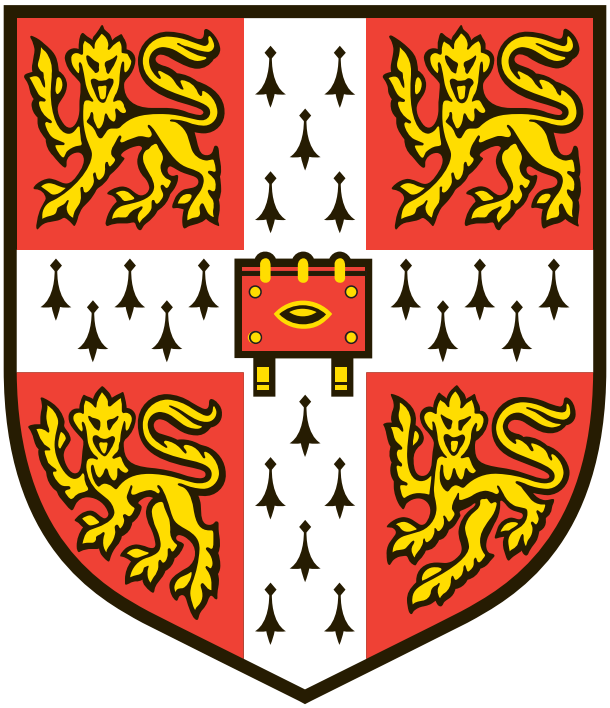
Alistair O'Brien

Typing OCaml in OCaml: A Constraint-Based Approach



Computer Science Tripos – Part II
Queens' College

May 13, 2022

Declaration

I, Alistair O'Brien of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Alistair O'Brien of Queens' College, am content for my dissertation to be made available to the students and staff of the University.

Signed: Alistair O'Brien
Date: May 13, 2022

Proforma

Candidate number:	**2377E**
Project Title:	**Typing OCaml in OCaml: A Constraint-Based Approach**
Examination:	**Computer Science Tripos – Part II, 2022**
Word Count:	**11999**[1]
Code Line Count:	**15888**[2]
Project Originator:	**The Dissertation Author**
Project Supervisor:	**Mistral Contrastin and Dr. Jeremy Yallop**

Original Aims of the Project

The project's original aim was to demonstrate the feasibility of a constraint-based type inference algorithm for a subset of OCaml, dubbed *Dromedary*. The subset would extend the ML calculus with *generalised algebraic data types*. Unlike OCaml's current inference algorithm, which has become challenging to maintain and evolve, Dromedary's constraint-based approach would prioritise modularity and correctness. The permissiveness and performance of Dromedary's inference algorithm would be evaluated against OCaml's current (4.12.0) implementation. Possible extensions included implementing side-effecting primitives, polymorphic variants, and semi-explicit first-class polymorphism.

Work Completed

Exceeded all success criteria and completed all extensions. Dromedary supports ML polymorphism, ADTs, patterns, records, side-effecting primitives, mutually recursive let-bindings and type definitions, GADTs, polymorphic variants, extensible variants, semi-explicit first-class polymorphism, type abbreviations, and structures. I formally defined Dromedary and its type system in a constraint-based setting. I developed a sufficiently expressive constraint language, with novel extensions on existing work. I implemented a modular and efficient constraint-based type inference algorithm for Dromedary, which is equally permissive and more performant in comparison to OCaml.

Special Difficulties

None.

[1] This word count was computed using `texcount`.

[2] This code line count was computed using `cloc` (excluding autogenerated test output).

Contents

List of Figures

List of Listings

1 Introduction

Since the late 1950s, many popular programming languages developed type systems and type checkers. Type checkers give the assurance of *type safety*: "well-typed programs cannot go wrong" [33], that is to say, a well-typed program is guaranteed not to violate any type system properties at runtime. One of the problems with many statically typed languages is that they require the programmer to annotate their programs with types. Type inference algorithms alleviate this issue by *inferring* the type annotations rather than requiring the programmer to provide them.

Type inference for functional programming languages such as Standard ML, Haskell and Objective Caml (OCaml) is based on the ML calculus defined by Milner [33], which provides *decidable type inference* for *let-based polymorphism*. Traditionally, inference algorithms for these languages are extensions of algorithms $\mathcal{W}$ or $\mathcal{J}$ [33], which use partial substitutions to reason about first-order equalities between types. However, these algorithms can become extremely complicated when extending the ML language with additional features.

The purpose of this dissertation is to investigate type inference algorithms using a constraint-based approach, specifically in the context of OCaml, to reduce the complexity introduced by these additional features.

1.1 OCaml

OCaml, introduced by Leroy [29], is a popular functional programming language with an advanced type system. The *core* language (referred to as Core ML) extends ML with the following features: mutually recursive let-bindings, algebraic data types, patterns, constants, records, mutable references (and the value restriction), exceptions and type annotations.

OCaml's major extensions on Core ML consist of first-class and recursive modules, classes and objects, polymorphic variants, semi-explicit first-class polymorphism, generalised algebraic data types (GADTs), the relaxed value restriction, type abbreviations, and labels.

OCaml's inference algorithm relies on an extension of algorithm $\mathcal{W}$ to efficiently deal with type generalisation, using a technique known as *rank-based generalisation* [41], with additional modifications for the above extensions.

However, it is widely accepted that OCaml's inference algorithm has become overly complex and difficult to maintain and evolve [52]. Constraint-based type inference proposes to solve these problems by separating type inference into three distinct phases: *constraint generation*, *solving*, and *type reconstruction* using a *small independent* first-order constraint language, making inference algorithms and theoretical proofs of correctness more modular.

The idea behind constraint-based type inference is elegantly simple: for some arbitrary term M, we generate a constraint C such that if C is true, then M is well-typed. After solving C, we construct $M_{:\tau}$, an explicitly-typed representation of the term M, during type reconstruction.

Figure 1.1: An overview of the constraint-based inference pipeline.

In this dissertation, we implement a type inference algorithm using a *constraint-based approach* for a subset of OCaml, dubbed *Dromedary*, consisting of Core ML with type annotations, polymorphic variants, extensible variants, semi-explicit first-class polymorphism, type abbreviations, structures, and GADTs.

1.2 Previous Work

Previous work to improve OCaml's inference algorithm has concentrated on incremental improvements to the existing implementation [52]. In contrast, our work is more ambitious and aims to provide the foundation for a complete rewrite – which we believe to be worthwhile.

Constraint-based inference for the ML type system has been extensively explored in Pierce's book [38]. Numerous extensions to OCaml's type systems have been independently formalised in a constraint-based setting [13, 18, 17]; however, we will provide the first unified work exploring constraint-based inference for OCaml's type system.

Many of the approaches described in [13, 18, 17] lack modularity due to the interleaving of constraint solving and type reconstruction. Our approach differs in that it builds upon Pottier's *modular constraint-based inference* [39].

1.3 Project Summary

I demonstrate the following in my dissertation:

- I define Dromedary and its type system in a constraint-based setting in Section 3.1.
- I implemented a constraint-based inference algorithm for *Dromedary*, focusing on modularity and efficiency (Section 3.2).
- I provide *empirical evidence* for the correctness of Dromedary's inference algorithm on a corpus of programs, and demonstrate that **Dromedary is equally permissive to OCaml**[1] (Section 4.2).
- In Section 4.3, I compare the performance of Dromedary's inference algorithm to OCaml's (4.12.0), demonstrating that **Dromedary is more performant than OCaml**.

[1] In the implemented extensions.

2 Preparation

In this chapter, we summarise the key background material for this project. In Section 2.1, we provide a detailed account of the existing theory for constraint-based type systems. Following that, we present a tutorial showcasing various features in OCaml that are implemented in Dromedary. Section 2.3 details the requirements of Dromedary's implementation and professional practices followed throughout the project.

2.1 Type Systems

Type systems are defined as a set of axioms and inductive rules (collectively known as *typing rules*) that constrain the form of an expression e according to its *type* τ – the expression `1 ^ "world"` is invalid since an `int` cannot be concatenated with a `string`.

In this section, we will explain the prerequisite theory and concepts required for formalising Dromedary's type system in Section 3.1, using the smaller and simpler ML calculus [33] for pedagogical purposes.

2.1.1 The ML Type System

The ML calculus [33] is presented in its implicitly-typed form, with *expressions* given by

$$e ::= x \mid c \mid \mathsf{fun}\ x \to e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e\,,$$

where x and c denote term variables and constants, respectively. The expression $\mathsf{fun}\ x \to e$ is an anonymous function that binds x in e, $e_1\ e_2$ is function application, and $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is a polymorphic let binding that binds e_1 to x in e_2.

