mid \Gamma, x : X \mid \Gamma, \langle\tau\rangle$

Figure 8. Temporal grades, types, and typing contexts of $\lambda_{\mathsf{Mille}[\tau]}$.

Well-typed values and computations of $\lambda_{\mathsf{Mille}[\tau]}$ are specified using two separate typing judgments $\Gamma \vDash_{\mathrm{v}} V : X$ and $\Gamma \vDash_{\mathrm{c}} M : X \,!\, \tau$, respectively. These state that under context Γ, a value V has type X, and a computation M yields a result of type X after τ temporal grades of computation, respectively.

Recall that a term is considered well-typed if it can be assigned a type according to the declarative typing rules of the language under a given typing context. The full set of typing rules for $\lambda_{\mathsf{Mille}[\tau]}$ is presented in Figure 9.

From here on, we need to compute the sum of temporal grades in non-polymorphic contexts Γ to formally describe unboxing operations. For that, we bring in the definition from [7].

Definition 4.1 (Sum of temporal grades in a non-polymorphic context)**.** *Let Γ be a non-polymorphic context. The operation τ_Γ is recursively defined as follows:*

$$\tau_\emptyset \;=\; 0, \qquad \tau_{(\Gamma,\, x \,:\, X)} \;=\; \tau_\Gamma, \qquad \tau_{(\Gamma,\, \langle \tau \rangle)} \;=\; \tau_\Gamma \oplus \tau$$

We can observe that variables $x : X$ do not contribute to the sum of temporal grades, and the sum of temporal grades in an empty context is 0.

A brief overview of typing rules that are either inherited from $\lambda_{[\tau]}$ or newly enriched to incorporate temporal effects is provided below. Rules that are syntactically and semantically unchanged from $\lambda_{\mathsf{Millet}}$ are omitted, as they were described in detail in Section 3.2.2.

- Fun defines the typing of function abstractions. If the computation M is well-typed with type $Y \,!\, \tau$, denoting that it produces a result of type Y after τ temporal grades have passed since its execution started, under an extended context that contains the binding $x : X$, then the function `fun` $x \mapsto M$ is well-typed as a value of function type $X \to Y \,!\, \tau$.
- RecFun defines the typing of recursive functions. Recursive functions must be pure in $\lambda_{\mathsf{Mille}[\tau]}$, meaning their bodies must be functions without temporal effects, to prevent unbounded temporal accumulation in recursive calls. This rule specializes RecFun from $\lambda_{\mathsf{Millet}}$ to the temporal setting.

$$\textsc{Unit}\ \frac{}{\Gamma \vdash_{\mathsf{v}} () : \mathsf{unit}} \qquad \textsc{Const}\ \frac{(\Gamma \vdash_{\mathsf{v}} V_i : b_i)_{1 \le i \le n}}{\Gamma \vdash_{\mathsf{v}} f(V_1, \ldots, V_n) : b} \qquad \textsc{Var}\ \frac{(x : X) \in \Gamma}{\Gamma \vdash_{\mathsf{v}} x : X} \qquad \textsc{Inl}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X}{\Gamma \vdash_{\mathsf{v}} \mathtt{inl}\ V : X + Y}$$

$$\textsc{Inr}\ \frac{\Gamma \vdash_{\mathsf{v}} V : Y}{\Gamma \vdash_{\mathsf{v}} \mathtt{inr}\ V : X + Y} \qquad \textsc{Pair}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X \qquad \Gamma \vdash_{\mathsf{v}} W : Y}{\Gamma \vdash_{\mathsf{v}} (V, W) : X \times Y} \qquad \textsc{Fun}\ \frac{\Gamma, x : X \vdash_{\mathsf{c}} M : Y \,!\, \tau}{\Gamma \vdash_{\mathsf{v}} \mathtt{fun}\ x \mapsto M : X \to Y \,!\, \tau}$$

$$\textsc{RecFun}\ \frac{\Gamma, f : X \to Y \,!\, 0, x : X \vdash_{\mathsf{c}} M : Y \,!\, 0}{\Gamma \vdash_{\mathsf{v}} \mathtt{rec\ fun}\ f\ x \mapsto M : X \to Y \,!\, 0} \qquad \textsc{Apply}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X \to Y \,!\, \tau \qquad \Gamma \vdash_{\mathsf{v}} W : X}{\Gamma \vdash_{\mathsf{c}} V\ W : Y \,!\, \tau}$$

$$\textsc{Return}\ \frac{\Gamma \vdash_{\mathsf{c}} V : X}{\Gamma \vdash_{\mathsf{c}} \mathtt{return}\ V : X \,!\, 0} \qquad \textsc{Let}\ \frac{\Gamma \vdash_{\mathsf{c}} M : X \,!\, \tau \qquad \Gamma, \langle \tau \rangle, x : X \vdash_{\mathsf{c}} N : Y \,!\, \tau'}{\Gamma \vdash_{\mathsf{c}} \mathtt{let}\ x = M\ \mathtt{in}\ N : Y \,!\, \tau \oplus \tau'}$$

$$\textsc{MatchPair}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X \times Y \qquad \Gamma, x : X, y : Y \vdash_{\mathsf{c}} M : Z \,!\, \tau}{\Gamma \vdash_{\mathsf{c}} \mathtt{match}\ V\ \mathtt{with}\ \{(x, y) \mapsto M\} : Z \,!\, \tau}$$

$$\textsc{MatchSum}\ \frac{\Gamma \vdash_{\mathsf{v}} V : X + Y \qquad \Gamma, x : X \vdash_{\mathsf{c}} M : Z \,!\, \tau \qquad \Gamma, y : Y \vdash_{\mathsf{c}} N : Z \,!\, \tau}{\Gamma \vdash_{\mathsf{c}} \mathtt{match}\ V\ \mathtt{with}\ \{\mathtt{inl}\ x \mapsto M \mid \mathtt{inr}\ y \mapsto N\} : Z \,!\, \tau}$$

$$\textsc{Op}\ \frac{\Gamma \vdash_{\mathsf{v}} V : A_{\mathrm{op}} \qquad \Gamma, \langle \tau_{\mathrm{op}} \rangle, x : B_{\mathrm{op}} \vdash_{\mathsf{v}} M : X \,!\, \tau}{\Gamma \vdash_{\mathsf{c}} \mathrm{op}\ V\ (x\ .\ M) : X \,!\, \tau_{\mathrm{op}} \oplus \tau} \qquad \textsc{Delay}\ \frac{\Gamma, \langle \tau \rangle \vdash_{\mathsf{c}} M : X \,!\, \tau'}{\Gamma \vdash_{\mathsf{c}} \mathtt{delay}\ \tau\ M : X \,!\, \tau \oplus \tau'}$$

$$\textsc{Handle}\ \frac{\Gamma \vdash_{\mathsf{c}} M : X \,!\, \tau \qquad \Gamma, \langle \tau \rangle, x : X \vdash_{\mathsf{c}} N : Y \,!\, \tau' \qquad H = (y\ .\ k\ .\ M_{\mathrm{op}})_{\mathrm{op} \in \Omega} \qquad (\forall \tau''.\ \Gamma, z : A_{\mathrm{op}}, k : [\tau_{\mathrm{op}}](B_{\mathrm{op}} \to Y \,!\, \tau'') \vdash_{\mathsf{c}} M_{\mathrm{op}} : Y \,!\, \tau_{\mathrm{op}} \oplus \tau'')_{\mathrm{op} \in \Omega}}{\Gamma \vdash_{\mathsf{c}} \mathtt{handle}\ M\ \mathtt{with}\ H\ \mathtt{to}\ x\ \mathtt{in}\ N : Y \,!\, \tau \oplus \tau'}$$

$$\textsc{Box}\ \frac{\Gamma, \langle \tau \rangle \vdash_{\mathsf{v}} V : X \qquad \Gamma, x : [\tau]X \vdash_{\mathsf{c}} N : Y \,!\, \tau'}{\Gamma \vdash_{\mathsf{c}} \mathtt{box}\ \tau\ V\ \mathtt{as}\ x\ \mathtt{in}\ N : Y \,!\, \tau'} \qquad \textsc{Unbox}\ \frac{\tau \le \tau_\Gamma \qquad \Gamma - \tau \vdash_{\mathsf{v}} V : [\tau]X \qquad \Gamma, x : X \vdash_{\mathsf{c}} N : Y \,!\, \tau'}{\Gamma \vdash_{\mathsf{c}} \mathtt{unbox}\ \tau\ V\ \mathtt{as}\ x\ \mathtt{in}\ N : Y \,!\, \tau'}$$

Figure 9. Typing rules of $\lambda_{\mathsf{Mille}[\tau]}$.

- Apply, MatchPair, MatchSum are inherited from λ_{Millet} but are now extended to include temporal grade annotations in the resulting computation types. Their structure remains otherwise unchanged.

- Return lifts a value into the computation layer, trivially returning it without performing any effects. If V is a well-typed value of type X, then `return` V is a well-typed computation of type $X \,!\, 0$.

- Let performs sequential composition while tracking temporal effects. If M incurs a temporal grade τ and its result is bound to $x : X$, and N incurs a temporal grade τ' under the extended context $\Gamma, \langle\tau\rangle, x : X$, then `let` $x = M$ `in` N is a computation of type $Y \,!\, \tau \oplus \tau'$.

- Op defines the typing of algebraic operations of the form op V $(x \,.\, M)$, which incur a temporal grade τ_{op} associated with the operation through its signature in Ω. The continuation M is evaluated under a temporal assumption of at least τ_{op}, and produces a result after an additional τ temporal grades have passed, corresponding to the temporal grade of evaluating the continuation. This yields a total temporal grade $\tau_{\text{op}} \oplus \tau$.

- Delay defines the typing of the construct `delay` τ M that postpones the execution of M by τ temporal grades. To type it, M must be typeable under a temporal assumption of at least τ, then the resulting delay accumulates as $\tau \oplus \tau'$.

- Handle defines the typing of handlers. Effect handling `handle` M `with` H `to` z `in` N intercepts algebraic operations from M, using a handler H to define how each operation is handled. After M completes, a temporal assumption τ is appended to the context, along with its result, which is bound to the variable z. Then N is evaluated with an additional temporal grade τ'. Each handler clause M_{op} must be typeable under a continuation type of the form $[\tau_{\text{op}}](B_{\text{op}} \to Y \,!\, \tau'')$, ensuring temporal compositionality. Temporal compositionality here is the property that handler clauses remain well-typed for all possible temporal grades for continuations, allowing the total temporal grade of a handled computation to be derived solely by composing the temporal grades of its subcomputations. The compositionality is induced by universally quantifying over all τ''-s. The total temporal grade is $\tau \oplus \tau'$.

- Box defines the typing of boxing temporal resources. The construct `box` τ V `as` x `in` M introduces a modal type. The value V is required to become available after τ temporal grades of computation, and x is bound to a placeholder for that delayed value with type $[\tau]X$ in the scope of M. This means that x does not refer to V directly, but to a resource that will yield V after τ temporal grades of computation.

- Unbox defines the typing of unboxing temporal resources. Unbox extracts a value by removing its modality, while requiring that $\tau \leq \tau_\Gamma$ (i.e., the context's current temporal assumption is sufficient to access a value that requires a temporal assumption of τ to access). Recall the definition of the operation τ_Γ, provided in Definition 4.1. It is also required that the value that is to be unboxed, exists in the context $\Gamma - \tau$ with the modal type $[\tau]X$. For the complete recursive definition of $\Gamma - \tau$, the reader is referred to [7].

  We illustrate the semantics of the operation with an example. Suppose that we have a context $\Gamma = x : [7]X, \langle 5\rangle, y : Y, \langle 3\rangle, z : Z$ and $\tau = 7$ in an unbox operation. Then $\tau_\Gamma = 5 \oplus 3 = 5 + 3 = 8$ and the $\tau \leq \tau_\Gamma$ condition is met because $7 \leq 8$. $\Gamma - \tau$ in the concrete example would result in a context $\Gamma - 7 = x : [7]X, \langle 1\rangle$. Therefore the condition $\Gamma - 7 \vDash_\forall x : [7]X$ is also met. Assuming that if x' is of type X, the computation N is of type $Y \,!\, \tau'$, the term `unbox` 7 x `as` x' `in` N will be well-typed with the type $Y \,!\, \tau'$. For the details on how τ_Γ is computed and the full definition of the operation $\Gamma - \tau$, the reader is referred to [7].

Notice that the value V in Unbox is always guaranteed to be a variable.

Theorem 4.1 (The argument value V of Unbox is a variable)**.** *If* $\Gamma \vDash_\forall V : [\tau]X$*, then* $V = x$ *for some variable* $x \in \Gamma$*.*

Proof. We proceed by induction on the derivation of $\Gamma \vDash_\forall V : [\tau]X$.

- Var

  If $\dfrac{(y : [\tau]X) \in \Gamma}{\Gamma \vDash_\forall y : [\tau]X}$ (Var), then $x = y$.

- Unit — impossible, as $[\tau]X$ is not a unit type.

- Const — impossible, as $[\tau]X$ is not a base type.
- Inl — impossible, as $[\tau]X$ is not a sum type.
- Inr — impossible, as $[\tau]X$ is not a sum type.
- Pair — impossible, as $[\tau]X$ is not a product type.
- Fun — impossible, as $[\tau]X$ is not a function type.
- RecFun — impossible, as $[\tau]X$ is not a function type.

Trivially, because $[\tau]X$ is not a computation type, none of the rules for typing computations can apply. Therefore, the argument value V of Unbox can not be anything other than a variable. □

4.2 Semantics

In this section, we present the operational semantics of $\lambda_{\mathsf{Mille}[\tau]}$, shown in Figure 10 and Figure 11. It is largely based on the semantics of $\lambda_{[\tau]}$ [7]. The only exceptions are the rules Sem-MatchInl and Sem-MatchInr, which are inherited from $\lambda_{\mathsf{Millet}}$.

Recall from Section 3.3 that in $\lambda_{\mathsf{Millet}}$, reduction applies only to computations. Values are treated as terminal – they do not reduce further.

The key extension in $\lambda_{\mathsf{Mille}[\tau]}$ is the inclusion of an explicit *state* $\mathbb{S}$ in the reduction relation. This addition enables reasoning about the temporal structure of available resources. A state $\mathbb{S}$ corresponds conceptually to a typing context Γ, but instead of tracking variable typings, it records when modally typed resources become available. The formal connection between states and contexts is developed in [7]. For the purposes of this thesis, the following definition is sufficient:

$$\mathbb{S} ::= \emptyset \mid \mathbb{S}, \langle \tau \rangle \mid \mathbb{S}, x \mapsto V$$

Here, $\langle \tau \rangle$ denotes the addition of a temporal value to the state, signifying that a computation with temporal cost has occurred, and $x \mapsto V$ records that a modally typed variable x maps to a value V, which is returned when the modally typed variable is unboxed.