Types and *type schemes* (or *polymorphic types*), denoted τ and σ, are defined by the following grammars:

$$\tau ::= \alpha \mid \overline{\tau}\ \mathsf{F}$$
$$\sigma ::= \forall\overline{\alpha}.\tau\,,$$

where α is a *type variable*, $\mathsf{F} ::= \cdot \to \cdot \mid \ldots$ is a *type former*[1] (or *type constructor*), and $\overline{\tau}$ represents a (possibly empty) vector of types of some finite length. Free variables $\mathsf{fv}(\cdot)$ and substitutions $\{\tau/\alpha\}(\cdot)$ on types and type schemes are defined as usual [33]. As a notational shorthand we write $\alpha \# \tau$ for $\alpha \notin \mathsf{fv}(\tau)$.

Typing judgements in the type system are of the form $\Gamma \vdash e : \tau$, read as: *the expression e has the type τ under the typing context Γ*. A *typing context* Γ (or typing environment) is a sequence of bindings of term variables x to type schemes σ, representing the current *scope*; free variables $\mathsf{fv}(\cdot)$ and substitutions are applied element-wise over Γ.

Typing rules are defined inductively, written as:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma_n \vdash e_n : \tau_n}{\Gamma \vdash e : \tau}\ (\mathsf{Rule\text{-}name})$$

Which is read as: *if the premises $\Gamma_i \vdash e_i : \tau_i$ listed above the bar hold for all $1 \le i \le n$, then the conclusion $\Gamma \vdash e : \tau$ below the bar holds.*

The ML typing rules are given in Figure 2.1, where Δ is an implicit typing context for *constants*. The two fundamental operations of the ML type system (implicit in the given presentation) are:

[1] The application of the former $\cdot \to \cdot$ is written in infix notation: $\tau_1 \to \tau_2$.

- **Generalisation**: The process of converting a *monomorphic* type τ (or *monotype*) into a type scheme σ, by binding the free variables of τ that are not present in Γ with a universal quantifier $\forall\overline{\alpha}.\tau$; this implicitly occurs in the ML-let rule.
- **Instantiation**: The process of specialising a type scheme σ into a type τ, by substituting the universally bound type variables $\overline{\alpha}$ with types $\overline{\tau}$. This implicitly occurs in the ML-var and ML-const rule.

$$\frac{x : \forall\overline{\alpha}.\tau \in \Gamma}{\Gamma \vdash x : \{\overline{\tau}/\overline{\alpha}\}\,\tau}\ (\textsf{ML-var}) \qquad \frac{c : \forall\overline{\alpha}.\tau \in \Delta}{\Gamma \vdash c : \{\overline{\tau}/\overline{\alpha}\}\,\tau}\ (\textsf{ML-const})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ (\textsf{ML-app}) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \textsf{fun}\ x \to e : \tau_1 \to \tau_2}\ (\textsf{ML-fun})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \overline{\alpha} = \textsf{fv}(\tau_1) \setminus \textsf{fv}(\Gamma) \qquad \Gamma, x : \forall\overline{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textsf{let}\ x = e_1\ \textsf{in}\ e_2 : \tau_2}\ (\textsf{ML-let})$$

Figure 2.1: The syntax-directed ML typing rules.

The ML typing rules read as follows:

- ML-var: If x has the type scheme $\sigma = \forall\overline{\alpha}.\tau$ in Γ, then x can be judged to have the type of an **instantiation** of σ – the monotype $\{\overline{\tau}/\overline{\alpha}\}\tau$. ML-const is analogous to ML-var.
- ML-app: If e_1 has the function type $\tau_1 \to \tau_2$, and the argument e_2 has the parameter type τ_1, then the application $e_1\ e_2$ may be judged to have the return type τ_2.
- ML-fun: Assuming x has the type τ_1, allowing us to conclude that function body e has the type τ_2, then we may deduce that the function fun $x \to e$ has the type $\tau_1 \to \tau_2$
- ML-let: Supposing e_1 has the type τ_1, and that we may **generalise** τ_1 to the type scheme $\forall\overline{\alpha}.\tau_1$, and that e_2 has the type τ_2 given that x has the type scheme $\forall\overline{\alpha}.\tau_1$, then we may conclude that the expression let $x = e_1$ in e_2 has the type τ_2.

Note that fun-bound variables in ML-fun are *monotypes*, thus polymorphism is only achieved via the use of let – as demonstrated below.

```
(* This is ill-typed - since [f] is monomorphic *)
fun f -> (f (fun x -> x)) (f 2)

(* This is well-typed - since [f] is polymorphic *)
let f = ...
in (f (fun x -> x)) (f 2)
```

Listing 2.1: ML let-based polymorphism in action – fun-bound variables are monomorphic, whereas let-bound variables are polymorphic.

Type inference is simply the process of finding a type τ for an expression e such that $\Gamma \vdash e : \tau$. Milner's essential insight for efficient type inference is that the typing rules are *syntax-directed*; that is to say, at most, one typing judgement applies to an expression. Consequently, the shape of the derivation tree for proving $\Gamma \vdash e : \tau$ is uniquely determined by the form of e.

So we can, in effect, run the typing rules "backwards" and "guess" types by introducing new type variables α; α is then subject to certain constraints induced by subsequent typing rules. Thus, type inference for ML decomposes into *constraint generation* followed by *constraint solving*.

Key Takeaway: ML *provides parametric polymorphism with efficient decidable type inference based on syntax-directed rules. Inference may be split up into constraint generation and constraint solving phases.*

2.1.2 Constraints

The interface between constraint generation and solving is the *constraint language*. The syntax of *constraints* and *constrained type schemes* [38] is given by the grammar:

$$\begin{aligned} C &::= \mathsf{true} \mid \mathsf{false} \mid \tau = \tau \mid C \wedge C \mid \exists \alpha.C \\ &\quad \mid \mathsf{def}\ x : \sigma\ \mathsf{in}\ C \mid x \leq \tau \mid \sigma \leq \tau\,, \end{aligned}$$

$$\sigma ::= \forall \overline{\alpha}.C \Rightarrow \tau\,.$$

Constraints naturally model a subset of first-order logic equipped with *equations* between types, consisting of *conjunction* and *existential quantification*.

Type inference for ML requires generalisation and instantiation. In order to permit these operations, we require the last three constraint constructs and constrained type schemes. The constraint $\mathsf{def}\ x : \sigma\ \mathsf{in}\ C$ associates the (constrained) type scheme σ with x in the constraint C (where x may appear as a free variable). The constraint $x \leq \tau$ (and $\sigma \leq \tau$) is an instantiation constraint, read as: τ *is an instance of* x which holds if type τ is an instance of the scheme σ associated with x. Constrained type schemes were introduced to avoid the interleaving of constraint generation and solving since determining the type scheme $\forall\overline{\alpha}.\tau$ requires constraint solving. We often write τ or $\forall\overline{\alpha}.\tau$ as *syntactic sugar* for $\forall \cdot .\mathsf{true} \Rightarrow \tau$ and $\forall\overline{\alpha}.\mathsf{true} \Rightarrow \tau$, respectively.

Constraints are a *formal logic*: a syntax with a semantic interpretation. Semantically, constraints are interpreted in the Herbrand universe, that is, the set of *ground types*:

$$\mathbf{t} ::= \overline{\mathbf{t}}\ \mathsf{F}\,.$$

A *ground assignment* φ is a partial function from type variables to ground types. Similarly, an *environment* ρ is a partial function from term variables x to *sets of ground types*. The interpretation of constraints is defined inductively in Figure 2.2, by the judgement $\varphi; \rho \Vdash C$, read as: *in the environment* ρ*, the ground assignment* φ *satisfies* C.

$$\frac{}{\varphi;\rho \Vdash \mathsf{true}} \qquad \frac{\varphi(\tau_1) = \varphi(\tau_2)}{\varphi;\rho \Vdash \tau_1 = \tau_2} \qquad \frac{\varphi;\rho \Vdash C_1 \qquad \varphi;\rho \Vdash C_2}{\varphi;\rho \Vdash C_1 \wedge C_2}$$

$$\frac{\varphi, \alpha \mapsto \mathbf{t};\rho \Vdash C}{\varphi;\rho \Vdash \exists\alpha.C} \qquad \frac{\varphi(\tau) \in \rho(x)}{\varphi;\rho \Vdash x \leq \tau} \qquad \frac{\varphi(\tau) \in (\varphi;\rho)(\sigma)}{\varphi;\rho \Vdash \sigma \leq \tau} \qquad \frac{\varphi;\rho, x \mapsto (\varphi;\rho)(\sigma) \Vdash C}{\varphi;\rho \Vdash \mathsf{def}\ x : \sigma\ \mathsf{in}\ C}$$

Figure 2.2: The inductive rules for semantic interpretation of constraints.