The small-step reduction relation in $\lambda_{\mathsf{Mille}[\tau]}$ is defined over a pair of a state and a computation:

$$\langle \mathbb{S} \mid M \rangle \rightsquigarrow \langle \mathbb{S}' \mid M' \rangle$$

This relation expresses that under state $\mathbb{S}$, the computation M takes a single reduction step to M' and updates the state to $\mathbb{S}'$.

Figure 10 lists the reduction rules of $\lambda_{\mathsf{Mille}[\tau]}$ that are inherited from $\lambda_{\mathsf{Millet}}$. The only difference is that they now include the aforementioned state $\mathbb{S}$. Similarly to Section 3.3, we adopt *Barendregt's convention* [12], which assumes that bound variables are always chosen distinct from free variables in any given context. As a result, we omit explicit variable renaming.

$$\textsc{Sem-App}\quad \frac{}{\langle \mathbb{S} \mid (\texttt{fun}\ x \mapsto M)\ V \rangle \rightsquigarrow \langle \mathbb{S} \mid M[V/x] \rangle}$$

$$\textsc{Sem-App-Rec}\quad \frac{}{\langle \mathbb{S} \mid (\texttt{rec fun}\ f\ x \mapsto M)\ V \rangle \rightsquigarrow \langle \mathbb{S} \mid M[V/x][(\texttt{rec fun}\ f\ x \mapsto M/f)] \rangle}$$

$$\textsc{Sem-Let-Return}\quad \frac{}{\langle \mathbb{S} \mid \texttt{let}\ x = (\texttt{return}\ V)\ \texttt{in}\ N \rangle \rightsquigarrow \langle \mathbb{S} \mid N[V/x] \rangle}$$

$$\textsc{Sem-Let-Cong}\quad \frac{\langle \mathbb{S} \mid M \rangle \rightsquigarrow \langle \mathbb{S}' \mid M' \rangle}{\langle \mathbb{S} \mid \texttt{let}\ x = M\ \texttt{in}\ N \rangle \rightsquigarrow \langle \mathbb{S}' \mid \texttt{let}\ x = M'\ \texttt{in}\ N \rangle}$$

$$\textsc{Sem-Match-Pair}\quad \frac{}{\langle \mathbb{S} \mid \texttt{match}\ (V, W)\ \texttt{with}\ \{(x, y) \mapsto N\} \rangle \rightsquigarrow \langle \mathbb{S} \mid N[V/x, W/y] \rangle}$$

$$\textsc{Sem-MatchInl}\quad \frac{}{\langle \mathbb{S} \mid \texttt{match}\ (\texttt{inl}\ V)\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\} \rangle \rightsquigarrow \langle \mathbb{S} \mid M[V/x] \rangle}$$

$$\textsc{Sem-MatchInr}\quad \frac{}{\langle \mathbb{S} \mid \texttt{match}\ (\texttt{inr}\ V)\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\} \rangle \rightsquigarrow \langle \mathbb{S} \mid N[V/y] \rangle}$$

Figure 10. Small-step operational semantics of $\lambda_{\mathsf{Mille}[\tau]}$, based almost entirely on the semantics of $\lambda_{\mathsf{Millet}}$, extended with states.

Figure 11 lists the reduction rules of $\lambda_{\mathsf{Mille}[\tau]}$ that are inherited from $\lambda_{[\tau]}$. The rules Sem-Delay, Sem-Box, and Sem-Unbox explicitly manipulate the state to implement temporally sensitive features such as delayed availability and resource unboxing. Note that the operation $\mathbb{S}[y]$ denotes a lookup operation for the variable y in the state $\mathbb{S}$. The precondition $y \in \mathbb{S}$ in the Sem-Unbox rule ensures that this operation is defined. For the complete definition of the operation, the reader is referred to [7].

Sem-Let-Op

$$\langle \mathbb{S} \mid \texttt{let}\ x = \mathrm{op}\ V\ (y\ .\ M)\ \texttt{in}\ N \rangle \rightsquigarrow \langle \mathbb{S} \mid \mathrm{op}\ V\ (y\ .\ \texttt{let}\ x = M\ \texttt{in}\ N) \rangle$$

Sem-Handle-Cong

$$\frac{\langle \mathbb{S} \mid M \rangle \rightsquigarrow \langle \mathbb{S}' \mid M' \rangle}{\langle \mathbb{S} \mid \texttt{handle}\ M\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N \rangle \rightsquigarrow \langle \mathbb{S}' \mid \texttt{handle}\ M'\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N \rangle}$$

Sem-Handle-Return

$$\langle \mathbb{S} \mid \texttt{handle}\ (\texttt{return}\ V)\ \texttt{with}\ H\texttt{to}\ z\ \texttt{in}\ N \rangle \rightsquigarrow \langle \mathbb{S} \mid N[V/z] \rangle$$

Sem-Handle-Op

$$\frac{H = (x\ .\ k\ .\ M_{\mathrm{op}})_{\mathrm{op} \in \Omega}}{\begin{array}{l}\langle \mathbb{S} \mid \texttt{handle}\ (\mathrm{op}\ V\ (y\ .\ M))\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N \rangle \rightsquigarrow \\ \langle \mathbb{S} \mid \texttt{box}\ \tau_{\mathrm{op}}\ (\texttt{fun}\ y \mapsto \texttt{handle}\ M\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N)\ \texttt{as}\ w\ \texttt{in}\ M_{\mathrm{op}}[V/x, w/k] \rangle\end{array}}$$

Sem-Delay

$$\langle \mathbb{S} \mid \texttt{delay}\ \tau\ M \rangle \rightsquigarrow \langle \mathbb{S}, \langle \tau \rangle \mid M \rangle$$

Sem-Box

$$\langle \mathbb{S} \mid \texttt{box}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ N \rangle \rightsquigarrow \langle \mathbb{S}, x \mapsto V \mid N \rangle$$

Sem-Unbox $\quad y \in \mathbb{S}$

$$\langle \mathbb{S} \mid \texttt{unbox}\ \tau\ y\ \texttt{as}\ x\ \texttt{in}\ N \rangle \rightsquigarrow \langle \mathbb{S} \mid N[\mathbb{S}[y]/x] \rangle$$

Figure 11. Small-step operational semantics of $\lambda_{\mathsf{Mille}[\tau]}$, inherited from $\lambda_{[\tau]}$.

4.3 Type Inference

Temporal Millet, like $\lambda_{\mathsf{Millet}}$, is based on a Hindley–Milner style type inference system. This section details the extensions to $\lambda_{\mathsf{Millet}}$'s type inference rules that enable the tracking of temporal information. A central contribution of this thesis is the formalization of these

extended rules and the development of the corresponding unification algorithm, alongside the implementation of Temporal Millet presented in Section 5.

4.3.1 Inference Rules

Recall that the declarative type system of $\lambda_{\mathsf{Millet}}$ allows type variables α, allowing unannotated terms and polymorphism. The same holds true for $\lambda_{\mathsf{Mille}[\tau]}$. Types and constraints are inferred by using similar syntax-directed type inference judgments for values and computations. The only difference is that in $\lambda_{\mathsf{Mille}[\tau]}$, computation types include temporal grade annotations. The type inference judgments of $\lambda_{\mathsf{Mille}[\tau]}$ are the following:

$$\Gamma; \Xi \vDash_{F} V \Rightarrow X \mid C \quad \text{and} \quad \Gamma; \Xi \vDash_{F} M \Rightarrow X \,!\, \tau \mid C.$$

The constraint set C is extended to include constraints over temporal grades τ in addition to types. Besides the type equality constraints $X \doteq Y$ (introduced in Section 3.4.1), C may now also contain temporal constraints of the forms $\tau \doteq \tau'$ and $\tau \dot{\geq} \tau'$. These express, respectively, that two temporal grades are equal, or that one is greater than or equal to the other. Constraints of the latter form are used to defer the validity check of $\Gamma - \tau$ in the Infer-Unbox rule. This deferral increases the likelihood that unification has progressed sufficiently to instantiate abstract temporal grades with concrete values, thereby enabling $\lambda_{\mathsf{Mille}[\tau]}$ to infer types for a broader class of programs.

Note that all assumptions presented in Section 3.4.1 are still relevant. We present the first set of type inference rules of $\lambda_{\mathsf{Mille}[\tau]}$ in Figure 12. The rules Infer-Unit, Infer-Const, Infer-Var, Infer-PolyVar, Infer-Inl, Inter-Inr and Infer-Pair are directly inherited from $\lambda_{\mathsf{Millet}}$ and are thus identical to those introduced in Section 3.4.1. Therefore, they, and their respective descriptions, are omitted for brevity. We review each rule similarly to Section 3.4.1, and highlight the modifications induced by introducing temporal grades in computation types:

- Infer-Return defines type inference for returning a value from a computation with no additional temporal cost. If V can be inferred to have type X under contexts $\Gamma; \Xi$, introducing the set of fresh variables F and the set of constraints C, then the term `return` V is inferred to have computation type $X \,!\, 0$, indicating that the computation produces a result of type X while incurring no temporal cost. The set of fresh variables and the set of constraints remain exactly as obtained from inferring V.

INFER-RETURN

$$\frac{\Gamma; \Xi \vDash_{F} V \Rightarrow X \mid C}{\Gamma; \Xi \vDash_{F} \texttt{return}\ V \Rightarrow X\ !\ 0 \mid C}$$

INFER-FUN

$$\frac{\Gamma, x : \alpha; \Xi \vDash_{F} M \Rightarrow X\ !\ \tau \mid C}{\Gamma; \Xi \vDash_{F,\alpha} \texttt{fun}\ x \mapsto M \Rightarrow \alpha \to X\ !\ \tau \mid C}$$

INFER-RECFUN

$$\frac{\Gamma, f : \alpha_1, x : \alpha_2; \Xi \vDash_{F} M \Rightarrow X\ !\ \tau \mid C}{\Gamma; \Xi \vDash_{F,\alpha_1,\alpha_2} \texttt{rec fun}\ f\ x \mapsto M \Rightarrow \alpha_2 \to X\ !\ \tau \mid C \cup \{(\alpha_1 \doteq \alpha_2 \to X\ !\ \tau), (\tau \doteq 0)\}}$$

INFER-APP

$$\frac{\Gamma; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma; \Xi \vDash_{F_2} W \Rightarrow Y \mid C_2}{\Gamma; \Xi \vDash_{F_1,F_2,\alpha,t} V\ W \Rightarrow \alpha\ !\ t \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\ !\ t\}}$$

INFER-LET

$$\frac{\Gamma; \Xi \vDash_{F_1} M \Rightarrow X\ !\ \tau_1 \mid C_1 \qquad \Gamma, \langle t \rangle, x : \alpha; \Xi \vDash_{F_2} N \Rightarrow Y\ !\ \tau_2 \mid C_2}{\Gamma; \Xi \vDash_{F_1,F_2,\alpha,t} \texttt{let}\ x = M\ \texttt{in}\ N \Rightarrow Y\ !\ t \oplus \tau_2 \mid C_1 \cup C_2 \cup \{(X \doteq \alpha), (\tau_1 \doteq t)\}}$$

INFER-MATCHPAIR

$$\frac{\Gamma; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma, x : \alpha_1, y : \alpha_2; \Xi \vDash_{F_2} M \Rightarrow Y\ !\ \tau \mid C_2}{\begin{array}{c}\Gamma; \Xi \vDash_{F_1,F_2,\alpha_1,\alpha_2} \texttt{match}\ V\ \texttt{with}\ \{(x, y) \mapsto M\} \Rightarrow \\ Y\ !\ \tau \mid C_1 \cup C_2 \cup \{X \doteq \alpha_1 \times \alpha_2\}\end{array}}$$

INFER-MATCHSUM

$$\frac{\begin{array}{c}\Gamma; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \\ \Gamma, x : \alpha_1; \Xi \vDash_{F_2} M \Rightarrow Y\ !\ \tau_1 \mid C_2 \qquad \Gamma, y : \alpha_2; \Xi \vDash_{F_3} N \Rightarrow Z\ !\ \tau_2 \mid C_3\end{array}}{\begin{array}{c}\Gamma; \Xi \vDash_{F_1,F_2,F_3,\alpha_1,\alpha_2,\alpha_3,\alpha_4,t} \texttt{match}\ V\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\} \\ \Rightarrow \alpha_4\ !\ t \mid C_1 \cup C_2 \cup C_3 \cup \{(X \doteq \alpha_1 + \alpha_2), (Y \doteq \alpha_4), (\tau_1 \doteq t), (Z \doteq \alpha_4), (\tau_2 \doteq t)\}\end{array}}$$

Figure 12. Inference rules for $\lambda_{\mathsf{Mille}[\tau]}$ constructs inherited from $\lambda_{\mathsf{Millet}}$, extended with temporal computation types.

- Infer-Fun defines type inference for functions whose bodies are computations with temporal effects. If the function body M can be inferred to have computation type $X \,!\, \tau$ under the contexts $\Gamma, x : \alpha; \Xi$, where α is a fresh type variable representing the argument type, then the term `fun` $x \mapsto M$ is inferred to have function type $\alpha \to X \,!\, \tau$. The set of fresh variables consists of those obtained from inferring M together with α, and the set of constraints remains equal to that obtained from inferring M.

- Infer-RecFun defines type inference for recursive functions whose bodies are temporal computations. If the body M can be inferred to have computation type $X \,!\, \tau$ under the contexts $\Gamma, f : \alpha_1, x : \alpha_2; \Xi$, where α_1 and α_2 are fresh type variables representing the function's type and argument type respectively, then the term `rec fun` $f\ x \mapsto M$ is inferred to have function type $\alpha_2 \to X \,!\, \tau$. The set of fresh variables consists of those obtained from inferring M together with α_1 and α_2. The set of constraints is the union of those obtained from inferring M with the additional constraints $(\alpha_1 \doteq \alpha_2 \to X \,!\, \tau)$, ensuring that the function type matches its argument and result types, and $(\tau \doteq 0)$, ensuring that the function itself introduces no additional temporal cost. This is necessary to prevent unbounded temporal accumulation in recursive calls. See Section 5.3 for further discussion on this.

- Infer-App defines type inference for function application with temporal effects. If V can be inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, and W can be inferred to have type Y, introducing the set of fresh variables F_2 and the set of constraints C_2, then the application $V\ W$ is inferred to have computation type $\alpha \,!\, t$, where α and t are fresh variables representing the result type and temporal grade of running the computation in the function body, respectively. The set of fresh variables is F_1, F_2, α, t, and the set of constraints is $C_1 \cup C_2 \cup \{\, X \doteq Y \to \alpha \,!\, t \,\}$, ensuring that the function type of V matches the argument type of W and yields the annotated result.

- Infer-Let defines type inference for `let`-bindings with temporal composition. If M can be inferred to have computation type $X \,!\, \tau_1$, introducing the set of fresh variables F_1 and the set of constraints C_1, and N can be inferred to have computation type $Y \,!\, \tau_2$ under the extended contexts $\Gamma, \langle t \rangle, x : \alpha; \Xi$, introducing the set of fresh

variables F_2 and the set of constraints C_2, then the term $\texttt{let}\ x = M\ \texttt{in}\ N$ is inferred to have computation type $Y\ !\ t \oplus \tau_2$. The set of fresh variables is F_1, F_2, α, t, and the set of constraints is $C_1 \cup C_2 \cup \{\, X \doteq \alpha,\ \tau_1 \doteq t \,\}$, where t captures the temporal grade of evaluating M and $\oplus$ composes it with the temporal grade of N. These constraints ensure that the bound variable x has the same type as the value computed by M, and that t is equal to the temporal grade of the computation M.