We interpret the constrained type scheme $\forall\overline{\alpha}.C \Rightarrow \tau$ under the assignment φ and environment ρ as the set of ground types $\varphi'(\tau)$ if the assignments φ and φ' are *equal modulo* $\overline{\alpha}$, denoted $\varphi =_{\setminus\overline{\alpha}} \varphi'$, and φ' satisfies C:

$$(\varphi;\rho)(\forall\overline{\alpha}.C \Rightarrow \tau) = \left\{\varphi'(\tau) : \varphi =_{\setminus\overline{\alpha}} \varphi' \wedge (\varphi';\rho \Vdash C)\right\}\,,$$

where assignments φ and φ' are said to be *equal modulo* $\overline{\alpha}$, if

$$\forall \beta \in (\mathsf{dom}(\varphi) \cap \mathsf{dom}(\varphi')) \setminus \overline{\alpha}.\ \varphi(\beta) = \varphi'(\beta)\,.$$

A constraint C_1 *entails* C_2, written $C_1 \vDash C_2$, if every context that satisfies C_1 also satisfies C_2. Similarly, equivalence $C_1 \simeq C_2$ holds if the property is bidirectional.

Entailment	$C_1 \vDash C_2$	$\forall \varphi, \rho.\ \ \varphi; \rho \Vdash C_1 \implies \varphi; \rho \Vdash C_2\,,$
Equivalence	$C_1 \simeq C_2$	$\forall \varphi, \rho.\ \ \varphi; \rho \Vdash C_1 \iff \varphi; \rho \Vdash C_2\,.$

As suggested in Section 2.1.1, we can now reduce the problem of type inference in ML to constraint solving by defining a mapping $[\![e : \tau]\!]$ of *candidate typings* to constraints, given in Figure 2.3.

$$\begin{aligned}
[\![x : \tau]\!] &= x \leq \tau \\
[\![c : \tau]\!] &= \Delta(c) \leq \tau \\
[\![\lambda x.e : \tau]\!] &= \exists \alpha_1 \alpha_2.\ \tau = \alpha_1 \to \alpha_2 \wedge \mathsf{def}\ x : \alpha_1\ \mathsf{in}\ [\![e : \alpha_2]\!] && \text{if } \alpha_1, \alpha_2 \,\#\, \tau \\
[\![e_1\ e_2 : \tau]\!] &= \exists \alpha.\ [\![e_1 : \alpha \to \tau]\!] \wedge [\![e_2 : \alpha]\!] && \text{if } \alpha \,\#\, \tau \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau]\!] &= (\exists \alpha.C) \wedge \mathsf{def}\ x : \forall \alpha.C \Rightarrow \alpha\ \mathsf{in}\ [\![e_2 : \tau]\!] \ \text{ where } C = [\![e_1 : \alpha]\!] && \text{if } \alpha \,\#\, \tau
\end{aligned}$$

Figure 2.3: The constraint generation mapping for ML – $[\![e : \tau]\!]$ is the constraint that holds if and only if e has the type τ.

A problem with the definition of constraint generation in Figure 2.3 is that the constraint $C = [\![e_1 : \alpha]\!]$ occurs twice in $[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau]\!]$, which can lead to exponential complexity. Fortunately, we can avoid this by extending the constraint language with the following construct:

$$C ::= \ldots \mid \mathsf{let}\ x : \sigma\ \mathsf{in}\ C,$$

semantically defined such that the following equivalence holds:

$$\mathsf{let}\ x : \forall \overline{\alpha}.C_1 \Rightarrow \tau\ \mathsf{in}\ C_2 \ \simeq\ \exists \overline{\alpha}.\ C_1 \wedge \mathsf{def}\ x : \forall \overline{\alpha}.C_1 \Rightarrow \tau\ \mathsf{in}\ C_2\,.$$

Linear constraint generation for $[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau]\!]$ is now given by

$$\begin{aligned}
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau]\!] &= \mathsf{let}\ x : (\!|e_1|\!)\ \ \mathsf{in}\ \ [\![e_2 : \tau]\!] \\
(\!|e|\!) &= \forall \alpha.[\![e : \alpha]\!] \Rightarrow \alpha\,,
\end{aligned}$$

where $(\!|e|\!)$ is a *principal constrainted type scheme* for e: it is the most general set of all ground types that e admits.

Key Takeaway: *Constraints form the interface between constraint generation and solving.* ML *constraint generation has linear complexity, making constraints suitable for an efficient and modular implementation of type inference. Extensibility of constraints demonstrates the ability to shift complexity between the constraint solving and generation phases.*

2.1.3 Constraint-Based ML : PCB

In [38], Rémy and Pottier present an alternative formalisation of the ML type system, dubbed PCB^2, whose distinctive feature is to exploit the constraints introduced in Section 2.1.2.

$$\frac{C \vDash x \leq \tau}{C \vdash x : \tau}\ (\textsf{PCB-var}) \qquad \frac{C_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad C_2 \vdash e_2 : \tau_1}{C_1 \wedge C_2 \vdash e_1\ e_2 : \tau_2}\ (\textsf{PCB-app})$$

$$\frac{C \vdash e : \tau_2}{\mathsf{def}\ x : \tau_1\ \mathsf{in}\ C \vdash \mathsf{fun}\ x \to e : \tau_1 \to \tau_2}\ (\textsf{PCB-fun})$$

$$\frac{C_1 \vdash e_1 : \tau_1 \qquad C_2 \vdash e_2 : \tau_2}{\mathsf{let}\ x : \forall \mathsf{fv}(C_1, \tau_1).C_1 \Rightarrow \tau_1\ \mathsf{in}\ C_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\textsf{PCB-let})$$

$$\frac{C \vdash e : \tau_1}{C \wedge \tau_1 = \tau_2 \vdash e : \tau_2}\ (\textsf{PCB-eq}) \qquad \frac{C \vdash e : \tau \qquad \alpha \# \tau}{\exists \alpha.C \vdash e : \tau}\ (\textsf{PCB-exists})$$

Figure 2.4: The PCB typing rules.

Judgements take the form $C \vdash e : \tau$, read as: *under the satisfiable assumptions C, the expression e has the type τ*, where the constraint C may contain free type *and* free term variables. We identify judgments modulo constraint equivalence of their assumptions, that is, $C \vdash e : \tau$ and $D \vdash e : \tau$ are equivalent when $C \simeq D$ holds. The type system is described in Figure 2.4.

The PCB-var rule states that x has the type τ under the assumption that τ is an instance of x. Unlike ML, no typing context is consulted. The environment is implicit within the constraint; thus, any free term (and type) variables in e also occur in C.

PCB-app is analogous to ML-app. PCB-fun requires the function body e to have the return type τ_2, under C. In the rule's conclusion, we wrap C in $\mathsf{def}\ x : \tau_1\ \mathsf{in}\ C$, assuming parameter x has type τ_1, permitting C to contain instantiation constraints of the form $x \leq \tau$. PCB-let is similar to PCB-fun, however, uses a let constraint to assign x a constrained type scheme; the free variables $\mathsf{fv}(C_1, \tau_1)$ in the quantifier ensure the scheme is closed. Both PCB-eq and PCB-exists are non-syntax directed rules required for *soundness*.

We are able to provide a more simplified formalisation by using PCB as the **basis of Dromedary's type system**. The inclusion of structured constraints in the typing rules benefits advanced features that rely on constraints, most notably GADTs (Section 3.1.6).

Additionally, metatheoretic properties such as soundness and completeness [38] of constraint-based inference for PCB may be stated directly without relying on substitutions, resulting in straightforward correctness proofs for Dromedary's inference[3].

Key Takeaway: PCB *is a purely constraint-based presentation of* ML*; its advantages include a simpler and more intuitive formalisation and easier correctness proofs for constraint-based inference, which Dromedary benefits from.*

2.2 OCaml

OCaml is a general-purpose, high-level programming language combining functional, object-oriented, and imperative paradigms – with one of the most sophisticated and powerful type inference algorithms available – based on the ML calculus from Section 2.1.1.

In this section, we describe several OCaml **design patterns** utilised throughout our codebase, as well as a selection of **sophisticated type system features** implemented in Dromedary.

[2] A **p**urely **c**onstraint-**b**ased type system.
[3] In future work.

OCaml extends ML with a wide range of features, including: mutual recursion, algebraic data types (ADTs), and patterns. Algebraic data types are defined using a combination of records and variants (so-called *product* and *sum* types). For example, Listing 2.2 defines the ADTs `bin_op`, `expr`, and `binding`.

```
type bin_op = Add | Sub

type expr =
  | Int of int
  (** Integer constant [1, -3, ...] *)
  | Var of string
  (** Variables [x, eval, ...] *)
  | Let of { bindings: binding list; in_: expr }
  (** Let bindings [let x1 = e1 and ... xn = en in e] *)
  | Bin_op of { left: expr; op: bin_op; right: expr }
  (** Infix binary operators [e1 + e2, e1 - e2] *)
and binding =
  { var: string
  ; exp: expr
  }
```

Listing 2.2: The type definitions for a simple language using algebraic data types in OCaml – `expr` and `bin_op` are *variant* types and `binding` is a *record* type.

Types, such as `bin_op`, can be constructed using *data constructors* (sometimes referred to as *variants* or *tags*) and deconstructed using *pattern matching*:

```
let eval_bin_op op n1 n2 =
  match op with
  | Add -> n1 + n2
  | Sub -> n1 - n2
```

As of OCaml 3, function arguments may be *labelled*, distinguished using a tilde ~ – permitting arguments to be specified by name instead of position (Listing 2.4). OCaml also supports an imperative style of programming, allowing side-effecting primitives such as *exceptions* and mutable *references*, as seen below in the imperative implementation of the factorial function:

```
let fact n =
  if n < 0 then raise Invalid_argument;
  let result = ref 1 in
  for i = 1 to n do
    result := i * !result
  done;
  !result
```

2.2.1 Modules and Functors

OCaml features a powerful module system [28] independent of the core language. Large OCaml programs are divided into modules, with each module consisting of a set of types and values.

Modules are defined by a *structure* and a *signature* (Listing 2.3). The *signature* defines the subset of types and values that can be used externally. Types may be specified *concretely* (exposing the underlying constructors) or *abstractly* (hiding the underlying implementation).

Values are specified using their type (*signature*). A *structure* is the implementation of a *signature* – implementing each type and value specified in the signature.

Components of a module are referred to through qualified identifiers (known as "dot notation") or using `open Module_name` for unqualified access (Listing 2.3).

Modules may be parameterised by other modules using *functors*[4], which are (informally) functions from modules to modules. In *Dromedary*, functors are foundational for implementing the modular constraints library.

```
open Core

module Env : sig
  (** Signature for [Env]. *)

  (** Abstract type [t] representing environment. *)
  type t

  (** [add t x n] adds the binding [(x, n)] to
      environment [t]. *)
  val add : t -> string -> int -> t

  (** [find_exn t x] returns the bound value of variable [x]
      in [t], raising [Not_found] if [x] is not in [t]. *)
  exception Not_found
  val find_exn : t -> string -> int
end = struct
  type t = (string, int) List.Assoc.t

  let add t x n =
    List.Assoc.add t x n ~equal:String.equal

  exception Not_found
  let find_exn t x =
    try List.Assoc.find_exn t x ~equal:String.equal
    with _ -> raise Not_found
end
```

Listing 2.3: A snippet demonstrating OCaml's module structures and signatures.

Key Takeaway: *Modules encapsulate and structure large OCaml programs. Functors are parameterised modules – used to decouple dependencies between modules, increasing modularity.*

2.2.2 Functors, Applicatives and Monads

Contrary to their cryptic nomenclature, functors, applicatives, and monads are simply functional programming *design patterns* analogous to object-oriented design patterns. They are based on the concept of composing various operations (or *effects*) to conceal complexity.

Intuitively, a *functor* is a polymorphic data structure `'a t` that 'wraps' values of type `'a`, with a function `map` which *lifts* a function `f` of type `'a -> 'b`, to a function on 'wrapped' values `'a t -> 'b t` (Listing 2.5). An *applicative*, or *applicative functor*, extends a functor by providing: (a) a function `return` that accepts any value and 'wraps' it; (b) an operation `both`, that takes two values `t1, t2` of types `'a t` and `'b t`, 'unwraps' *both* their values and

[4]Not to be confused with functors (Section 2.2.2).

```
let rec eval exp ~env =
  match exp with
  | Int n -> n
  | Var x ->
    Env.find_exn env x
  | Let { bindings; in_ } ->
    let env = bind ~env bindings in
    eval ~env in_
  | Bin_op { left; op; right } ->
    let n1 = eval ~env left
    and n2 = eval ~env right in
    eval_bin_op op n1 n2
and bind bindings ~env =
  (* Iterates over [bindings], adding each to [env] using
     [List.fold_right], returning the extended environment. *)
  List.fold_right bindings
    ~init:env
    ~f:(fun { var; exp } env ->
         Env.add env var (eval ~env exp))
```

Listing 2.4: An interpreter for the simple language from Listing 2.2. The implementation of `eval` uses mutual recursion and labelled arguments.

're-wraps' them into a pair, yielding a value of type `('a * 'b) t` – allowing *independent* operations to be sequenced. *Monads* extend applicative functors further, adding an operation `bind`, which permits the sequencing of *dependent* operations. Each structure and its operations must satisfy various laws known as the *functor, applicative, and monad laws* [55, 31] – which we omit.

Applicatives are a fundamental abstraction in our constraints library (Section 3.2.2). Additionally, we use monads extensively in our codebase as a generic design pattern for encapsulating side-effects, such as explicitly propagating failure using the `Result` monad.

Key Takeaway: *Functors, applicatives and monads are functional programming design patterns (similar to OOP design patterns) that are used to hide complexity by providing the ability to compose operations (or effects) on 'wrapped' values.*

```
module type Functor = sig
  type 'a t
  val map : 'a t -> f:('a -> 'b) -> 'b t
end

module type Applicative = sig
  include Functor
  val return : 'a -> 'a t
  val both : 'a t -> 'b t -> ('a * 'b) t
end

module type Monad = sig
  include Applicative
  val bind : 'a t -> f:('a -> 'b t) -> 'b t
end
```

Listing 2.5: The signatures for functors, appliactives, and monads.

2.2.3 Generalised Algebraic Data Types

Generalised algebraic data types (GADTs), introduced by Xi et al. [56], allow one to describe richer constraints between constructors and their types. The canonical example of GADTs is a typed *domain-specific language* (DSL):

```
(** ADT encoding of [expr] *)
type expr =
  | Int of int
  | Pair of expr * expr
  | Fst of expr
  | Snd of expr

(** GADT encoding of [expr] where ['t expr] is [expr]
    of type ['t].  *)
type _ expr =
  | Int : int -> int expr
  | Pair : 'a expr * 'b expr -> ('a * 'b) expr
  | Fst : ('a * 'b) expr -> 'a expr
  | Snd : ('a * 'b) expr -> 'b expr
```

Listing 2.6: The type definition of a simple DSL in OCaml using ADTs and GADTs.

The formal details of the GADT definition are explained in Section 3.1.6. The important point is that it allows us to a give a more precise type for each constructor:

- `Int n` has the type `int` in the DSL, thus its type is `int expr`.
- The constructor `Pair` produces a pair from two expressions of types `'a`, `'b`, thus its type is `('a * 'b) expr`.
- The constructor `Fst` projects the first element from a pair `'a * 'b`, thus its type is `'a expr`.

Thus, with the GADT encoding, expressions such as `Fst (Int 1)` in OCaml are *ill-typed*, avoiding an error-prone programming style. Other compelling applications of GADTs will be discussed throughout this dissertation (Sections 2.2.5, 3.2.2).

Problems with Inference One of the characteristic features of the ML type system is its ability to infer the *principal* (or *most general*) type for any well-typed expression.

Among the primary difficulties associated with inference in the presence of GADTs is the loss of principality. Sulzmann et al. [46] demonstrated that programs with GADTs frequently have more than one principal type. To illustrate this, we consider the following example:

```
type (_, _) eq = Refl : ('a, 'a) eq

let coerce eq x = match eq with Refl -> x
```

Listing 2.7: The definition of the equality GADT in OCaml – the type `('a, 'b) eq` encodes a "proof" that `'a` is equal to `'b`.

Informally, a value of type `('a, 'b) eq` is a "proof" of equality between the type `'a` and `'b`. Thus, when we pattern match on a term `eq` with type `('a, 'b) eq`, we obtain the constraint `'a = 'b`. Considering `coerce` in Listing 2.7, if `eq` has the type `('a, 'b) eq`

and `x` has the type `'a`, then it may also have the type `'b`. So one may deduce that `coerce` has the type `('a, 'b) eq -> 'a -> 'b`.

However, there are in fact three principal types for `coerce`:

- `('a, 'b) eq -> 'c -> 'c`
- `('a, 'b) eq -> 'a -> 'b`
- `('a, 'b) eq -> 'b -> 'a`

This poses various problems. To begin, principality is a central property for efficient type inference since it allows us to make *locally optimal* decisions. Second, should a program have more than one principal type, which should we infer? To circumvent this, we often *restrict* the type system or rely on explicit annotations.

As described above, deconstructing GADTs using *pattern matching* introduces local typing constraints. However, these constraints may result in differing branch types in a `match` expression. Reconciling these types is difficult.

Key Takeaway: *GADTs allow richer constraints between data constructors and their types. However, inference is notoriously difficult, suffering from a loss of principality, irreconcilable branch types and reliance on polymorphic recursion (Section 2.2.4).*

2.2.4 Polymorphic Recursion

In OCaml, unannotated recursive functions are *monomorphically recursive*, which refers to a recursive parametric polymorphic function where the type parameters are *monomorphic*[5] for each recursive occurrence. For example, we consider the following algebraic data type for a *perfect binary tree* [36]:

```
type 'a perfect_tree =
  | Leaf of 'a
  | Node of 'a * ('a * 'a) perfect_tree
```

Listing 2.8: A type definition for perfect trees in OCaml, taken from [36].