- Infer-MatchPair defines type inference for pattern matching on pairs in computations with temporal effects. This case is identical to the one presented in Section 3.4.1, modulo the temporal grade annotation in the return type.

- Infer-MatchSum defines type inference for pattern matching on sum types in computations with temporal effects. If V can be inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, the left branch M can be inferred to have computation type $Y\ !\ \tau_1$ under $\Gamma, x : \alpha_1; \Xi$, introducing F_2 and C_2, and the right branch N can be inferred to have computation type $Z\ !\ \tau_2$ under $\Gamma, y : \alpha_2; \Xi$, introducing F_3 and C_3, then the term $\texttt{match}\ V\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\}$ is inferred to have computation type $\alpha_4\ !\ t$, where α_4 and t are fresh variables representing the common result type and the common temporal grade of the two branches, respectively. The set of fresh variables is $F_1, F_2, F_3, \alpha_1, \alpha_2, \alpha_3, \alpha_4, t$, and the set of constraints is $C_1 \cup C_2 \cup C_3 \cup \{\, X \doteq \alpha_1 + \alpha_2,\ Y \doteq \alpha_4,\ \tau_1 \doteq t,\ Z \doteq \alpha_4,\ \tau_2 \doteq t \,\}$. These constraints ensure that:

  1. V has a sum type whose left and right components match the types bound in the respective branches;

  2. both branches return the same result type α_4;

  3. both branches have the same temporal grade t, enforcing temporal uniformity in the match expression.

Next, we present the second set of type inference rules of $\lambda_{\mathsf{Mille}[\tau]}$, covering the terms inherited from λ_τ [7] in Figure 13. We define an auxiliary operation LookupCtxTau in order to keep the Infer-Unbox rule cleaner. It is defined as follows:

Definition 4.2 (Context based temporal grade lookup)**.** *Let Γ be a non-polymorphic context, let Ξ be a context of type schemes, and let x be a variable of type X. The lookup operation*

INFER-OP

$$\frac{\Gamma;\Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma, \langle \tau_{\mathrm{op}} \rangle, x : B_{\mathrm{op}}; \Xi \vDash_{F_2} M \Rightarrow Y \,!\, \tau \mid C_2}{\Gamma;\Xi \vDash_{F_1,F_2} \mathrm{op}\ V\ (x\ .\ M) \Rightarrow Y \,!\, \tau_{\mathrm{op}} \oplus \tau \mid C_1 \cup C_2 \cup \{X \doteq A_{\mathrm{op}}\}}$$

INFER-HANDLE

$$\frac{\begin{array}{c}\Gamma;\Xi \vDash_{F_1} M \Rightarrow X \,!\, \tau_1 \mid C_1 \\ \Gamma, \langle t \rangle, x : \alpha; \Xi \vDash_{F_2} N \Rightarrow Y \,!\, \tau_2 \mid C_2 \qquad H = (y\ .\ k\ .\ M_{\mathrm{op}})_{\mathrm{op}\in\Omega} \\ \forall \mathrm{op} \in \Omega.\ \Gamma, z : A_{\mathrm{op}}, k : [\tau_{\mathrm{op}}](B_{\mathrm{op}} \to Z_{\mathrm{op}} \,!\, t_{\mathrm{op}}); \Xi \vDash_{F_{\mathrm{op}}} M_{\mathrm{op}} \Rightarrow Z_{\mathrm{op}} \,!\, \tau'_{\mathrm{op}} \mid C_{\mathrm{op}}\end{array}}{\begin{array}{c}\Gamma;\Xi \vDash_{(F_{\mathrm{op}})_{\mathrm{op}\in\Omega},(t_{\mathrm{op}})_{\mathrm{op}\in\Omega},F_1,F_2,\alpha,t} \texttt{handle}\ M\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N \Rightarrow \\ Y \,!\, t \oplus \tau_2 \mid (\bigcup_{\mathrm{op}\in\Omega} (C_{\mathrm{op}} \cup \{(Z_{\mathrm{op}} \doteq Y), (\tau'_{\mathrm{op}} \doteq \tau_{\mathrm{op}} \oplus t_{\mathrm{op}})\})) \cup C_1 \cup C_2 \cup \{(X \doteq \alpha), (\tau_1 \doteq t)\}\end{array}}$$

INFER-DELAY

$$\frac{\Gamma, \langle \tau \rangle; \Xi \vDash_F M \Rightarrow X \,!\, \tau' \mid C}{\Gamma;\Xi \vDash_F \texttt{delay}\ \tau\ M \Rightarrow X \,!\, \tau \oplus \tau' \mid C}$$

INFER-BOX

$$\frac{\Gamma, \langle \tau \rangle; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1 \qquad \Gamma, x : \alpha; \Xi \vDash_{F_2} M \Rightarrow Y \,!\, \tau' \mid C_2}{\Gamma;\Xi \vDash_{F_1,F_2,\alpha} \texttt{box}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M \Rightarrow Y \,!\, \tau' \mid C_1 \cup C_2 \cup \{[\tau]X \doteq \alpha\}}$$

INFER-UNBOX

$$\frac{V = x \qquad (F_1, X, \tau_1) = \textsc{LookupCtxTau}(\Gamma,\ \Xi,\ \mathrm{x}) \qquad \Gamma, y : \alpha; \Xi \vDash_{F_2} M \Rightarrow Z \,!\, \tau_2 \mid C}{\Gamma;\Xi \vDash_{F_1,F_2,\alpha} \texttt{unbox}\ \tau\ V\ \texttt{as}\ y\ \texttt{in}\ M \Rightarrow Z \,!\, \tau_2 \mid C \cup \{(X \doteq [\tau]\alpha), (\tau_1 \dot{\geq} \tau)\}}$$

Figure 13. Inference rules for $\lambda_{\mathsf{Mille}[\tau]}$ constructs inherited from λ_τ.

LookupCtxTau is defined as follows:

$$\textit{LookupCtxTau}(\Gamma, \Xi, x) = \begin{cases} (\emptyset, X, \tau_{\Gamma_2}) & \textit{if } \Gamma = \Gamma_1,\ x : X,\ \Gamma_2, \\ (F, X, 0) & \textit{if } \Xi = \Xi_1,\ x : \forall F.\ X,\ \Xi_2 \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Observe that τ_{Γ_2} denotes the sum of temporal grades in a non-polymorphic context, as described in Definition 4.1.

We now review each rule presented in Figure 13:

- INFER-OP defines type inference for invoking an algebraic operation with a continuation in computations with temporal effects. If the argument V can be

inferred to have type X, introducing the set of fresh variables F_1 and the set of constraints C_1, and the continuation body M can be inferred to have computation type $Y \,!\, \tau$ under the extended contexts $\Gamma, \langle \tau_{\mathrm{op}} \rangle, x : B_{\mathrm{op}}; \Xi$, introducing the set of fresh variables F_2 and the set of constraints C_2, then the term $\mathrm{op}\ V\ (x.\ M)$ is inferred to have computation type $Y \,!\, \tau_{\mathrm{op}} \oplus \tau$. The set of fresh variables is F_1, F_2, and the set of constraints is $C_1 \cup C_2 \cup \{\, X \doteq A_{\mathrm{op}} \,\}$. This constraint ensures that the operation's argument matches its parameter type A_{op}, while the temporal annotation $\tau_{\mathrm{op}} \oplus \tau$ reflects the composition of the operation's intrinsic temporal grade with the temporal grade of the continuation. Note that this rule uses B_{op} and τ_{op} directly instead of introducing fresh type variables or abstract temporal grades, because these are available in the set of all operations Ω.

- Infer-Handle defines type inference for effect handling with temporal composition. If the handled computation M can be inferred to have computation type $X \,!\, \tau_1$, introducing F_1 and C_1, and the continuation N can be inferred to have computation type $Y \,!\, \tau_2$ under the extended contexts $\Gamma, \langle t \rangle, x : \alpha; \Xi$, introducing F_2 and C_2, and if the handler H consists of clauses $(y\ .\ k\ .\ M_{\mathrm{op}})_{\mathrm{op} \in \Omega}$ such that for each $\mathrm{op} \in \Omega$ the clause body can be inferred under

$$\Gamma,\ z : A_{\mathrm{op}},\ k : [\tau_{\mathrm{op}}]\big(B_{\mathrm{op}} \to Z_{\mathrm{op}} \,!\, t_{\mathrm{op}}\big); \Xi$$

to have computation type $Z_{\mathrm{op}} \,!\, \tau'_{\mathrm{op}}$, introducing F_{op} and C_{op}, then

$$\texttt{handle}\ M\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N$$

is inferred to have computation type $Y \,!\, (t \oplus \tau_2)$. The set of fresh variables is $(F_{\mathrm{op}})_{\mathrm{op} \in \Omega}, (t_{\mathrm{op}})_{\mathrm{op} \in \Omega}, F_1, F_2, \alpha, t$, and the set of constraints is

$$\Big(\bigcup_{\mathrm{op} \in \Omega} \big(C_{\mathrm{op}} \cup \{\, Z_{\mathrm{op}} \doteq Y,\ \tau'_{\mathrm{op}} \doteq \tau_{\mathrm{op}} \oplus t_{\mathrm{op}} \,\} \big) \Big) \ \cup\ C_1 \ \cup\ C_2 \ \cup\ \{\, X \doteq \alpha,\ \tau_1 \doteq t \,\}.$$

These constraints ensure that:

1. the result type is uniform across all handler clauses ($Z_{\mathrm{op}} \doteq Y$);
2. the temporal grade of each clause composes the intrinsic operation temporal grade with the resumed continuation's temporal grade ($\tau'_{\mathrm{op}} \doteq \tau_{\mathrm{op}} \oplus t_{\mathrm{op}}$), enforcing temporal compositionality;
3. the value produced by M is bound to x with the correct type ($X \doteq \alpha$);

4. the temporal grade preceding the execution of N is equal to the temporal grade of the computation M ($\tau_1 \doteq t$), so the overall temporal grade is $t \oplus \tau_2$.

- Infer-Delay defines type inference for delaying a computation by a temporal amount. If M can be inferred to have computation type $X \,!\, \tau'$ under the extended contexts $\Gamma, \langle\tau\rangle; \Xi$, introducing the set of fresh variables F and the set of constraints C, then the term `delay` τ M is inferred to have computation type $X \,!\, \tau \oplus \tau'$. The set of fresh variables is F, and the set of constraints remains C. This reflects that the delay contributes a temporal grade τ which composes (via $\oplus$) with the temporal grade τ' of running M.

- Infer-Box defines type inference for introducing a temporally boxed value and binding it in a computation. If V can be inferred to have type X under $\Gamma, \langle\tau\rangle; \Xi$, introducing F_1 and C_1, and M can be inferred to have computation type $Y \,!\, \tau'$ under the extended contexts $\Gamma, x : \alpha; \Xi$, introducing F_2 and C_2, then the term `box` τ V `as` x `in` M is inferred to have computation type $Y \,!\, \tau'$. The set of fresh variables is F_1, F_2, α, and the set of constraints is $C_1 \cup C_2 \cup \{\, [\tau]X \doteq \alpha \,\}$. The constraint $[\tau]X \doteq \alpha$ ensures that the bound variable x receives the modal type obtained by boxing X with the temporal grade τ; the computation itself incurs no additional temporal grade beyond that of M (hence the resulting temporal grade remains τ').

- Infer-Unbox defines type inference for eliminating a temporally boxed value. It is known from Theorem 4.1, that can not be anything other value other than a variable ($V = x$). If the auxiliary lookup returns $(F_1, X, \tau_1) = \textsc{LookupCtxTau}(\Gamma, \Xi, x)$, and if the body M can be inferred to have computation type $Z \,!\, \tau_2$ under the extended contexts $\Gamma, y : \alpha; \Xi$, introducing the set of fresh variables F_2 and the set of constraints C_2, then the term `unbox` τ V `as` y `in` M is inferred to have computation type $Z \,!\, \tau_2$. The set of fresh variables is F_1, F_2, α, and the set of constraints is $C_1 \cup C_2 \cup \{\, X \doteq [\tau]\alpha,\ \tau_1 \dot{\geq} \tau \,\}$. Here LookupCtxTau supplies: either $(\emptyset, X, \tau_{\Gamma_2})$ when $x : X$ is found in $\Gamma = \Gamma_1, x : X, \Gamma_2$, or $(F, X, 0)$ when $x : \forall F.\, X$ is found in $\Xi = \Xi_1, x : \forall F.\, X, \Xi_2$ based on 4.2. These constraints ensure that:

  1. the bound variable y receives the appropriately boxed type (enforced by $X \doteq [\tau]\alpha$); and

2. the available temporal grade from the lookup (τ_1) suffices to unbox with temporal grade τ (enforced by $\tau_1 \dot{\geq} \tau$), i.e., unboxing is permitted only when enough temporal computation has occurred after the variable x was brought into scope.

Note that although handlers and algebraic operations are not yet available in the implementation of Temporal Millet, they are still included in the core calculus and included in the soundness proof as a basis for future work.

4.3.2 Unification

In this subsection, we present the extended constraint unification procedure of $\lambda_{\mathsf{Mille}[\tau]}$.

We begin by describing the necessary modifications to the supporting structures, including redefinitions where required. We then introduce crucial subprocedures, followed by a formal representation of the unification algorithm, which extends traditional type unification to handle temporal effects by solving constraints over temporal grades. Finally, we discuss key properties of the algorithm and its output.

We begin by extending the definition of a substitution, first described in Definition 3.1, to include temporal grades when substituting modal and function types.

Definition 4.3 (Substitution)**.** *A* substitution of type variables and abstract temporal grades *is a finite mapping σ from temporal variables to abstract temporal grades and from type variables to types. We write* $\mathrm{dom}(\sigma)$ *for the domain of σ, i.e., the set of type variables and abstract temporal grades that σ replaces. If α is a type variable and X is a type, then $\sigma = [\alpha \mapsto X]$ denotes the singleton substitution that replaces α with X. Similarly, if t is an abstract temporal grade, then $\sigma = [t \mapsto 5]$ denotes the singleton substitution that replaces t with the natural number* 5*.*

Substitutions can still be applied to types and composed. By the extended definition, they can also be applied directly to temporal grades τ.

Definition 4.4 (Substitution Application to Temporal Grades)**.** *Let σ be a substitution of type variables and temporal grades. The application of a substitution σ to an abstract*

temporal grade t, written $\sigma(t)$, is defined recursively as follows:

$$\sigma(t) = \begin{cases} \tau & \textit{if } t \mapsto \tau \in \sigma \\ t & \textit{otherwise} \end{cases}$$

$$\sigma(n) = n, \textit{where } n \in \mathbb{N}$$

$$\sigma(\tau \oplus \tau') = \sigma(\tau) \oplus \sigma(\tau')$$

The extended definition of substitution application to types, now mutually recursive with substitution application to temporal grades, is the following:

Definition 4.5 (Substitution Application to Types)**.** *Let σ be a substitution of type variables and temporal grades. The application of a substitution σ to a type X, written $\sigma(X)$, is defined recursively as follows:*

$$\sigma(\alpha) = \begin{cases} X & \textit{if } \alpha \mapsto X \in \sigma \\ \alpha & \textit{otherwise} \end{cases}$$

$$\sigma(b) = b, \textit{where } b \textit{ is a base type (e.g., } \mathsf{int}, \mathsf{bool}\textit{)}$$

$$\sigma(\mathit{unit}) = \mathit{unit}$$

$$\sigma(X \to Y \mathbin{!} \tau) = \sigma(X) \to \sigma(Y) \mathbin{!} \sigma(\tau)$$

$$\sigma(X + Y) = \sigma(X) + \sigma(Y)$$

$$\sigma(X \times Y) = \sigma(X) \times \sigma(Y)$$

$$\sigma([\tau]X) = [\sigma(\tau)]\sigma(X)$$

The definition of substitution composition remains unchanged from Definition 3.3, except that it now implicitly relies on the extended application definition in Definition 4.5.

Before getting into the unification algorithm of $\lambda_{\mathsf{Mille}[\tau]}$, we define the subprocedures Simplify, ParamsOf, CancelCommon, BuildTau, and Eval and review their respective use cases.

Algorithm 3 SIMPLIFY Normalize temporal grades

```
1: function SIMPLIFY(τ)
2:     match τ with
3:         case τ₁ ⊕ τ₂:
4:             τ₁' ← SIMPLIFY(τ₁)
5:             τ₂' ← SIMPLIFY(τ₂)
6:             match (τ₁', τ₂') with
7:                 case (τ₁' ∈ ℕ, τ₂' ∈ ℕ):
8:                     return τ₁' + τ₂'
9:                 case (0, t) or (t, 0):
10:                    return t
11:                default: return τ₁' ⊕ τ₂'
12:        default: return τ
```

SIMPLIFY takes a temporal grade τ as input and attempts to simplify any nested abstract addition nodes where concrete values are present. If the input is an abstract temporal grade t or a concrete natural number, it is directly returned. In the case of abstract addition, if either is 0, the other side is returned. If both sides are natural numbers, their sum is returned. Otherwise, an abstract addition of simplified temporal grades is returned. SIMPLIFY is called before the case analysis of constraints of the form $\tau \doteq \tau'$ and $\tau \dot{\geq} \tau'$ to normalize temporal grades.

Algorithm 4 PARAMSOF Flatten and sort an abstract temporal grade

```
1: function FLATTEN(τ)
2:     if τ = τ₁ ⊕ τ₂ then
3:         params₁ ← FLATTEN(τ₁)
4:         params₂ ← FLATTEN(τ₂)
5:         return params₁ ++ params₂          ▷ ++ denotes list concatenation
6:     else
7:         return [τ]

8: function PARAMSOF(τ)
9:     params ← FLATTEN(τ)
10:    return SORT(params)
```

ParamsOf takes a temporal grade τ as input and flattens all nested abstract addition nodes into a list of abstract temporal grades t and natural numbers. The list is then sorted by preferring abstract temporal grades t, which are sorted lexicographically, followed by natural numbers in ascending order. The sorting procedure is omitted for brevity.

It is used as a preprocessing step in the unification algorithm of $\lambda_{\mathsf{Mille}[\tau]}$, for cancelling common elements when solving constraints of the form $\tau_1 + \tau_2 \doteq \tau$ and $\tau \doteq \tau_1 + \tau_2$. For example, take a fully expanded constraint $5 + t_1 + t_2 \doteq t_3 + 5 + t_1 + t_4$. After the invocation of ParamsOf, the left-hand side is turned into $[t_1, t_2, 5]$ and the right-hand side is turned into $[t_1, t_3, t_4, 5]$.

Algorithm 5 CancelCommon Remove matching elements from two sorted lists

```
1:  function CancelCommon(left, right)
2:      return Aux(left, right, [], [])

3:  function Aux(l, r, acc_L, acc_R)
4:      if l = [] and r = [] then
5:          return (acc_L, acc_R)
6:      else if l = [] then
7:          return (acc_L, acc_R ++ r)          ▷ ++ denotes list concatenation
8:      else if r = [] then
9:          return (acc_L ++ l, acc_R)          ▷ ++ denotes list concatenation
10:     else
11:         (lhd :: ltl, rhd :: rtl) ← (l, r)
12:         if lhd = rhd then
13:             return Aux(ltl, rtl, acc_L, acc_R)
14:         else if lhd < rhd then
15:             return Aux(ltl, r, acc_L ++ [lhd], acc_R)
16:         else
17:             return Aux(l, rtl, acc_L, acc_R ++ [rhd])
```

CancelCommon removes common elements from the beginnings of two sorted lists of temporal grades (as produced by ParamsOf). It assumes that the lists are sorted using the same ordering (which prioritizes abstract temporal grades over concrete ones, and sorts both groups internally).

The recursive helper Aux performs a pairwise comparison of the two lists:

- If the head elements are equal, they are dropped from both lists.
- If the head of the left list is smaller, it is added to the left accumulator.
- If the head of the right list is smaller, it is added to the right accumulator.

Once one of the lists is exhausted, the remaining elements from the other list are appended to the corresponding accumulator. The result is a pair of lists where shared prefixes have been removed and only the differing components remain. Similarly to ParamsOf, it is used in the unification algorithm of $\lambda_{\mathsf{Mille}[\tau]}$ for cancelling common elements when solving constraints of the form $\tau_1 + \tau_2 \doteq \tau$ and $\tau \doteq \tau_1 + \tau_2$. Take for example the input:

$$left = [t_1, t_2, 5], right = [t_1, t_3, t_4, 5],$$

then after applying CancelCommon, we have:

$$left = [t_2], right = [t_3, t_4].$$

Algorithm 6 BuildTau Reconstruct a temporal grade from a flattened list

```
1: function BuildTau(params)
2:     if params = [] then
3:         return 0
4:     else
5:         acc ← head(params)
6:         for τ in tail(params) do
7:             acc ← acc ⊕ τ
8:         return acc
```

BuildTau reconstructs an abstract temporal grade τ from a flattened and sorted list of parameters produced by ParamsOf. Each element in the list represents either an abstract temporal grade t or a concrete temporal grade $\tau \in \mathbb{N}$.

- If the list is empty, the result is the constant 0.
- Otherwise, the elements are combined into a nested abstract addition expression using a left fold. The fold begins with the first element and incrementally adds the rest using the binary operator $\oplus$.

For example, given the input $[t_2]$, BUILDTAU returns t_2, and given the input $[t_3, t_4]$, it returns $t_3 \oplus t_4$, etc.

Algorithm 7 EVAL Evaluate a fully concrete temporal grade

```
1:  function EVAL(τ)
2:      match τ with
3:          case τ ∈ ℕ:
4:              return τ
5:          case t:
6:              raise error (UnknownValueInEval)
7:          case τ₁ ⊕ τ₂:
8:              τ₁′ ← EVAL(τ₁)
9:              τ₂′ ← EVAL(τ₂)
10:             return τ₁′ + τ₂′
```

EVAL evaluates a temporal grade τ into a concrete numeric value. It is used for solving constraints of the form $\tau \dot{\geq} \tau'$ after normalization to determine whether the constraint is satisfied. If either side contains abstract parameters at the point of evaluation, an exception is raised and the evaluation is deferred. The complete evaluation strategy is as follows:

- If τ is a natural number, the value is returned directly.
- If τ is an abstract temporal grade t, an exception is raised, as EVAL requires that all inputs are fully simplified and concrete.
- If τ is an abstract addition $\tau_1 \oplus \tau_2$, both subterms are recursively evaluated and their values are added.

We now define the unification algorithm used in $\lambda_{\mathsf{Mille}[\tau]}$, which solves a set of constraints involving both type variables and temporal grades. The algorithm takes as input:

- A natural number P, representing the size of the unsolved constraint set from the previous iteration.
- A constraint set U denoting the set of constraints known not to be solvable in the current pass of C.

- A constraint set C that the algorithm directly traverses, produced during type inference.

The purpose of P is to determine whether progress is being made by checking if $|U| < P$. If so, the algorithm continues to iterate over U. The algorithm returns a substitution σ over both type variables and temporal grades if the constraint set is successfully unified.

Algorithm 8 Recursive unification algorithm of $\lambda_{\mathsf{Mille}[\tau]}$

1: **function** UNIFY(P, U, C)
2: **if** $C = \emptyset$ **then**
3: **if** $|U| = 0$ **then**
4: **return** $\emptyset$
5: **else if** $|U| = P$ **then**
6: **fail**("`Could not solve remaining constraints U)`"
7: **else**
8: **return** UNIFY($|U|$, $\emptyset$, U)
9: **else**
10: Let $c \in C$
11: Let $C' = C \setminus \{c\}$
12: **if** $c = (\tau_1 \doteq \tau_2)$ **then**
13: Let $\tau_1' \leftarrow$ SIMPLIFY(τ_1)
14: Let $\tau_2' \leftarrow$ SIMPLIFY(τ_2)
15: **if** $\tau_1' = \tau_2'$ **then**
16: **return** UNIFY(P, U, C')
17: **else if** $\tau_1' = t$ and $t \notin \text{fv}(\tau_2')$ **then**
18: Let $C'' = C'$ with t replaced by τ_2' in all temporal grades
19: Let $\sigma' =$ UNIFY(P, U, C'')
20: **return** $\sigma' \cup \{t \mapsto \sigma'(\tau_2')\}$
21: **else if** $\tau_2' = t$ and $t \notin \text{fv}(\tau_1')$ **then**
22: Let $C'' = C'$ with t replaced by τ_1' in all temporal grades
23: Let $\sigma' =$ UNIFY(P, U, C'')
24: **return** $\sigma' \cup \{t \mapsto \sigma'(\tau_1')\}$
25: **else if** $\tau_1' = 0$ and $\tau_2' = \tau_3 \oplus \tau_4$ **or vice versa then**
26: Let $C'' \leftarrow \{\tau_3 \doteq 0, \tau_4 \doteq 0\} \cup C'$
27: **return** UNIFY(P, U, C'')

28: **else if** $\tau_1' = t \oplus t'$ **or vice versa then**

29: Let $L_1 \leftarrow \textsc{paramsOf}(\tau_1')$

30: Let $L_2 \leftarrow \textsc{paramsOf}(\tau_2')$

31: Let $(L_1', L_2') \leftarrow \textsc{cancelCommon}(L_1, L_2)$

32: Let $\tau_1'' \leftarrow \textsc{buildTau}(L_1')$

33: Let $\tau_2'' \leftarrow \textsc{buildTau}(L_2')$

34: **if** $\tau_1'' = \tau_1'$ and $\tau_2'' = \tau_2'$ **then**

35: **return** $\textsc{unify}(P, \{\tau_1'' \doteq \tau_2''\} \cup U, C')$

36: **else**

37: **return** $\textsc{unify}(P, U, \{\tau_1'' \doteq \tau_2''\} \cup C')$

38: **else**

39: **return** $\textsc{unify}(P, \{\tau_1' \doteq \tau_2'\} \cup U, C')$

40: **else if** $c = (\tau_1 \dot{\geq} \tau_2)$ **then**

41: Let $\tau_1' \leftarrow \textsc{Simplify}(\tau_1)$

42: Let $\tau_2' \leftarrow \textsc{Simplify}(\tau_2)$

43: **try**

44: Let $v_1 \leftarrow \textsc{Eval}(\tau_1')$

45: Let $v_2 \leftarrow \textsc{Eval}(\tau_2')$

46: **if not** $v_1 \geq v_2$ **then**

47: **fail**("`Cannot unify temporal values` $\tau_1' \dot{\geq} \tau_2'$")

48: **else**

49: **return** $\textsc{unify}(P, U, C')$

50: **catch** `UnknownValueInEval`:

51: **return** $\textsc{unify}(P, U \cup \{\tau_1' \dot{\geq} \tau_2'\}, C')$

52: **else if** $c = (X \doteq Y)$ **then**

53: **if** $X = Y$ **then**

54: **return** $\textsc{unify}(P, U, C')$

55: **else if** $X = X_1 + X_2$ and $Y = Y_1 + Y_2$ **then**

56: **return** $\textsc{unify}(P, U, \{(X_1 \doteq Y_1), (X_2 \doteq Y_2)\} \cup C')$

57: **else if** $X = X_1 \times X_2$ and $Y = Y_1 \times Y_2$ **then**

58: **return** $\textsc{unify}(P, U, \{(X_1 \doteq Y_1), (X_2 \doteq Y_2)\} \cup C')$

59: **else if** $X = X_1 \to X_2 \mathbin{!} \tau_1$ and $Y = Y_1 \to Y_2 \mathbin{!} \tau_2$ **then**

60: **return** $\textsc{unify}(P, U, \{(X_1 \doteq Y_1), (X_2 \doteq Y_2), (\tau_1 \doteq \tau_2)\} \cup C')$

61: **else if** $X = \alpha$ and $\alpha \notin fv(Y)$ **then**

62: Let $C'' = C'$ with α replaced by Y in all types

63: Let $\sigma' = \text{UNIFY}(P, U, C'')$

64: **return** $\sigma' \cup \{\alpha \mapsto \sigma'(Y)\}$

65: **else if** $Y = \alpha$ and $\alpha \notin fv(X)$ **then**

66: Let $C'' = C'$ with α replaced by X in all types

67: Let $\sigma' = \text{UNIFY}(P, U, C'')$

68: **return** $\sigma' \cup \{\alpha \mapsto \sigma'(X)\}$

69: **else if** $X = [\tau_1]X'$ and $Y = [\tau_2]Y'$ **then**

70: **return** $\text{UNIFY}(P, U, \{(X' \doteq Y'), (\tau_1 \doteq \tau_2)\} \cup C')$

71: **else**

72: **fail**("`Cannot unify` X `and` Y")

Algorithm 8 attempts to simplify and solve constraints of three different forms:

1. Temporal grade equality constraints ($\tau_1 \doteq \tau_2$).
2. Temporal grade greater than or equal to constraints ($\tau_1 \dot{\geq} \tau_2$).
3. Type equality constraints ($X \doteq Y$).

The subprocedures Simplify, ParamsOf, CancelCommon, BuildTau, and Eval are employed to normalize temporal grades, compare and cancel common terms, rebuild simplified sums, and evaluate constant temporal grades when possible.