We now consider writing a function `length : 'a perfect_tree -> int` to determine the number of nodes in a perfect tree. If we were to write `length` naively, we would receive a type error: this is due to the monomorphically recursive occurrence of `length` with the type `'_a perfect_tree -> int`, which cannot be applied to `t` which has the type `('_a * '_a) perfect_tree`.

```
let rec length t =
  match t with
  | Leaf _ -> 1
  | Node (_, t) ->
      1 + 2 * (length t)
```

```
Error: This expression has type
  ('_a * '_a) perfect_tree
but an expression was expected of type
  '_a perfect_tree
```

One solution is to make `length` *polymorphically recursive*, where each recursive occurrence is a (non-trivial) instantiation of a polymorphic type scheme. Regrettably, inference for *polymorphic recursion* is undecidable [25, 20]. The Milner-Mycroft calculus [35] addresses this by using programmer-supplied type annotations to provide decidable type inference. See Figure 2.5 and Listing 2.9 for details.

[5] `'_a` is used to denote a *weak* or *monomorphic* type variable in OCaml.

Key Takeaway: *Polymorphic recursion refers to recursive functions where each recursive occurrence is a non-trivial instantiation of a type scheme. Notable type system features that rely on polymorphic recursion include GADTs, region-based memory management [54], and binding-time analysis [10].*

$$e ::= \ldots \mid \mathsf{let\ rec}\ x : \forall \overline{\alpha}.\tau = e\ \mathsf{in}\ e$$

$$\frac{\Gamma, x : \forall \overline{\alpha}.\tau_1 \vdash e_1 : \tau_1 \qquad \overline{\alpha} = \mathsf{fv}(\tau_1) \setminus \mathsf{fv}(\Gamma) \qquad \Gamma, x : \forall \overline{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let\ rec}\ x : \forall \overline{\alpha}.\tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\mathsf{ML\text{-}poly\text{-}rec})$$

Figure 2.5: The ML typing rules for polymorphic recursion from the Milner-Mycroft calculus [35].

```
let rec length : type a. a perfect_tree -> int =
  fun t -> match t with
  | Leaf _ -> 1
  | Node (_, t) -> 1 + 2 * (length t)
```

Listing 2.9: An example of polymorphic recursion in OCaml – requiring an explicit polymorphic annotation for decidable type inference.

2.2.5 Semi-Explicit First-Class Polymorphism

Numerous characteristics contribute to OCaml's success. Undoubtedly, ML-style polymorphism, with its decidable type inference, is a substantial benefit. However, there are some cases where one would like to have *first-class polymorphism*, as in System F.

To demonstrate this use case, we consider a *dependent associative list*. Informally, a dependent associative list is a list of key-value pairs where the type of the value depends on the key. In OCaml, such a data structure may be encoded using GADTs:

```
(** Type for keys in associative list. The parameter ['a] is
    the type of values that will be associated to the key. *)
type _ key =
  | Int : int -> string key
  (** Integers are mapped to strings *)
  | Float : float -> bool key
  (** Floats are mapped to bools *)

(** Type for elements in the associative list. Each
    consisting of a key-value pair with the correct
    types -- an ['value key] with a value ['value]. *)
type elem = Elem : 'value key * 'value -> elem

type t = elem list
```

Listing 2.10: The type definition of dependent associative list in OCaml using GADTs.

Now suppose we wish to write a function `map` that applies a function `f` to each `'a` value for each `'a key`. For instance, we could write:

```
let map_elem (Elem (key, val_)) ~f =
  Elem (key, f key val_)

let map t ~f = List.map t ~f:(map_elem ~f)
```

However, this is ill-typed in OCaml. As with polymorphic recursion, this is because `f` occurs monomorphically in `map_elem`. Since `f` must be instantiated with an *arbitrary* key type `'a`, `map_elem`'s *correct type* in System F would be

$$\texttt{map\_elem} : \texttt{elem} \to \texttt{f} : (\forall\alpha.\alpha\ \texttt{key} \to \alpha \to \alpha) \to \texttt{elem}\,.$$

Unfortunately, OCaml does not support this form of *higher-rank polymorphism*, and its inference is undecidable. In OCaml, we use *semi-explicit first-class polymorphism* [18] in record types to introduce these universally quantified types:

```
type elem_mapper = { f : 'a. 'a key -> 'a -> 'a }

let map_elem (Elem (key, val_)) ~f:mapper =
  Elem (key, mapper.f key val_)
```

Listing 2.11: A demonstration of semi-explicit first-class polymorphism in OCaml, encoding the polymorphic type $\forall\alpha.\alpha\ \texttt{key} \to \alpha \to \alpha$ in the `elem_mapper` type.

Key Takeaway: *Semi-explicit first-class polymorphism provides the ability to express higher-rank polymorphism, as in System* F*, using polymorphic records. As a result, we can express more programs that would otherwise be ill-typed in the Damas-Milner "sweet spot".*

2.2.6 Polymorphic Variants

Polymorphic variants are similar to variant types defined by algebraic data types. Syntactically, polymorphic variants are distinguished from variants with a leading backtick:

```
let nan = `Not_a_number
```

The primary characteristic that differentiates polymorphic variants from variant types is their ability to be utilised without an *explicit type declaration*[6]. The polymorphic variant `` `Not_a_number ``, for example, has the inferred type `[> `Not_a_number]`. The `>` symbol at the beginning of a polymorphic variant type indicates that the type is a *lower bound*. We can interpret `[> `Not_a_number ]` as a variant that *at least* contains the constructor `` `Not_a_number ``. Similarly, polymorphic variants may also have an *upper bound*. For instance:

```
let is_a_number t =
  match t with
  | `Not_a_number -> false
  | `Int _ -> true
```

has the inferred type: `[< `Int of int | `Not_a_number ] -> bool`, interpreted as a variant containing constructors `` `Int `` or `` `Not_a_number ``, or some subset. These upper and lower bounds form a *subtyping* relation on polymorphic variants, giving them their increased expressiveness compared to ordinary variants. Polymorphic variants may also have no subtyping, as shown below:

[6] Sometimes referred to as *anonymous variants*.

```
type number = [ `Int of int | `Float of float | `Not_a_number ]
```

One of the pragmatic uses of polymorphic variants is *extensible error types*. For example, in Listing 2.12, the `solver_error` type extends the `unifier_error` type.

While polymorphic variants appear to be a superset of ordinary variants, their inference is far more complex – potentially resulting in cyclic types. Furthermore, due to their more expressive typing rules, they are less likely to catch type-level bugs[7].

```
type unifier_error =
  [ `Cyclic_type of Type.t
  | `Cannot_unify of Type.t * Type.t
  ]

type solver_error =
  [ unifier_error | `Unbound_variable of string ]
```

Listing 2.12: Extensible error types using polymorphic variants in OCaml, taken from the project implementation.

Key Takeaway: *Polymorphic variants enhance the flexibility and modularity of ordinary variants by leveraging structural polymorphism (subtyping). However, their inference is far more complex, and they incur a runtime performance cost.*

2.3 Requirements Analysis

The analysis of requirements of the core project is given in the **Success Critera** of the Project Proposal (Appendix E). The primary goal of the project was to investigate a constraint-based approach to type inference for a **subset of OCaml with GADTs**.

Achieving this goal requires three components. First, the design of a minimal subset of OCaml with GADTs, dubbed *Dromedary*. The second is the design and implementation of a *modular constraints library*. Finally, the implementation of constraint generation and type reconstruction using the aforementioned constraints library.

After completing the primary deliverable, many additional language features were added as stretch requirements. I performed a MoSCoW[8] analysis of these features, shown in Table 2.1. Features were ranked by their relevance (records are more relevant than modules), implementation complexity (side-effecting primitives are easier to implement than semi-explicit first-class polymorphism), and feasibility within the given time frame. While these extensions are not essential for the project's success, their implementation demonstrates the suitability of a modular constraint-based approach for a larger subset of OCaml.

Following this analysis, I intended to implement **all** "should-have" and the following "could-have" extensions: polymorphic variants, semi-explicit first-class polymorphism, and side-effecting primitives. I had not planned to implement a lexer and a parser, but they proved crucial for effective unit testing. Additional unanticipated extensions included: type abbreviations, extensible variants, and structures; although implementing them required extra time, their inclusion led to improved abstractions across our codebase.

[7] In practice.

[8] **M**ust have, **S**hould have, **C**ould have, **W**on't have.

MoSCow Priority	Feature
Must Have	ML type system; GADTs; Constraint-based inference
Should Have	Records; Type annotations; Mutually recursive let-bindings
Could Have	Side-effecting primitives (references, exceptions); Polymorphic variants; Semi-explicit first-class polymorphism; Type abbreviations; Lexer and Parser; Extensible Variants
Won't Have	Objects; Modules

Table 2.1: A MoSCoW [5] analysis of features to include in the project.

2.3.1 Model of Software Development

The project was divided into various milestones, such as independent modules and language features that could be added iteratively to the primary deliverable. This design lends itself to the *waterfall model* of software development [45], where each new feature undergoes a complete waterfall development cycle – design, implement, integrate, and test.