The algorithm iteratively extracts a constraint $c \in C$ and attempts to solve it:

- If $C = \emptyset$, the algorithm inspects U. If U is empty, a unifier $\sigma = \emptyset$ is returned. If $|U| = P$, then no progress was made during the last pass, and unification fails. Otherwise, U becomes the new C, and the algorithm repeats with $P = |U|$.
- For constraints ($\tau_1 \doteq \tau_2$), both sides are simplified, and the algorithm tries to eliminate variables or constants, cancel common elements, or defer the constraint if no further simplification is possible.
- For ($\tau_1 \dot{\geq} \tau_2$), both sides are simplified and evaluated using Eval if possible. If both evaluate to concrete values, they are directly compared to enforce the ordering. If not, the constraint is deferred by adding it to U for a later iteration.

- For $(X \doteq Y)$, the algorithm decomposes complex type constructors (functions, products, sums, modal types) and recursively unifies their components. If a type variable α can be substituted safely (i.e., $\alpha \notin fv(Y)$), it is replaced, and the substitution is recorded. This case is very similar to Algorithm 2, the unification algorithm of $\lambda_{\mathsf{Millet}}$.

The output is a substitution σ mapping both type variables and temporal variables to their unified values if all constraints can be solved.

Termination is guaranteed because each recursive call reduces the size or complexity of the constraint set:

- Every successful simplification step either removes a constraint from C or replaces it with simpler constraints (e.g., decomposing $X_1 \times X_2 \doteq Y_1 \times Y_2$ into two smaller constraints).
- The measure P ensures that the algorithm does not enter an infinite loop of deferrals: if no progress is made on the unsolved set U (i.e., $|U| = P$), the algorithm fails rather than looping.
- Time-grade simplification with Simplify and CancelCommon reduces nested additions to normalized forms, and since each addition node is strictly finite, this process terminates.

Additionally, we can observe that the algorithm is sound, e.g. fails to produce a substitution σ for programs that are ill-typed or have conflicting constraints:

- For types X and Y with incompatible constructors (e.g., $X = X_1 \times X_2$, $Y = X' \to Y'$), no decomposition rule applies, and the algorithm fails.
- For temporal constraints $(\tau_1 \dot{\geq} \tau_2)$, if both sides simplify to concrete values and $v_1 \geq v_2$ does not hold, the algorithm raises a failure, signaling that the required ordering cannot be satisfied.
- The occurs check ($t \notin fv(\tau_2')$ or $\alpha \notin fv(Y)$) prevents unification of a variable with a structure that contains it, avoiding unsound infinite types or circular definitions of temporal grades.

- If a constraint cannot be simplified or resolved due to abstract parameters, it is deferred to U. If subsequent iterations cannot make progress (i.e., $|U| = P$), the algorithm concludes that the program's constraints are unsolvable and fails.

We now redefine *constraint satisfaction*, and *principal types* for values and computations of $\lambda_{\mathsf{Mille}[\tau]}$.

Definition 4.6 (Constraint Satisfaction)**.** *Let C be a set of constraints of the form given by Definition 4.3. A substitution σ* satisfies *C, written $\sigma \vDash C$, if for every constraint $(X \doteq Y), (\tau \doteq \tau'), (\tau \dot{\geq} \tau') \in C$, we have:*

$$\sigma(X) = \sigma(Y), \sigma(\tau) = \sigma(\tau'), \sigma(\tau) \geq \sigma(\tau'),$$

respectively, where equality is understood as syntactic equality of types or temporal grades after applying σ and $\geq$ denotes a greater than or equal requirement between temporal grades drawn from natural numbers.

Algorithm 8 produces a *principal unifier* when the input constraint set is unifiable, and fails otherwise. The principal unifier can then be applied to the type generated during inference to obtain a *principal type*.

Definition 4.7 (Principal Type of a Term)**.** *Let V be a value in $\lambda_{\mathsf{Mille}[\tau]}$. A type X is said to be a* principal type *of V if:*

- *V has type X in the declarative type system, and*
- *for every type Y such that V also has type Y, there exists a substitution σ such that $Y = \sigma(X)$.*

That is, X is the most general type assignable to V. All other valid types are its instances.

Let M be a computation in the $\lambda_{\mathsf{Mille}[\tau]}$. A type $X \,!\, \tau$ is said to be a principal type *of M if:*

- *M has type $X \,!\, \tau$ in the declarative type system, and*
- *for every type $Y \,!\, \tau'$ such that M also has type $Y \,!\, \tau'$, there exists a substitution σ such that $Y \,!\, \tau' = \sigma(X) \,!\, \sigma(\tau)$.*

That is, $X \,!\, \tau$ is the most general type assignable to M. All other valid types are its instances.

Similarly to $\lambda_{\mathsf{Millet}}$, in both the value and computation fragments of the language, principal types are computed in two phases:

1. Generate a type X or $X \,!\, \tau$ and a set of constraints C for the value or computation using the corresponding inference rules.
2. Compute a principal unifier σ of the constraints C.

The resulting principal type is $\sigma(X)$ or $\sigma(X) \,!\, \sigma(\tau)$ for values and computations, respectively.

4.4 Soundness of Type Inference

In this section, we follow the same strategy as in Section 3.5. First, we show that polymorphic variable usage is linearizable in the presence of temporal effects by stating and proving Theorem 4.2. Then we redefine substitution application to computations and contexts, that are now enriched with temporal grades, and define substitution application to handlers. Finally, we show that the type inference of $\lambda_{\mathsf{Mille}[\tau]}$ without polymorphism is sound by stating and proving Theorem 4.3 and conclude that the type inference of $\lambda_{\mathsf{Mille}[\tau]}$ is generally sound after stating proving Theorem 4.4.

Theorem 4.2 (Polymorphic variable usage is linearizable in the presence of temporal effects)**.** *For all inference trees for the judgment* $\Gamma; \Xi \vDash_{\mathsf{V}\ F} V \Rightarrow X \mid C$*, there exist* Γ', F', V' *such that* $\Gamma', \Gamma \vDash_{\mathsf{V}\ F'} V' \Rightarrow X \mid C$*, where* Γ' *denotes a context derived from* Ξ*. Similarly, for all inference trees for the judgment* $\Gamma; \Xi \vDash_{\mathsf{C}\ F} M \Rightarrow X \,!\, \tau \mid C$*, there exist* Γ', F', M' *such that* $\Gamma', \Gamma \vDash_{\mathsf{C}\ F'} M' \Rightarrow X \,!\, \tau \mid C$*, where* Γ' *denotes a context derived from* Ξ*.*

Proof. We define two mutually recursive functions

$$\begin{aligned}\textsc{vlinearize}: \quad & (\Gamma; \Xi \vDash_{\mathsf{V}\ F} V \Rightarrow X \mid C) \times \mathbb{N} \to \\ & (\Gamma' : \mathsf{Ctx}) \times (F' : \mathcal{P}(\mathsf{Vars})) \times (V' : \mathsf{Val}) \times (\Gamma', \Gamma \vDash_{\mathsf{V}\ F'} V' \Rightarrow X \mid C) \times \mathbb{N}\end{aligned}$$

$$\begin{aligned}\textsc{clinearize}: \quad & (\Gamma; \Xi \vDash_{\mathsf{C}\ F} M \Rightarrow X \,!\, \tau \mid C) \times \mathbb{N} \to \\ & (\Gamma' : \mathsf{Ctx}) \times (F' : \mathcal{P}(\mathsf{Vars})) \times (M' : \mathsf{Comp}) \times (\Gamma', \Gamma \vDash_{\mathsf{C}\ F'} M' \Rightarrow X \,!\, \tau \mid C) \times \mathbb{N}\end{aligned}$$

for linearizing inference trees on value types and computation types, respectively. The core of the proof is identical to that of Theorem 3.1, modulo the presence of temporal grade annotations in computation types. We revisit the cases from Theorem 3.1, extended with temporal grade annotations, and examine the inference rules introduced in Figure 13 case by case for completeness.

- For rules with multiple premises (e.g., Infer-App, Infer-Pair), we recursively traverse the first subderivation to obtain an updated derivation and index, then apply the updated index to the second subderivation. For example:

  Let d_1 be an inference tree for $\Gamma; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1$.
  Let d_2 be an inference tree for $\Gamma; \Xi \vDash_{F_2} W \Rightarrow Y \mid C_2$.
  Let $(\Gamma_1', F_1', V', d_1', i_1) = \textsc{vlinearize}(d_1, i)$.
  Let $(\Gamma_2', F_2', W', d_2', i_2) = \textsc{vlinearize}(d_2, i_1)$.
  Let $\Gamma' = \Gamma_1', \Gamma_2'$.
  Let $F' = F_1', F_2', \alpha, t$.

  Then

  $$\textsc{clinearize}\left(\left(\dfrac{\dfrac{d_1 \qquad d_2}{\Gamma; \Xi \vDash_{F_1, F_2, \alpha, t} V\ W \Rightarrow \alpha\ !\ t \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\ !\ t\}}\ \textsc{Infer-App}}\right), i\right) =$$

  $$\left(\Gamma', F', V'\ W', \left(\dfrac{\dfrac{d_1' \qquad d_2'}{\Gamma', \Gamma \vDash_{F'} V'\ W' \Rightarrow \alpha\ !\ t \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha\ !\ t\}}\ \textsc{Infer-App}}\right), i_2\right).$$

- For rules that involve variable bindings, (e.g Infer-Fun, Infer-Let), we notice that freshly introduced bound variables are not affected by linearization, as they are always non-polymorphic and therefore not affected by Ξ. For example:

  Let $\Gamma', \Gamma, x : \alpha \vDash_{F'} M' \Rightarrow X\ !\ \tau \mid C$ be the root of the linearized inference tree corresponding to $\Gamma, x : \alpha; \Xi \vDash_{F} M \Rightarrow X\ !\ \tau \mid C$, and i' the updated index that was returned from the linearization call of that tree, which was passed i.
  Let $F'' = F', \alpha$.

  Then $\textsc{vlinearize}\left(\left(\dfrac{\dfrac{\Gamma, x : \alpha; \Xi \vDash_{F} M \Rightarrow X\ !\ \tau \mid C}{\Gamma; \Xi \vDash_{F, \alpha} \texttt{fun}\ x \mapsto M \Rightarrow \alpha \to X\ !\ \tau \mid C}\ \textsc{Infer-Fun}}\right), i\right) =$

  $$\left(\Gamma', F'', \texttt{fun}\ x \mapsto M', \left(\dfrac{\dfrac{\Gamma', \Gamma, x : \alpha \vDash_{F'} M' \Rightarrow X\ !\ \tau \mid C}{\Gamma', \Gamma \vDash_{F''} \texttt{fun}\ x \mapsto M' \Rightarrow \alpha \to X\ !\ \tau \mid C}\ \textsc{Infer-Fun}}\right), i'\right).$$

- For Infer-Delay, we observe that the temporal assumption $\langle \tau \rangle$ remains unaffected by linearization. This is because variable availability in Ξ is independent of temporal

assumptions, and the context Γ', derived from Ξ, does not include any such assumptions. Since Γ' is prepended to Γ, the original temporal structure is preserved. Aside from this observation, the case proceeds analogously to Infer-Fun.

Let $\Gamma', \Gamma, \langle\tau\rangle \vDash_{F'} M' \Rightarrow X \mathbin{!} \tau' \mid C$ be the root of the linearized inference tree corresponding to $\Gamma, \langle\tau\rangle; \Xi \vDash_{F} M \Rightarrow X \mathbin{!} \tau' \mid C$, and i' the updated index that was returned from the linearization call of that tree, which was passed i.

Then clinearize $\left(\left(\dfrac{\Gamma, \langle\tau\rangle; \Xi \vDash_{F} M \Rightarrow X \mathbin{!} \tau' \mid C}{\Gamma; \Xi \vDash_{F} \texttt{delay}\ \tau\ M \Rightarrow X \mathbin{!} \tau \oplus \tau' \mid C}\ \text{Infer-Delay}\right), i\right) =$

$$\left(\Gamma', F', \texttt{delay}\ \tau\ M', \left(\frac{\Gamma', \Gamma, \langle\tau\rangle \vDash_{F'} M' \Rightarrow X \mathbin{!} \tau' \mid C}{\Gamma'; \Gamma \vDash_{F'} \texttt{delay}\ \tau\ M' \Rightarrow X \mathbin{!} \tau \oplus \tau' \mid C}\ \text{Infer-Delay}\right), i'\right).$$

- For Infer-Op, we observe that the temporal assumption $\langle\tau_{\text{op}}\rangle$ added in the second premise is not affected by linearization, for the same reasons discussed in the Infer-Delay case. Aside from this observation, the case proceeds analogously to Infer-App.

  Let d_1 be an inference tree for $\Gamma; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1$.
  Let d_2 be an inference tree for $\Gamma, \langle\tau_{\text{op}}\rangle, x : B_{\text{op}}; \Xi \vDash_{F_2} M \Rightarrow Y \mathbin{!} \tau \mid C_2$.
  Let $(\Gamma'_1, F'_1, V', d'_1, i_1) = \text{vlinearize}(d_1, i)$.
  Let $(\Gamma'_2, F'_2, M', d'_2, i_2) = \text{clinearize}(d_2, i_1)$.
  Let $\Gamma' = \Gamma'_1, \Gamma'_2$.
  Let $F' = F'_1, F'_2$.
  Let $C = C_1 \cup C_2 \cup \{X \doteq A_{\text{op}}\}$.

  Then clinearize $\left(\left(\dfrac{d_1 \qquad d_2}{\Gamma; \Xi \vDash_{F_1, F_2} \text{op}\ V\ (x.\ M) \Rightarrow Y \mathbin{!} \tau_{\text{op}} \oplus \tau \mid C}\ \text{Infer-Op}\right), i\right) =$

  $$\left(\Gamma', F', \text{op}\ V'\ (x.\ M'), \left(\frac{d'_1 \qquad d'_2}{\Gamma'; \Gamma \vDash_{F'} \text{op}\ V'\ (x.\ M') \Rightarrow Y \mathbin{!} \tau_{\text{op}} \oplus \tau \mid C}\ \text{Infer-Op}\right), i_2\right).$$

- For Infer-Box, we also observe a case analogous to Infer-App.

  Let d_1 be an inference tree for $\Gamma, \langle\tau\rangle; \Xi \vDash_{F_1} V \Rightarrow X \mid C_1$.

  Let d_2 be an inference tree for $\Gamma, x : \alpha; \Xi \vDash_{F_2} M \Rightarrow Y \,!\, \tau' \mid C_2$.

  Let $(\Gamma'_1, F'_1, V', d'_1, i_1) = \textsc{vlinearize}(d_1, i)$.

  Let $(\Gamma'_2, F'_2, M', d'_2, i_2) = \textsc{clinearize}(d_2, i_1)$.

  Let $\Gamma' = \Gamma'_1, \Gamma'_2$.

  Let $F' = F'_1, F'_2, \alpha$.

  Let $C = C_1 \cup C_2 \cup \{[\tau]X \doteq \alpha\}$.

  Then $\textsc{clinearize}\left(\left(\dfrac{\begin{array}{cc} d_1 & d_2 \end{array}}{\Gamma; \Xi \vDash_{F_1,F_2,\alpha} \texttt{box}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M \Rightarrow Y \,!\, \tau' \mid C}\ \textsc{Infer-Box}\right), i\right) =$

  $$\left(\Gamma', F', \texttt{box}\ \tau\ V'\ \texttt{as}\ x\ \texttt{in}\ M', \left(\frac{\begin{array}{cc} d'_1 & d'_2 \end{array}}{\Gamma'; \Gamma \vDash_{F'} \texttt{box}\ \tau\ V'\ \texttt{as}\ x\ \texttt{in}\ M' \Rightarrow Y \,!\, \tau' \mid C}\ \textsc{Infer-Box}\right), i_2\right).$$

- For Infer-Unbox, which is one of the most interesting cases, we first need to recall Definition 4.2 for the details of the operation LookupCtxTau. If $V = x$ and $x : \forall F.X \in \Xi$, then we proceed similarly to the Infer-PolyVar case presented in the proof of Theorem 3.1. In this case, we define clinearize as follows:

  Let $(F, X, 0) = \textsc{LookupCtxTau}(\Gamma, \Xi, x)$.

  Let $(x_i : X) = \textsc{Fresh}(i, x : \forall F.X)$,

  Let d_2 be an inference tree for $\Gamma, y : \alpha; \Xi \vDash_{F_2} M \Rightarrow Z \,!\, \tau_2 \mid C$.

  Let $(\Gamma'_2, F'_2, M', d'_2, i') = \textsc{clinearize}(d_2, i + 1)$.

  Let $\Gamma' = x_i : X, \Gamma'_2$.

  Let $F' = F'_2, \alpha$.

  Let $C' = C \cup \{(X \doteq [\tau]\alpha),\ (0 \dot{\geq} \tau)\}$.

  Let $M_{\texttt{unbox}} = \texttt{unbox}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M'$.

  Then

  $\textsc{clinearize}\left(\left(\dfrac{\begin{array}{ccc} V = x & (F, X, 0) & d_2 \end{array}}{\Gamma; \Xi \vDash_{F,F_2,\alpha} \texttt{unbox}\ \tau\ V\ \texttt{as}\ y\ \texttt{in}\ M \Rightarrow Z \,!\, \tau_2 \mid C'}\ \textsc{Infer-Unbox}\right), i\right) =$

$$\left(\Gamma', F', M_{\texttt{unbox}}, \left(\frac{V = x_i \qquad (\emptyset, X, 0) \qquad d_2'}{\Gamma'; \Gamma \vdash_{\mathfrak{C}\ F'} M_{\texttt{unbox}} \Rightarrow Z\ !\ \tau_2 \mid C'}\ \text{Infer-Unbox}\right), i'\right).$$

In the other case, if $x : X \in \Gamma$, we define CLINEARIZE as follows:

Let $(\emptyset, X, \tau_1) = \textsc{LookupCtxTau}(\Gamma, \Xi, x)$.

Let d_2 be an inference tree for $\Gamma, y : \alpha; \Xi \vdash_{\mathfrak{C}\ F_2} M \Rightarrow Z\ !\ \tau_2 \mid C$.

Let $(\Gamma_2', F_2', M', d_2', i') = \textsc{clinearize}(d_2, i)$.

Let $\Gamma' = \Gamma_2'$.

Let $F' = F_2', \alpha$.

Let $C' = C \cup \{(X \doteq [\tau]\alpha),\ (\tau_1 \dot{\geq} \tau)\}$.

Let $M_{\texttt{unbox}} = \texttt{unbox}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M'$.

Then

$$\textsc{clinearize}\left(\left(\frac{V = x \qquad (\emptyset, X, \tau_1) \qquad d_2}{\Gamma; \Xi \vdash_{\mathfrak{C}\ F_2, \alpha} \texttt{unbox}\ \tau\ V\ \texttt{as}\ y\ \texttt{in}\ M \Rightarrow Z\ !\ \tau_2 \mid C'}\ \text{Infer-Unbox}\right), i\right) =$$

$$\left(\Gamma', F', M_{\texttt{unbox}}, \left(\frac{V = x \qquad (\emptyset, X, \tau_1) \qquad d_2'}{\Gamma'; \Gamma \vdash_{\mathfrak{C}\ F'} M_{\texttt{unbox}} \Rightarrow Z\ !\ \tau_2 \mid C'}\ \text{Infer-Unbox}\right), i'\right).$$

We omit Infer-Handle for brevity, as it is clear that the temporal assumptions are unaffected by linearization and it also follows the same structure as Infer-Fun for variable bindings and Infer-App in general, modulo having n premises instead of 2. It follows that the linearization of polymorphic variable usage in the inference trees of $\lambda_{\mathsf{Mille}[\tau]}$ can be achieved by applying the functions VLINEARIZE and CLINEARIZE to relevant nodes. Therefore, polymorphic variable usage is linearizable in the presence of temporal effects. □

Note that Lemma 3.2 continues to hold without modification, since both the auxiliary function Fresh and the case for Infer-PolyVar remain unchanged in the proof of Theorem 4.2. The reader is referred to Theorem 3.1 for details.

The soundness theorem requires applying substitutions to values, computations, and handlers. We now define substitution application to values, computations, and handlers. The definitions are mutually recursive.

Definition 4.8 (Substitution Application to Values)**.** *Let σ be a substitution of type variables. The application of σ to a value V, written $\sigma_V(V)$, is defined recursively as follows:*

$$\begin{aligned}
\sigma_V(()) &= () \\
\sigma_V(f(V_1, \ldots, V_n)) &= f(\sigma_V(V_1), \ldots, \sigma_V(V_n)) \\
\sigma_V(x) &= x \\
\sigma_V(\texttt{inl}\ V) &= \texttt{inl}\ \sigma_V(V) \\
\sigma_V(\texttt{inr}\ V) &= \texttt{inr}\ \sigma_V(V) \\
\sigma_V((V_1, V_2)) &= (\sigma_V(V_1), \sigma_V(V_2)) \\
\sigma_V(\texttt{fun}\ x \mapsto M) &= \texttt{fun}\ x \mapsto \sigma_C(M) \\
\sigma_V(\texttt{rec fun}\ f\ x \mapsto M) &= \texttt{rec fun}\ f\ x \mapsto \sigma_C(M)
\end{aligned}$$

Notice that in the function cases, we invoke both substitution application on types, defined in Definition 4.5, and substitution application on computations, written σ_C, given by the following definition.

Definition 4.9 (Substitution application to Computations)**.** *Let σ be a substitution of type variables and temporal grades. The application of σ to a computation C, written $\sigma_C(C)$, is defined recursively as follows:*

$$\begin{aligned}
\sigma_C(V\ W) &= \sigma_V(V)\ \sigma_V(W) \\
\sigma_C(\texttt{return}\ V) &= \texttt{return}\ \sigma_V(V) \\
\sigma_C(\texttt{let}\ x = M\ \texttt{in}\ N) &= \texttt{let}\ x = \sigma_C(M)\ \texttt{in}\ \sigma_C(N) \\
\sigma_C(\texttt{match}\ V\ \texttt{with}\ \{(x, y) \mapsto M)\} &= \texttt{match}\ \sigma_V(V)\ \texttt{with}\{(x, y) \mapsto \sigma_C(M)\} \\
\sigma_C(\texttt{match}\ V\ \texttt{with}\ \{\texttt{inl}\ x \mapsto M \mid \texttt{inr}\ y \mapsto N\}) &= \texttt{match}\ \sigma_V(V)\ \texttt{with} \\
&\quad \{\texttt{inl}\ x \mapsto \sigma_C(M) \mid \texttt{inr}\ y \mapsto \sigma_C(N)\} \\
\sigma_C(\mathsf{op}\ V\ (x\ .\ M)) &= \mathsf{op}\ \sigma_V(V)\ (x\ .\ \sigma_C(M)) \\
\sigma_C(\texttt{delay}\ \tau\ M) &= \texttt{delay}\ \sigma(\tau)\ \sigma_C(M) \\
\sigma_C(\texttt{handle}\ M\ \texttt{with}\ H\ \texttt{to}\ z\ \texttt{in}\ N) &= \texttt{handle}\ \sigma_C(M)\ \texttt{with} \\
&\quad \sigma_H(H)\ \texttt{to}\ z\ \texttt{in}\ \sigma_C(N) \\
\sigma_C(\texttt{box}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M) &= \texttt{box}\ \sigma(\tau)\ \sigma_V(V)\ \texttt{as}\ x\ \texttt{in}\ \sigma_C(M) \\
\sigma_C(\texttt{unbox}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M) &= \texttt{unbox}\ \sigma(\tau)\ \sigma_V(V)\ \texttt{as}\ x\ \texttt{in}\ \sigma_C(M)
\end{aligned}$$

Definition 4.10 (Substitution application to Handlers)**.** *Let σ be a substitution of type variables and temporal grades. The application of σ to a handler H, written $\sigma_H(C)$, is defined recursively as follows:*

$$\sigma_H((x\ .\ k\ .\ M_{\mathsf{op}})_{\mathsf{op}\in\Omega}) = (x\ .\ k\ .\ \sigma_C(M_{\mathsf{op}})_{\mathsf{op}\in\Omega})$$

We also redefine substitution application to contexts.

Definition 4.11 (Substitution application to Contexts)**.** *Let σ be a substitution of type variables and temporal grades. The application of σ to a context $\Gamma = \{x_1 : X_1, \ldots \langle\tau\rangle, \ldots, x_n : X_n\}$, written σ_Γ, is defined as $\sigma_\Gamma(\Gamma) = \{x_1 : \sigma(X_1), \ldots \langle\sigma(\tau)\rangle, \ldots, x_n : \sigma(X_n)\}$.*

With these definitions in mind, we present the soundness results.

Theorem 4.3 (Soundness of type inference of $\lambda_{\mathsf{Mille}[\tau]}$ without polymorphism)**.** *Let $\Gamma; \Xi \vDash_{F} V \Rightarrow X \mid C$ be an inference derivation where $\Xi = \emptyset$. Suppose that $\sigma = \textsc{unify}(0, \emptyset, C)$ is a principal unifier for the constraint set C. Then it follows that:*

$$\sigma_\Gamma(\Gamma) \vDash \sigma_V(V) : \sigma(X)$$

in the declarative type system.

Analogously, if $\Gamma; \Xi \vDash_{F} M \Rightarrow X\ !\ \tau \mid C$ and $\sigma = \textsc{unify}(0, \emptyset, C)$, then:

$$\sigma_\Gamma(\Gamma) \vDash \sigma_C(M) : \sigma(X)\ !\ \sigma(\tau)$$

in the declarative type system.

In short, the provided program will successfully typecheck in the declarative type system and have the principal type $\sigma(X)$ if type inference and unification succeed and polymorphic variables are not used.

Proof. We proceed by induction on the structure of the given type inference derivation. For each rule, we construct a corresponding declarative typing derivation under the unified context by applying structural induction on the inference derivation, each time invoking the induction hypothesis on subderivations, applying the principal unifier σ, and reconstructing the corresponding declarative derivation under $\sigma_\Gamma(\Gamma)$. In the following we omit Ξ as it is assumed to be empty and therefore has no effect on the derivations.

Cases for the inference rules Infer-Unit, Infer-Const, Infer-Var, Infer-PolyVar, Infer-Inl, Infer-Inr and Infer-Pair are identical to those in Theorem 3.3. Infer-Fun and Infer-RecFun are analogous, modulo the temporal grade annotations in the computations, to which we also apply σ. Similarly to the proof of Theorem 3.3, Infer-Return is analogous to the case Infer-Fun, and Infer-Let, Infer-MatchPair and Infer-MatchSum are analogous to Infer-App.

We review the extended cases of Infer-Fun and Infer-App for clarity. The type inference rules newly introduced in $\lambda_{\mathsf{Mille}[\tau]}$ are then reviewed case by case, excluding Infer-Handle for brevity, as the full proof is very long and analogous to an extended combination of Infer-App and Infer-Fun.

- Infer-Fun: Suppose we have a derivation:

$$\textsc{Infer-Fun}\ \frac{\Gamma, x : \alpha \vDash_{\ F} M \Rightarrow X \,!\, \tau \mid C}{\Gamma \vDash_{\ F,\alpha} \texttt{fun}\ x \mapsto M \Rightarrow \alpha \to X \,!\, \tau \mid C}$$

  Let $\sigma = \textsc{unify}(0, \emptyset, C)$. By IH, exists σ such that $\sigma_\Gamma(\Gamma, x : \alpha) \vDash \sigma_C(M) : \sigma(X) \,!\, \sigma(\tau)$ in the declarative type system. Since σ unifies the constraint set, α is fresh and no constraints have been introduced to it in C, it is not in σ and remains unaltered. We omit explicit substitution application steps and conclude:

$$\textsc{Fun}\ \frac{\sigma_\Gamma(\Gamma, x : \alpha) \vDash \sigma_C(M) : \sigma(X) \,!\, \sigma(\tau)}{\sigma_\Gamma(\Gamma) \vDash \texttt{fun}\ x \mapsto \sigma_C(M) : \sigma(\alpha) \to \sigma(X) \,!\, \sigma(\tau)}$$

  in the declarative type system.

- Infer-App: Suppose we have a derivation:

$$\textsc{Infer-App}\ \frac{\Gamma \vDash_{\ F_1} V \Rightarrow X \mid C_1 \qquad \Gamma \vDash_{\ F_2} W \Rightarrow Y \mid C_2}{\Gamma \vDash_{\ F_1,F_2,\alpha,t} V\ W \Rightarrow \alpha \,!\, t \mid C_1 \cup C_2 \cup \{X \doteq Y \to \alpha \,!\, t\}}$$

  Let $\sigma = \textsc{unify}(0, \emptyset, C_1 \cup C_2 \cup \{X \doteq Y \to \alpha \,!\, t\})$.

  By IH, there exist substitutions $\sigma_1 = \textsc{unify}(0, \emptyset, C_1)$ and $\sigma_2 = \textsc{unify}(0, \emptyset, C_2)$ such that

$$\sigma_{1\,\Gamma}(\Gamma) \vDash \sigma_{1\,V}(V) : \sigma_1(X) \quad \text{and} \quad \sigma_{2\,\Gamma}(\Gamma) \vDash \sigma_{2\,V}(W) : \sigma_2(Y)$$

in the declarative type system. Since σ unifies the combined constraint set, and $C_1 \subseteq C$, there exists a substitution σ_1' such that $\sigma = \sigma_1' \circ \sigma_1$. Similarly for σ_2. Therefore, we can apply σ_1' and σ_2' to the above derivations, respectively, to obtain:

$$\sigma_\Gamma(\Gamma) \vdash_{\mathbb{V}} \sigma_V(V) : \sigma(X) \quad \text{and} \quad \sigma_\Gamma(\Gamma) \vdash_{\mathbb{V}} \sigma_V(W) : \sigma(Y)$$

From the constraint $X \doteq Y \to \alpha \,!\, t$, it follows that $\sigma(X) = \sigma(Y) \to \sigma(\alpha) \,!\, \sigma(t)$. Hence, using the declarative typing rule Apply, we can conclude:

$$\textsc{Apply}\ \frac{\sigma_\Gamma(\Gamma) \vdash_{\mathbb{V}} \sigma_V(V) : \sigma(Y) \to \sigma(\alpha) \,!\, \sigma(t) \qquad \sigma_\Gamma(\Gamma) \vdash_{\mathbb{V}} \sigma_V(W) : \sigma(Y)}{\sigma_\Gamma(\Gamma) \vdash_{\mathbb{E}} \sigma_V(V\ W) : \sigma(\alpha) \,!\, \sigma(t)}$$

in the declarative type system.