Milestones were based on the MoSCoW analysis, prioritising the primary deliverable requirements, subsequently focusing on "should have" and "could have" features. Milestones were divided into **24 tasks**, with each task's development tracked using a Kanban board [19] on GitHub.

2.3.2 Tools Used

I chose OCaml version 4.12.0 as the implementation language since many phases of constraint-based inference involve traversing of trees of some form, and hence features such as pattern matching and algebraic data types were crucial. Additionally, side-effecting operations enable efficient implementations, the powerful module system facilitates the creation of robust modular interfaces, and the preprocessor extension (PPX) ecosystem supports the embedding of domain-specific languages such as a constraint language.

I used Dune [48] as the build system, integrating it with OPAM [53], OCaml's package manager. Packages are installed under a *local switch* (local environment) to avoid clashes with existing OCaml installations on the same machine. The OCamlFormat tool ensures our codebase uses a consistent style, promoting readability. Makefile targets are provided as convenient command aliases for installing, building, and testing the project from scratch.

Git was used for version control, with remotes backed up on GitHub. Development of milestones took place in branches, creating a total of **23 branches**. Several branches were added for the experimental design of features and implementations. Once I decided on a particular design/implementation, I completed its waterfall cycle, merging its branch into `main`. At the time of writing, the `main` branch has 242 commits.

A critical stage of each waterfall development cycle is testing. For property-based unit testing, I used Alcotest [50] and QCheck [8]. The remaining integration and unit tests in the suite were written using Jane Street's **expect tests** library; each test executes a fragment of OCaml code, captures the result, and compares it to the correct output through textual diff. Rather than writing the expected output manually, I inspected the captured output of the test case, either

accepting it as correct or rectifying a fault. This was particularly advantageous when writing tests involving large syntax trees generated by our inference algorithm. In total, I wrote **514 tests**.

I built Continuous Integration workflows that automate the execution of regression tests, coverage checks and formatters for each commit. Coverage was tracked using `Coveralls.io` [7] and Bisect [2], achieving a **75% test coverage**[9].

I used LexiFi's landmarks [49] library for profiling, allowing me to optimise Dromedary's constraint solver. In the evaluation, I used Core Bench [3] to micro-benchmark Dromedary and OCaml: running the type checkers multiple times, finding mean runtime and memory usage with accuracy bounds.

2.3.3 License

This project is intended as a proof-of-concept and foundation for implementing constraint-based type inference for OCaml. Therefore, the code was made publicly available on GitHub under the MIT licence [22] – permitting any person to use, copy, modify, and distribute the software.

2.4 Starting Point

I'm familiar with types, having studied *Semantics of Programming Languages*. I have no previous experience in type inference beyond ML's classical inference algorithms [33]. I have a basic understanding of constraint solving as a result of studying *Prolog* and *Logic and Proof*.

Before starting, I conducted research on OCaml's type system to investigate the project's feasibility. From *Foundations of Computer Science* and extra-circular study, I have practical experience writing OCaml programs. I have previously worked with the OCaml compiler codebase.

2.5 Summary

We introduced the ML calculus and its constraint-based counterpart PCB in Section 2.1, which will serve as the foundation of Dromedary's type system. We discussed various advanced type system features in OCaml that we implement in Dromedary and some design patterns that are prevalent in our implementation. We also defended decisions on features, tools, and dependencies used and our overall software engineering methodology.

[9] Many unchecked lines included interface and type definitions, resulting in a lower percentage of checked code.

3 Implementation

The implementation of Dromedary is described in two sections. The first (Section 3.1) covers the design of Dromedary, along with the description of its type system. The second (Section 3.2) explores the practical implementation of Dromedary's inference, with a particular emphasis on design decisions that result in efficient and modular constraint-based inference.

3.1 Dromedary

Dromedary is a subset of the OCaml language, supporting **all of Core ML**: ML polymorphism, algebraic data types, type annotations, and side-effecting primitives. Additionally, Dromedary includes **many of OCaml's advanced type system features**, namely type abbreviations, extensible variants, abstract types, polymorphic recursion, semi-explicit first-class polymorphism, GADTs, and polymorphic variants.

The untyped syntax of Dromedary is given using BNF in Appendix A. We have designed a type system for Dromedary, based on PCB's type system presented in Section 2.1.3. It is the **the first** unified formalisation of (a substantial subset of) OCaml's type system in a constraint-based setting.

This section discusses a selection of the aforementioned language features and their formalisation – referencing selected typing rules. The majority of features are *orthogonal* and will be discussed as independent extensions to the ML type system. For the mathematically inclined reader, Appendix C provides a complete formalisation of the type system.

3.1.1 Algebraic Data Types

Algebraic data types (ADT) are defined as an abstract type $\overline{\alpha}\ \mathsf{F}$, with an *isomorphism* to some finite *sum* or *product* type – which may be recursive. Let K and ℓ range over *data constructors* and *record labels*. Formally, an *algebraic data type definition* is given by either a *variant type* or a *record type*, whose respective (closed) forms are

$$\mathsf{type}\ \overline{\alpha}\ \mathsf{F} \cong \sum_{i=1}^{n} K_i\ [\mathsf{of}\ \tau_i] \qquad \text{and} \qquad \mathsf{type}\ \overline{\alpha}\ \mathsf{F} \cong \prod_{i=1}^{n} \ell_i : \tau_i\,,$$

where $[S]$ denotes that the syntactic element S is *optional*. A *structural environment* Ψ consists of a sequence of typing definitions. The (closed) type scheme assigned to K and ℓ in Ψ, which one may derive from the type definition of F, are written as:

$$\Psi \vdash K : \forall\overline{\alpha}.[\tau \to]\ \overline{\alpha}\ \mathsf{F}\,,$$
$$\Psi \vdash \ell : \forall\overline{\alpha}.\tau \to \overline{\alpha}\ \mathsf{F}\,.$$

For example, the algebraic data type α perfect_tree (Listing 2.8) has the following constructors:

$$\mathsf{Leaf} : \forall\alpha.\alpha \to \alpha\ \mathsf{perfect\_tree}\,,$$
$$\mathsf{Node} : \forall\alpha.\alpha \times (\alpha \times \alpha)\ \mathsf{perfect\_tree} \to \alpha\ \mathsf{perfect\_tree}\,.$$

We extend the constraint language with instantiation constraints for data constructors $K \leq \tau$ and labels $\ell \leq \tau$, semantically defined such that the following equivalences hold:

$$K \leq [\tau_1 \to]\ \tau_2 \ \simeq\ \exists\overline{\alpha}.\ [\tau_1 = \tau\wedge]\ \tau_2 = \overline{\alpha}\ \mathsf{F} \qquad \text{if } \Psi \vdash K : \forall\overline{\alpha}.[\tau \to]\ \overline{\alpha}\ \mathsf{F}$$
$$\ell \leq \tau_1 \to \tau_2 \ \simeq\ \exists\overline{\alpha}.\ \tau_1 = \tau \wedge \tau_2 = \overline{\alpha}\ \mathsf{F} \qquad \text{if } \Psi \vdash \ell : \forall\overline{\alpha}.\tau \to \overline{\alpha}\ \mathsf{F}\,,$$

where Ψ is implicit.

To support binding multiple variables at once for patterns, we need to introduce the notion of *constrained contexts* and *fragments*. A *fragment* Δ is a mapping between term variables and their types: $\Delta ::= \cdot \mid \Delta, x : \tau$. Fragments intuitively reflect the typing context introduced when a value is successfully matched against a pattern.

A *constrained context* $\Gamma ::= \forall \overline{\alpha}.C \Rightarrow \Delta$ specifies a mapping from term variables x to (constrained) type schemes $\forall \overline{\alpha}.C \Rightarrow \Delta(x)$. These are required for the efficient (linear) generation of constraints for let-bound pattern matching. We write Δ for the context of the form $\forall \cdot .\mathsf{true} \Rightarrow \Delta$. The constraint language is suitably extended with *constrained contexts*:

$$C ::= \ldots \mid \mathsf{def}\ \Gamma\ \mathsf{in}\ C \mid \mathsf{let}\ \Gamma\ \mathsf{in}\ C\,.$$

These multi-variadic bindings are semantically equivalent to nested def and let constraints:

$$\begin{aligned}\mathsf{def}\ \forall \overline{\alpha}.C_1 \Rightarrow \overline{x_i : \tau_i}\ \mathsf{in}\ C_2 \ &\simeq\ \mathsf{def}\ x_1 : \forall \overline{\alpha}.C_1 \Rightarrow \tau_1\ \mathsf{in}\ \ldots\ \mathsf{def}\ x_n : \forall \overline{\alpha}.C_1 \Rightarrow \tau_n\ \mathsf{in}\ C_2\,,\\ \mathsf{let}\ \forall \overline{\alpha}.C_1 \Rightarrow \overline{x_i : \tau_i}\ \mathsf{in}\ C_2 \ &\simeq\ \mathsf{let}\ x_1 : \forall \overline{\alpha}.C_1 \Rightarrow \tau_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_n : \forall \overline{\alpha}.C_1 \Rightarrow \tau_n\ \mathsf{in}\ C_2\,.\end{aligned}$$

The typing judgements for algebraic data types feature three judgements corresponding to *patterns*, *cases*, and *expressions*. Judgements for patterns and cases are of the form: $C \vdash p : \tau \rightsquigarrow \Delta$ and $C \vdash p \to e : \tau_1 \Rightarrow \tau_2$; interpreted as: *under the satisfiable assumptions C, the pattern p has the type τ, binding variables in fragment Δ* and *under the satisfiable assumptions C, the case $p \to e$ matches values of type τ_1 returning values of type τ_2*, respectively.