- Infer-Delay: Suppose we have a derivation:

$$\textsc{Infer-Delay}\ \frac{\Gamma, \langle\tau\rangle; \Xi \vdash_{F}\ M \Rightarrow X \,!\, \tau' \mid C}{\Gamma; \Xi \vdash_{F}\ \texttt{delay}\ \tau\ M \Rightarrow X \,!\, \tau \oplus \tau' \mid C}$$

Let $\sigma = \textsc{unify}(0, \emptyset, C)$. By IH, exists σ such that $\sigma_\Gamma(\Gamma, \langle\tau\rangle) \vdash_{\mathbb{E}} \sigma_C(M) : \sigma(X) \,!\, \sigma(\tau')$ in the declarative type system. Since σ unifies the constraint set and no new constraints have been introduced to C, we can omit the explicit substitution application steps and conclude:

$$\textsc{Delay}\ \frac{\sigma_\Gamma(\Gamma, \langle\tau\rangle) \vdash_{\mathbb{E}} \sigma_C(M) : \sigma(X) \,!\, \sigma(\tau')}{\sigma_\Gamma(\Gamma) \vdash_{\mathbb{E}} \texttt{delay}\ \sigma(\tau)\ \sigma_C(M) : \sigma(X) \,!\, \sigma(\tau) \oplus \sigma(\tau')}$$

in the declarative type system.

- Infer-Op: Suppose we have a derivation:

$$\textsc{Infer-Op}\ \frac{\Gamma; \Xi \vdash_{\mathbb{V}_{F_1}}\ V \Rightarrow X \mid C_1 \qquad \Gamma, \langle\tau_{\text{op}}\rangle, x : B_{\text{op}}; \Xi \vdash_{\mathbb{E}_{F_2}}\ M \Rightarrow Y \,!\, \tau \mid C_2}{\Gamma; \Xi \vdash_{\mathbb{E}_{F_1,F_2}}\ \text{op}\ V\ (x\ .\ M) \Rightarrow Y \,!\, \tau_{\text{op}} \oplus \tau \mid C_1 \cup C_2 \cup \{X \doteq A_{\text{op}}\}}$$

Let $\sigma = \textsc{unify}(0, \emptyset, C_1 \cup C_2 \cup \{X \doteq A_{\text{op}},\ B_{\text{op}} \doteq \alpha\})$.

By IH, there exist substitutions $\sigma_1 = \textsc{unify}(0, \emptyset, C_1)$ and $\sigma_2 = \textsc{unify}(0, \emptyset, C_2)$ such that

$$\sigma_1\, \Gamma(\Gamma) \vDash_{\mathsf{v}} \sigma_1\, V(V) : \sigma_1(X) \quad \text{and} \quad \sigma_2\, \Gamma(\Gamma, \langle \tau_{\text{op}} \rangle, x : \alpha) \vDash_{\mathsf{c}} \sigma_2\, M(M) : \sigma_2(Y) \,!\, \sigma_2(\tau)$$

in the declarative type system.

Since σ unifies the combined constraint set, and $C_1 \subseteq C$, there exists a substitution σ_1' such that $\sigma = \sigma_1' \circ \sigma_1$. Similarly for σ_2. Therefore, we can apply σ_1' and σ_2' to the above derivations, respectively, to obtain:

$$\sigma_\Gamma(\Gamma) \vDash_{\mathsf{v}} \sigma_V(V) : \sigma(X) \quad \text{and} \quad \sigma_\Gamma(\Gamma, \langle \tau_{\text{op}} \rangle, x : \alpha) \vDash_{\mathsf{c}} \sigma_C(M) : \sigma(Y) \,!\, \sigma(\tau)$$

From the constraints $X \doteq A_{\text{op}}$ and $B_{\text{op}} \doteq \alpha$, we obtain $\sigma(X) = \sigma(A_{\text{op}})$ and $\sigma(\alpha) = \sigma(B_{\text{op}})$.

Hence, using the declarative typing rule Op, we can conclude:

$$\textsc{Op}\ \frac{\sigma_\Gamma(\Gamma) \vDash_{\mathsf{v}} \sigma_V(V) : \sigma(A_{\text{op}}) \qquad \sigma_\Gamma(\Gamma, \langle \tau_{\text{op}} \rangle, x : \sigma(B_{\text{op}})) \vDash_{\mathsf{c}} \sigma_C(M) : \sigma(Y) \,!\, \sigma(\tau)}{\sigma_\Gamma(\Gamma) \vDash_{\mathsf{c}} \text{op}\ \sigma_V(V)\ (x\ .\ \sigma_C(M)) : \sigma(Y) \,!\, \sigma(\tau_{\text{op}}) \oplus \sigma(\tau)}$$

in the declarative type system.

- Infer-Box: Suppose we have a derivation:

$$\textsc{Infer-Box}\ \frac{\Gamma, \langle \tau \rangle; \Xi \vDash_{\mathsf{v}\ F_1} V \Rightarrow X \mid C_1 \qquad \Gamma, x : \alpha; \Xi \vDash_{\mathsf{c}\ F_2} M \Rightarrow Y \,!\, \tau' \mid C_2}{\Gamma; \Xi \vDash_{\mathsf{c}\ F_1, F_2, \alpha} \texttt{box}\ \tau\ V\ \texttt{as}\ x\ \texttt{in}\ M \Rightarrow Y \,!\, \tau' \mid C_1 \cup C_2 \cup \{[\tau]X \doteq \alpha\}}$$

Let $\sigma = \textsc{unify}(0, \emptyset, C_1 \cup C_2 \cup \{[\tau]X \doteq \alpha\})$.

By IH, there exist substitutions $\sigma_1 = \textsc{unify}(0, \emptyset, C_1)$ and $\sigma_2 = \textsc{unify}(0, \emptyset, C_2)$ such that:

$$\sigma_1\, \Gamma(\Gamma, \langle \tau \rangle) \vDash_{\mathsf{v}} \sigma_1\, V(V) : \sigma_1(X) \quad \text{and} \quad \sigma_2\, \Gamma(\Gamma, x : \alpha) \vDash_{\mathsf{c}} \sigma_2\, M(M) : \sigma_2(Y) \,!\, \sigma_2(\tau')$$

in the declarative type system.

Since σ unifies the combined constraint set, and $C_1 \subseteq C$, there exists a substitution σ_1' such that $\sigma = \sigma_1' \circ \sigma_1$. Similarly for σ_2. Therefore, we can apply σ_1' and σ_2' to the above derivations, respectively, to obtain:

$$\sigma_\Gamma(\Gamma, \langle \tau \rangle) \vDash_{\mathsf{v}} \sigma_V(V) : \sigma(X) \quad \text{and} \quad \sigma_\Gamma(\Gamma, x : \alpha) \vDash_{\mathsf{c}} \sigma_C(M) : \sigma(Y) \,!\, \sigma(\tau')$$

From the constraint $[\tau]X \doteq \alpha$, it follows that $[\sigma(\tau)]\sigma(X) = \sigma(\alpha)$. Hence, using the declarative typing rule Box, we can conclude:

$$\textsc{Box}\quad \frac{\sigma_\Gamma(\Gamma, \langle\tau\rangle) \vDash_{\mathbb{V}} \sigma_V(V) : \sigma(X) \qquad \sigma_\Gamma(\Gamma, x : \alpha) \vDash_{\mathbb{T}} \sigma_C(M) : \sigma(Y) \,!\, \sigma(\tau')}{\sigma_\Gamma(\Gamma) \vDash_{\mathbb{T}} \texttt{box}\ \sigma(\tau)\ \sigma_V(V)\ \texttt{as}\ x\ \texttt{in}\ \sigma_C(M) : \sigma(Y) \,!\, \sigma(\tau')}$$

in the declarative type system.

- Infer-Unbox: Suppose we have a derivation:

$$\textsc{Infer-Unbox}\quad \frac{V = x \qquad (\emptyset, X, \tau_1) = \textsc{LookupCtxTau}(\Gamma,\ \Xi,\ x) \qquad \Gamma, y : \alpha; \Xi \vDash_{F_2} M \Rightarrow Z \,!\, \tau_2 \mid C}{\Gamma; \Xi \vDash_{F_2,\alpha} \texttt{unbox}\ \tau\ V\ \texttt{as}\ y\ \texttt{in}\ M \Rightarrow Z \,!\, \tau_2 \mid C \cup \{(X \doteq [\tau]\alpha), (\tau_1 \dot{\geq} \tau)\}}$$

Let $\sigma = \textsc{unify}(0, \emptyset, C \cup \{(X \doteq [\tau]\alpha),\ (\tau_1 \dot{\geq} \tau)\})$.

From the assumption $(\emptyset, X, \tau_1) = \textsc{LookupCtxTau}(\Gamma,\ \Xi,\ x)$, we have $x : X \in \Gamma$.

By IH, there exists a substitution $\sigma' = \textsc{unify}(0, \emptyset, C)$ such that

$$\sigma'_\Gamma(\Gamma, y : \alpha) \vDash_{\mathbb{T}} \sigma'_C(M) : \sigma'(Z) \,!\, \sigma'(\tau_2)$$

Since σ unifies the combined constraint set and $C \subset C \cup \{(X \doteq [\tau]\alpha), (\tau_1 \dot{\geq} \tau)\}$, there exists a substitution σ such that $\sigma = \sigma'' \circ \sigma'$. Therefore, we can apply σ'' to the above derivation, to obtain:

$$\sigma_\Gamma(\Gamma, y : \alpha) \vDash_{\mathbb{T}} \sigma_C(M) : \sigma(Z) \,!\, \sigma(\tau_2).$$

From $X \doteq [\tau]\alpha$ we obtain $\sigma(X) = [\sigma(\tau)]\,\sigma(\alpha)$. From $V = x$ and $\tau_1 \dot{\geq} \tau$ we derive

1. the side condition $\sigma(\tau) \leq \sigma(\tau_\Gamma)$, certifying that unboxing is temporally admissible in $\sigma_\Gamma(\Gamma)$, and
2. a derivation of $\sigma_\Gamma(\Gamma - \tau) \vDash_{\mathbb{V}} \sigma_V(V) : [\sigma(\tau)]\,\sigma(\alpha)$, by using the Var rule.

By construction, τ_1 is the sum of temporal grades occurring to the right of x in Γ, hence $\tau_1 \leq \tau_\Gamma$; together with $\tau_1 \dot{\geq} \tau$ this yields $\sigma(\tau) \leq \sigma(\tau_\Gamma)$, so the subtraction $\sigma_\Gamma(\Gamma - \tau)$ is defined in the declarative system and $x : [\sigma(\tau)]\,\sigma(\alpha) \in \sigma_\Gamma(\Gamma - \tau)$.

Hence, using the declarative typing rule Unbox, we can conclude:

UNBOX

$$\frac{\sigma(\tau) \leq \sigma(\tau_\Gamma) \qquad \sigma_\Gamma(\Gamma - \tau) \vdash_\forall \sigma_V(V) : [\sigma(\tau)]\,\sigma(\alpha) \qquad \sigma_\Gamma(\Gamma, y : \alpha) \vdash_\varepsilon \sigma_C(M) : \sigma(Z) \,!\, \sigma(\tau_2)}{\sigma_\Gamma(\Gamma) \vdash_\varepsilon \texttt{unbox}\ \sigma(\tau)\ \sigma_V(V)\ \texttt{as}\ y\ \texttt{in}\ \sigma_C(M) : \sigma(Z) \,!\, \sigma(\tau_2)}$$

in the declarative type system.

Therefore, the type inference of $\lambda_{\mathsf{Mille}[\tau]}$ is sound when polymorphic variables are not used. □

Finally, we present the general soundness theorem and proof for $\lambda_{\mathsf{Mille}[\tau]}$.

Theorem 4.4 (The type inference of $\lambda_{\mathsf{Mille}[\tau]}$ is sound)**.** *Let* $\Gamma; \Xi \vdash_{\forall\ F} V \Rightarrow X \mid C$ *be an inference derivation, with a possibly non-empty* Ξ*. Suppose that* $\sigma = \textsc{unify}(0, \emptyset, C)$ *is a principal unifier for the constraint set* C*. Then it follows that:*

$$\exists\, \Gamma', V' \,.\, \sigma_\Gamma(\Gamma', \Gamma) \vdash_\forall \sigma_V(V') : \sigma(X)$$

in the declarative type system, where Γ' *results from linearizing* Ξ *and* V' *differs from* V *only in the names of the linearized polymorphic variables from* Ξ*.*

Analogously, if $\Gamma; \Xi \vdash_{\varepsilon\ F} M \Rightarrow X \,!\, \tau \mid C$ *and* $\sigma = \textsc{unify}(0, \emptyset, C)$*, then:*

$$\exists\, \Gamma', M' \,.\, \sigma_\Gamma(\Gamma', \Gamma) \vdash_\varepsilon \sigma_C(M') : \sigma(X) \,!\, \sigma(\tau)$$

in the declarative type system, where Γ' *results from linearizing* Ξ *and* M' *differs from* M *only in the names of the linearized polymorphic variables from* Ξ*.*

Proof. The proof is analogous to the proof of Theorem 3.4. By Theorem 4.2, any inference tree can be linearized, and an analogue of Lemma 3.2 shows that this transformation preserves structure. In addition, Theorem 4.3 establishes the soundness of $\lambda_{\mathsf{Mille}[\tau]}$'s type inference in the absence of polymorphic variable usage.

Considering these results, it is clear that any inference tree for which a principal unifier can be constructed can be typed in the declarative type system. If the polymorphic variables are present, the inference tree can be linearized. Therefore, the type inference of $\lambda_{\mathsf{Mille}[\tau]}$ is sound. □

Before moving on to the Section 5, it is important to note that as there are no rules for typing polymorphic let expressions in $\lambda_{\mathsf{Mille}[\tau]}$, the soundness of type inference shown in

this section applies strictly to the bodies of `run` blocks in terms of the surface language Temporal Millet.

5. Implementation

In this section, we present an overview of the programming language Temporal Millet, developed as part of this thesis [9]. Temporal Millet was built on an existing implementation of a pure ML-like language Millet [8]. It serves as the surface language of $\lambda_{\mathsf{Mille}[\tau]}$. We begin by outlining its properties and features that are not directly tied to the core calculus, followed by discussing key implementation decisions, including temporal restrictions on recursive functions and the introduction of temporal abstractions in Section 5.1. We then showcase illustrative sample programs in Section 5.2, and conclude with a summary of known limitations in Section 5.3.