The typing rules in Figure 3.1 may be read as follows:

- Dromedary-pat-var: If the pattern x matches a value of type τ, then it binds x with type τ in the fragment.
- Dromedary-pat-construct: We check the constructor K instantiates to $\tau_1 \to \tau_2$, and the pattern p has the type τ_1, binding Δ, concluding that the pattern $K\ p$ has the type τ_2, binding Δ.
- Dromedary-case: We require the pattern p to have the type τ_1, binding Δ assuming C_1 holds. Then, we check whether the body of the case e has the return type τ_2, under assumptions C_2 – which is permitted to contain assumptions about the bound variables Δ from the pattern p due to the def constraint in the conclusion.

$$\frac{}{C \vdash x : \tau \rightsquigarrow x : \tau}\ (\text{Dromedary-pat-var})$$

$$\frac{C \vDash K \leq \tau_1 \to \tau_2 \qquad C \vdash p : \tau_1 \rightsquigarrow \Delta}{C \vdash K\ p : \tau_2 \rightsquigarrow \Delta}\ (\text{Dromedary-pat-construct})$$

$$\frac{C_1 \vdash p : \tau_1 \rightsquigarrow \Delta \qquad C_2 \vdash e : \tau_2}{C_1 \wedge \mathsf{def}\ \Delta\ \mathsf{in}\ C_2 \vdash p \to e : \tau_1 \Rightarrow \tau_2}\ (\text{Dromedary-case})$$

$$\frac{C_p \vdash p : \tau_1 \rightsquigarrow \Delta \qquad C_1 \vdash e_1 : \tau_1 \qquad C_2 \vdash e_2 : \tau_2}{\mathsf{let}\ \forall \mathsf{fv}(C_p, C_1, \Delta).C_p \wedge C_1 \Rightarrow \Delta\ \mathsf{in}\ C_2 \vdash \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 : \tau_2}\ (\text{Dromary-exp-let})$$

Figure 3.1: A selection of Dromedary's typing rules related to algebraic data types.

- Dromedary-exp-let: This is analogous to the PCB-let rule (Section 2.1.3). The novelty is the use of a *constrained context* and pattern judgement to determine the bound variables.

3.1.2 Annotations and Polymorphic Recursion

Dromedary allows programmers to annotate their expressions with a type:

$$e ::= \ldots \mid (e : \tau)\,.$$

The typing rule for expressions of this form is fairly obvious:

$$\frac{C \vdash e : \tau}{C \vdash (e : \tau) : \tau}\ \text{(Dromedary-exp-constraint)}$$

However, unlike OCaml, type variables α are not implicitly bound in Dromedary; to be used in an annotation, the type variables must be introduced. This design choice was chosen to ensure a more uniform and principled approach to annotations:

$$e ::= \ldots \mid \mathsf{forall}\ (\mathsf{type}\ \overline{\alpha}) \to e \mid \mathsf{exists}\ (\mathsf{type}\ \overline{\alpha}) \to e\,.$$

```
(* flexible variables *)
let succ (x : 'a) : 'a =
  x + 1

(* rigid variables *)
let id (type a) (x : a) : a =
  x
```

```
(* flexible variables *)
let succ = exists (type 'a) ->
    fun (x : 'a) : 'a -> x + 1

(* rigid variables *)
let id = forall (type 'a) ->
    fun (x : 'a) : 'a -> x
```

Listing 3.1: Examples of annotations in OCaml (on the left) and Dromedary (on the right), illustrating the differences in the introduction of bounded type variables in expressions.

Type variables are either bound *existentially* (*flexibly*) or *universally* (*rigidly*). If α is existentially bound, then the expressions $(\mathsf{fun}\ x \to x + 1 : \alpha \to \alpha)$ and $(\mathsf{fun}\ x \to x : \alpha \to \alpha)$ are well-typed, whereas if α was universally bound, only $(\mathsf{fun}\ x \to x : \alpha \to \alpha)$ is well-typed since $(\mathsf{fun}\ x \to x + 1 : \alpha \to \alpha)$ is only well-typed for $\alpha = \mathsf{int}$.

The typing rule for the existential form is straightforward, simply binding the variables $\overline{\alpha}$ using an existential quantifier in the conclusion:

$$\frac{C \vdash e : \tau \qquad \overline{\alpha} \# \tau}{\exists \overline{\alpha}.C \vdash \mathsf{exists}\ (\mathsf{type}\ \overline{\alpha}) \to e : \tau}\ \text{(Dromedary-exp-exists)}$$

The universal case is more difficult. In order to check that $\mathsf{forall}\ (\mathsf{type}\ \overline{\alpha}) \to e$ has the type τ, we must check that e has the type τ *for all instances of* $\overline{\alpha}$. To express this, we introduce *universal quantification* into the constraints language:

$$C ::= \ldots \mid \forall \alpha.C\,.$$

It is semantically defined as:

$$\frac{\forall \mathbf{t}.\ \ \varphi, \alpha \mapsto \mathbf{t}; \rho \Vdash C}{\varphi; \rho \Vdash \forall \alpha.C}$$

While universal quantification is sufficient for typing the forall construct, to permit *linear* complexity for type checking (and constraint generation) we extend *constrained contexts* (Section 3.1.1) with universally quantified variables $\overline{\alpha}$:

$$\Gamma ::= \forall \overline{\alpha}, \overline{\beta}.C \Rightarrow \Delta \,,$$

where let Γ in C is semantically defined by equivalence:

$$\mathsf{let}\ \forall \overline{\alpha}, \overline{\beta}.C_1 \Rightarrow \Delta\ \mathsf{in}\ C_2 \ \simeq\ \forall \overline{\alpha}.\exists \overline{\beta}.\ C_1 \wedge \mathsf{def}\ \forall \overline{\alpha}, \overline{\beta}.C_1 \Rightarrow \Delta\ \mathsf{in}\ C_2 \,.$$

The (linear) typing rule is given by:

$$\frac{C_1 \vdash e : \tau_1 \qquad C_2 \vDash x \leq \tau_2}{\mathsf{let}\ \forall \overline{\alpha}, \mathsf{fv}(\tau_1).C_1 \Rightarrow x : \tau_1\ \mathsf{in}\ C_2 \vdash \mathsf{forall}\ (\mathsf{type}\ \overline{\alpha}) \to e : \tau_2}\ \text{(Dromedary-exp-forall)}$$

The reader may note the instantiation $x \leq \tau_2$ in Dromedary-exp-forall; this is required as forall (type $\overline{\alpha}$) $\to$ e is not necessarily polymorphic, it may implicitly be instantiated to a monotype. To illustrate this, in OCaml, the following is well-typed:

```
let id_int (n : int) =
  (fun (type a) (x : a) : a -> x) n
```

as the expression `fun (type a) (x : a) : a -> x` is *instantiated* to `int -> int`.

With annotations formalised, we can now attack the problem of *polymorphic recursion* (Section 2.2.4). We begin by discussing recursion in the constraints language, extending it with recursive def and let forms:

$$C ::= \ldots \mid \mathsf{def\ rec}\ x : \pi\ \mathsf{in}\ C \mid \mathsf{let\ rec}\ x : \pi\ \mathsf{in}\ C \,,$$

where π denotes a recursive binding. For recursion in *Dromedary*, we require two kinds of recursive bindings[1]:

$$\pi ::= \forall \overline{\alpha}.C \Rightarrow \tau \mid \forall \overline{\alpha}.C \Leftarrow \tau \,.$$

Intuitively, the *checking* binding $\forall \overline{\alpha}.C \Leftarrow \tau$ *asserts* that the binding has the type scheme $\forall \overline{\alpha}.\tau$ in the recursive constraint C, whereas the *synthesising* (or *inferring*) binding $\forall \overline{\alpha}.C \Rightarrow \tau$ *assumes* that the binding has type τ in the recursive constraint C when synthesising the binding's type scheme.