5.1 Overview

Temporal Millet inherits all the properties of Millet. Therefore, the overview of notable properties of Millet provided in Section 3.1 remains relevant and is not repeated here.

The surface language Temporal Millet and the core calculus $\lambda_{\mathsf{Mille}[\tau]}$ differ in several aspects. In particular, Temporal Millet allows type annotations for terms, implements product and sum types with an arbitrary number of arguments, and supports flexible pattern matching. All of these features are inherited from Millet and are not discussed in detail in this thesis, as they represent standard programming language features that are outside the scope of this work.

The implementation of Temporal Millet was developed in parallel with the formal description of the $\lambda_{\mathsf{Mille}[\tau]}$ core calculus. The implementation began with the type inference rules, starting from the simplest temporal operation, `delay`. From there, function types in Millet were extended with temporal grade annotations, which required extensive modifications across the codebase, from the abstract syntax tree definitions to the interpreter.

A key challenge was determining an appropriate structure for variable contexts with temporal grades. After experimentation, the most suitable design proved to be a list containing either variable maps equivalent to Millet's original standalone variable map or temporal grades. Implementing this structure required new functions for inserting and retrieving variables and temporal grades, as well as for computing the temporal grade of an entire context. As with the introduction of temporal annotations on function types, the use of the original variable map had to be refactored throughout the implementation.

With the revised variable map and `delay` implemented, temporal grades could be introduced into contexts, although they had no operational effect without the `box` and `unbox` constructs. Before implementing these, it became apparent that introducing fresh types also required introducing fresh temporal grades with unknown concrete values. This led to the introduction of abstract temporal grades and an associated addition operation, formally defined in Section 4.1. While this increased the complexity of constraint solving during unification, the alternative of relying on explicit annotations for each temporal grade was rejected. Such an approach would have trivialized type inference, shifted the burden of correctness to the programmer, and reduced the generality of top-level polymorphic functions.

The next step was to implement `box` and `unbox`. Initially, the `unbox` operation caused typing errors in almost all programs due to a naive subtraction strategy. The original approach was based directly on the declarative UNBOX rule in $\lambda_{\mathsf{Mille}[\tau]}$'s declarative type system (Section 4.1), where subtraction from the context is performed. However, in the presence of abstract temporal grades, this subtraction is not well-defined. During type inference, most temporal grades in the context are abstract, and the initial approach of interpreting them as 0 proved too aggressive: variables were frequently removed from the context, leading to widespread typing failures.

For a time, these limitations were accepted, as there was no straightforward way to defer subtraction to the unification phase. Even so, many programs continued to fail to type check in more complex scenarios. For example, constraints such as $0 + t_2 + t_4 = 5 + t_4$ could arise, which were unsolvable without additional processing. This motivated the introduction of auxiliary functions for simplifying abstract sums and cancelling common elements on both sides of a constraint, as described in Section 4.3.2.

The ability to simplify abstract temporal grades enabled more realistic examples, but programs remained limited to those where an explicit `delay` preceded every `unbox`. A key observation was that the value V passed to `unbox` can always be assumed to be a variable (see Theorem 4.1). This allowed determining the position of the variable in the context and constructing an abstract sum of all temporal grades occurring after it. A constraint could then be added requiring this sum to be greater than or equal to the temporal grade of the `unbox`, ensuring that sufficient temporal computation had occurred to make the operation valid. Unlike the subtraction step in the UNBOX rule of $\lambda_{\mathsf{Mille}[\tau]}$'s declarative

type system, this approach does not reconstruct a prior context; rather, it establishes that, once all constraints are solved, the corresponding contextual subtraction is admissible. This refinement was the final significant step toward supporting more general examples and enabled prototyping realistic use cases without resorting to artificially placed `delay` operations.

5.2 Sample Programs

To illustrate the capabilities of Temporal Millet in modelling temporal resources, this section presents a series of sample programs. These range from minimal examples, intended to demonstrate basic temporal constructs, to a prototype modelling the 3D printing scenario introduced in Section 1.

The most basic temporally relevant program consists of boxing a value, delaying the continuation sufficiently to make the boxed value eligible for unboxing, and then unboxing and returning it without modification. Such a program is showcased below.

```
run
    box 5 "test" as boxed in
    delay 5
    unbox 5 boxed as value in
    value
```

A slightly more complex example demonstrates the use of custom types to model the painting of a car part. Initially, the part is in a plain state. To simulate an algebraic operation such as painting, a top-level function is defined that accepts a part, delays for 20 temporal grades to represent the painting process, and returns a boxed painted part. The boxed value cannot be unboxed until the drying period, represented by an additional delay of 15 temporal grades, has elapsed. We present the program below.

```
type part = Part of string
type painted = Painted of part

let paint part =
    delay 20
    box 15 (Painted part) as paintedPart in
    paintedPart
```

```
run
    let rightDoor = Part "Right Door" in
    let leftDoor = Part "Left Door" in
    let paintedRightDoor = paint rightDoor in
    let paintedLeftDoor = paint leftDoor in
    unbox 15 paintedRightDoor as dryRightDoor in
    (paintedLeftDoor, dryRightDoor)
```

This program serves as a minimal prototype illustrating Temporal Millet's ability to enforce temporal correctness via the type system, while allowing temporal computations other than explicit delays to occur before unboxing. Here, `paint leftDoor` incurs enough temporal cost for `paintedRightDoor` to dry; therefore, it is safe to unbox.

The final example follows the 3D printing scenario described in Section 1. In addition to enforcing temporal safety, this program models the sequential progression from a digital model, to a fresh print, to a cooled print, to a UV-cured print, and finally to a completed product. Each stage is represented by a distinct type, ensuring that only valid transitions between states are possible. The program below demonstrates printing a hammer and a sword.

```
type model = Model of string
type fresh = Fresh of model
type cooled = Cooled of fresh
type uvCured = UvCured of cooled
type complete = Complete of uvCured

(* Mock 3D printing operation with signature model -> [5]fresh # 7 *)
let printResinModel model =
            delay 7
            box 5 (Fresh model) as
            printingResult in printingResult

(* Mock UV curing operation with signature cooled -> uvCured # 10 *)
let uvCure cooledObject = delay 10 (UvCured cooledObject)

run
```

```
    let freshSword = printResinModel (Model "Sword") in
    let freshHammer = printResinModel (Model "Hammer") in
    unbox 5 freshSword as cooledSword in
    let curedSword = uvCure (Cooled cooledSword) in
    unbox 5 freshHammer as cooledHammer in
    let curedHammer = uvCure (Cooled cooledHammer) in
    (Complete curedSword, Complete curedHammer)
```

This program also illustrates one of the current limitations of the implementation. Ideally, the unboxing could be performed within the `uvCure` function, thereby producing cleaner and more concise code. However, this is not currently supported.

5.3 Known Limitations

Not all challenges encountered during the implementation were resolved. Two notable limitations remain:

1. Unbounded temporal accumulation in recursive functions with non-zero temporal grades.
2. Unboxing within functions that do not perform temporal operations prior to unboxing.

Recursive functions whose bodies incur non-zero temporal grades are currently untypable in Temporal Millet. This is because the temporal type of such functions would depend on their arguments, and Temporal Millet does not presently support dependent types. For example, consider the program in Figure 14.

```
run
    let rec aux n a b =
        if n = 0 then delay 2 a else delay 2 aux (n - 1) b (a + b)
    in
    let fib n =
        aux n 0 1
    in
    (fib 7) + (fib 8)
```

Figure 14. Trivial untypable program creating unbounded temporal accumulation.

In this program, the call to `fib` passes its argument to `aux`, which is invoked n times. Consequently, `fib 7` would have type `int -> int # 14`, whereas `fib 8` would have type `int -> int # 16`. The temporal signature of `fib` thus varies with its argument n, demonstrating the need for a mechanism akin to dependent types to accurately track temporal grades in recursive functions.

A second limitation concerns unboxing in functions that do not perform temporal operations before the unbox. This prevents writing programs such as the one in Figure 15.

```
run
    box 10 "test" as b in
    delay 10
    let unb v = unbox 10 v as ub in ub in
    unb b
```

Figure 15. Trivial untypable program using `unbox` in a function without temporal operations.

In Figure 15, the function body of `unb` brings the bound variable v into scope after the delay. Because no temporal operations are performed within the function body, the temporal grade of the context following v is unified with 0. This causes type checking to fail with the error "Typing error: Cannot unify temporal values $0 \geq 10$".

Finally, the current implementation lacks algebraic operations and handlers. As a result, all prototype programs and examples rely on mock functions that introduce temporal grades via explicit delays to simulate such operations. Implementing algebraic operations and handlers was beyond the scope of this thesis.

6. Future Work

As discussed in Section 5, the implementation of algebraic operations and effect handlers was beyond the scope of this thesis. Nevertheless, this work contributes the type inference rules Infer-Op and Infer-Handle, described in 4.3.1, as a formal foundation to support the eventual implementation of these features in future work. In addition to algebraic operations and effect handlers, addressing the issues related to temporal recursion and general unboxing in function bodies, as reviewed in Section 5.3, remains an important avenue for further investigation.

Beyond the limitations explicitly identified in this thesis, a natural next step is to investigate the completeness of $\lambda_{\mathsf{Mille}[\tau]}$. With the introduction of deferring subtraction to the unification phase, it is no longer immediately clear that $\lambda_{\mathsf{Mille}[\tau]}$ is incomplete. This question warrants dedicated research to establish whether completeness can be achieved.

The broader, long-term objective to which this thesis contributes is the development of a sound and complete compiler capable of translating programs into low-level bytecode while preserving temporal safety guarantees through the type system. In support of this goal, this thesis provides a prototype implementation consisting of a type checker and a basic interpreter. This prototype has exposed several important challenges that require further study, thereby informing the path toward a full-scale implementation.

7. Conclusion

The primary objective of this thesis was to design and implement a prototype programming language capable of modelling temporal resources. To facilitate implementation, the existing programming language Millet was adopted as the baseline. This decision proved advantageous in the long term, as Millet provides a richer and more mature infrastructure compared to a minimal implementation of the standard λ-calculus. Leveraging Millet not only accelerated development but also enabled the formalization of its core calculus $\lambda_{\mathsf{Millet}}$ and the provision of a structured overview of the language. These results can serve as a reference for future researchers seeking to extend Millet or adapt it to new domains.

The central theoretical contribution of this thesis is the formalization of the $\lambda_{\mathsf{Mille}[\tau]}$-calculus, together with the type inference rules and unification algorithm presented in Sections 4.3.1 and 4.3.2, respectively. These formal components directly informed the implementation of Temporal Millet on top of Millet and guided the resolution of complex challenges, such as the context subtraction issue discussed in Section 5. The implementation of Temporal Millet constitutes the principal practical contribution of this work.

Not all of the original objectives could be realised within the scope of the practical implementation, as discussed in Section 6. Nonetheless, the work conducted has revealed several important avenues for further research in the modelling of temporal resources and has established a robust theoretical and practical foundation for future developments in this area.

References

[1] Plotkin G. and Power J. Notions of Computation Determine Monads. Vol. 2303. Dec. 2001. DOI: 10.1007/3-540-45931-6_24.

[2] Plotkin G. D. and Pretnar M. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). DOI: 10.2168/lmcs-9(4:23)2013.

[3] Kammar O., Lindley S., and Oury N. Handlers in action. *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 145–158. DOI: 10.1145/2544174.2500590.

[4] Leijen D. Implementing Algebraic Effects in C. Nov. 2017, pp. 339–363. DOI: 10.1007/978-3-319-71237-6_17.

[5] Bauer A. and Pretnar M. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84.1 (Jan. 2015), pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001.

[6] Ahman D. When Programs Have to Watch Paint Dry. *Foundations of Software Science and Computation Structures*. Springer Nature Switzerland, 2023, pp. 1–23. DOI: 10.1007/978-3-031-30829-1_1.

[7] Ahman D. and Žajdela G. A Stateful Time-Aware Operational Semantics for Temporal Resources. *Proceedings of the 10th Workshop on Mathematically Structured Functional Programming (MSFP 2024)*. Ed. by Gibbons J. and (Favonia) K.-B. H. Electronic Proceedings in Theoretical Computer Science. To appear. Tallinn, Estonia, July 2024.

[8] Pretnar M. Millet: A Pure ML-like Functional Language. Accessed: 2025-05-07. 2025. https://github.com/matijapretnar/millet.

[9] Tavits J. Temporal Millet. Version v1. Aug. 2025. DOI: 10.5281/zenodo.16812073.

[10] OpenAI. GPT-4.1 (OpenAI) — large language model. Online model release via OpenAI API and ChatGPT platform. GPT-4.1 replaces GPT-4o and GPT-4.5; released April 14, 2025. Apr. 2025. https://openai.com/index/gpt-4-1/.

[11] Church A. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58.2 (1936), pp. 345–363.

[12] Barendregt H. P. The Lambda Calculus: Its Syntax and Semantics. Vol. 103. Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland, 1984.

[13] Barendregt H., Dekkers W., and Statman R. Lambda Calculus with Types. USA: Cambridge University Press, 2013.

[14] Plotkin G. Call-by-name, call-by-value and the -calculus. *Theoretical Computer Science* 1.2 (1975), pp. 125–159. DOI: https://doi.org/10.1016/0304-3975(75)90017-1.

[15] Levy P., Power J., and Thielecke H. Modelling environments in call-by-value programming languages. *Information and Computation* 185.2 (2003), pp. 182–210. DOI: https://doi.org/10.1016/S0890-5401(03)00088-9.

[16] Levy P. B. Call-by-Push-Value: A Functional/Imperative Synthesis. Vol. 2. Semantics Structures in Computation. Berlin, Heidelberg: Springer, 2004. DOI: 10.1007/978-3-540-24643-7.

[17] Pierce B. C. Types and Programming Languages. 1st. The MIT Press, 2002.

[18] Wadler P. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)* (1987), pp. 307–313.

[19] Milner R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: https://doi.org/10.1016/0022-0000(78)90014-4.

[20] Damas L. and Milner R. Principal type-schemes for functional programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212. DOI: 10.1145/582153.582176.

[21] Wright A. K. Simple imperative polymorphism. *Lisp and symbolic computation* 8.4 (1995), pp. 343–355.

[22] Lepigre R. A classical realizability model for a semantical value restriction. *European Symposium on Programming*. Springer. 2016, pp. 476–502.

[23] Pretnar M. Inferring Algebraic Effects. *Logical Methods in Computer Science* Volume 10, Issue 3 (Sept. 2014). DOI: 10.2168/lmcs-10(3:21)2014.

[24] Clouston R. Fitch-Style Modal Lambda Calculi. *CoRR* abs/1710.08326 (2017). arXiv: 1710.08326.

License

Non-exclusive license to reproduce the thesis and make the thesis public

I, Joosep Tavits,

1. grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis **Implementing Temporal Resources**, supervised by Danel Ahman and Vesal Vojdani;

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons license CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Joosep Tavits

12/08/2025