Semantically, the interpretation (*the set of ground types*) of these bindings is defined as:

$$(\varphi; \rho)\,(x : \forall \overline{\alpha}.C \Rightarrow \tau) = \left\{\varphi'(\tau) : \varphi =_{\setminus \overline{\alpha}} \varphi' \wedge (\varphi'; \rho, x \mapsto \varphi'(\tau) \Vdash C)\right\} \,,$$

$$(\varphi; \rho)\,(x : \forall \overline{\alpha}.C \Leftarrow \tau) = \left\{\varphi'(\tau) : \varphi =_{\setminus \overline{\alpha}} \varphi' \wedge (\varphi'; \rho, x \mapsto \varphi'(\forall \overline{\alpha}.\tau) \Vdash C)\right\} \,.$$

The semantics for recursive def and let constraints are analogues to their non-recursive counterparts, using the above interpretations for π-bindings.

Our constraints give rise to the following typing rules for *monomorphic* and *polymorphic* recursion, using the inferred and checked bindings, respectively:

$$\frac{C_1 \vdash e_1 : \tau_1 \qquad C_2 \vdash e_2 : \tau_2}{\mathsf{let\ rec}\ x : \forall \mathsf{fv}(C_1, \tau_1).C_1 \Rightarrow \tau_1\ \mathsf{in}\ C_2 \vdash \mathsf{let\ rec}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \text{(Dromedary-exp-rec-mono)}$$

$$\frac{C_1 \vdash e_1 : \tau_1 \qquad C_2 \vdash e_2 : \tau_2}{\mathsf{let\ rec}\ x : \forall \overline{\alpha}.C_1 \Leftarrow \tau_1\ \mathsf{in}\ C_2 \vdash \mathsf{let\ rec}\ x : \forall \overline{\alpha}.\tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \text{(Dromedary-exp-rec-poly)}$$

[1] Notation inspired by bidirectional type checking [9].

3.1.3 Semi-explicit First-class Polymorphism

Recall that OCaml (Section 2.2.5) permits programmers to specify *first-class polymorphism* explicitly using records with *polymorphic fields*, where creating a record $\{\overline{\ell = e}\}$ *introduces* these polymorphic values wrapped in the record, and record field access $e.\ell$ *eliminates* a polymorphic value by instantiating it.

In Dromedary, we extend our formalisation of *algebraic data types* (Section 3.1.1), adding the polymorphic fields required to express *semi-explicit first-class polymorphic* types:

$$\mathsf{type}\ \overline{\alpha}\ \mathsf{F} \cong \prod_{i=1}^{n} \ell_i : \forall\overline{\beta_i}.\tau_i\,,$$

where the type scheme for the label ℓ in context Ψ is written as:

$$\Psi \vdash \ell : \forall\overline{\alpha}.(\forall\overline{\beta}.\tau) \to \overline{\alpha}\ \mathsf{F}\,.$$

To ensure that the record field $\ell = e$ is well-typed, where $\ell : \forall\overline{\alpha}.(\forall\overline{\beta}.\tau) \to \overline{\alpha}\ \mathsf{F}$, we verify that e has the type τ *for some instance of* $\overline{\alpha}$ and *all instances of* $\overline{\beta}$ – as $\overline{\beta}$ must be generic. Similarly, to determine if $e.\ell$ is well-typed, we check that e has the type $\overline{\alpha}\ \mathsf{F}$ *for some instance of* $\overline{\alpha}$ and that $e.\ell$ has the type τ *for some instance of* $\overline{\beta}$.

This reasoning may be expressed using the *existential and universal quantification* constraints introduced in Sections 2.1.2 and 3.1.2, respectively, resulting in the *label instantiation* constraints $\ell \leq \sigma \to \tau$ and $\ell \leq \tau$, semantically defined as:

$$\begin{aligned}
\ell \leq \tau_1 \to \tau_2 &\ \simeq\ \exists\overline{\alpha}, \overline{\beta}.\ \tau = \tau_1 \wedge \overline{\alpha}\ \mathsf{F} = \tau_2 && \text{if } \Psi \vdash \ell : \forall\overline{\alpha}.(\forall\overline{\beta}.\tau) \to \overline{\alpha}\ \mathsf{F}\,,\\
\ell \leq (\forall\overline{\beta}.C \Rightarrow \tau_1) \to \tau_2 &\ \simeq\ \exists\overline{\alpha}.\ \overline{\alpha}\ \mathsf{F} = \tau_2 \wedge \forall\overline{\beta}.\ \tau_1 = \tau \wedge C && \text{if } \Psi \vdash \ell : \forall\overline{\alpha}.(\forall\overline{\beta}.\tau) \to \overline{\alpha}\ \mathsf{F}\,.
\end{aligned}$$

As a result of these constraints, the typing rules for semi-explicit first-class polymorphism are as follows:

$$\frac{C \vDash \ell \leq \tau_1 \to \tau_2 \qquad C \vdash e : \tau_2}{C \vdash e.\ell : \tau_1}\ (\textsf{Dromedary-exp-field})$$

$$\frac{C \vdash e : \tau_1}{\ell \leq (\forall\overline{\beta}.C \Rightarrow \tau_1) \to \tau_2 \vdash \ell = e : \tau_2}\ (\textsf{Dromedary-exp-record})$$

3.1.4 Sharing

Sharing is the process of removing repeated types and variables; it is a critical technique for efficient type inference in ML. It also plays a role in more sophisticated type systems such as *ambivalent types* (Section 3.1.6) and ML^F [44].

In practice, types are shared by representing them as a *directed acyclic graphs* rather than *trees*. To illustrate this, the type `('a -> 'a) -> 'a -> 'a` is represented by the graphs depicted in Figure 3.2. The deduplication of repeated types is key to representing exponentially sized types using a linear graphical representation, as demonstrated in our evaluation (Section 4.3).

A formal treatment of sharing requires the concept of a *shallow type* ψ. In a graph-based description of types (as illustrated below), they are the structure of the internal nodes:

$$\psi ::= \overline{\alpha}\ \mathsf{F}\,,$$

where type variables represent *pointers*. By explicitly specifying variables (pointers), types are not duplicated. The formal details of graphical types and converting between (deep) types τ and shallow types are given in Appendices B, C. In type systems (such as Dromedary's), only the sharing of variables is significant: the sharing of internal nodes is not – we explain this further in Section 3.1.6.

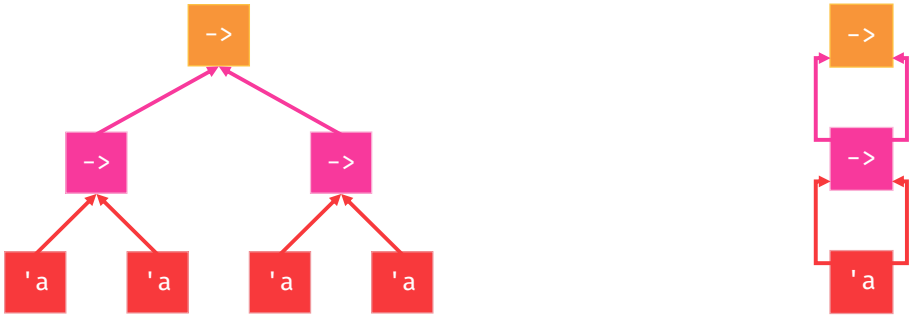


Figure 3.2: The tree-based (left) and graphical (right) representations of the type `('a -> 'a) -> 'a -> 'a`.

3.1.5 Polymorphic Variants

Intuitively, the type of a polymorphic variant consists of a sequence of *labelled types*. We call such a sequence a *row*. We implement a *multi-sorted algebra* for rows *à la Rémy* [42], where types τ are extended with *rows*, as shown below:

$$\tau ::= \ldots \mid \ell : \tau :: \tau \mid \partial\tau\,.$$

$\partial\tau$ is an infinite row, whose type is τ for every label, $\ell : \tau :: r$ is the row consisting of row r except the type for label ℓ is τ. Type variables used within the context of a row are called *row variables*, denoted ρ.

Labels within a row are annotated with *presence information*; a label is either `absent` or `present` with type τ, encoded using the unary type former τ `present` and nullary former `absent`.

Variants To encode variants, we use the unary type former Σ, where $\Sigma\ r$ denotes the type of a polymorphic variant with row r. OCaml and Dromedary syntactically hide the rows and row variables in polymorphic variants, simplifying the types exposed to the programmer.

However, this requires encoding our *empty, open and closed* polymorphic variants into *row-based* representation before type inference:

$$\begin{aligned}
[> \overline{\text{`}K\ [\ \text{of}\ \tau]}] &\simeq \Sigma\ (K_1 : [\tau_1] :: \ldots :: K_n : [\tau_n] :: \rho) && \text{where } \rho \text{ is free}\\
[< \overline{\text{`}K\ [\ \text{of}\ \tau]}] &\simeq \Sigma\ (K_1 : [\tau_1] :: \ldots :: K_n : [\tau_n] :: \partial\ \mathsf{absent})\\
[\,] &\simeq \Sigma\ (\partial\ \mathsf{absent})\,,
\end{aligned}$$

where $[\tau]$ is $\mathsf{unit}\ \mathsf{present}$ in the optional case, and $\tau\ \mathsf{present}$ otherwise.

Equi-recursive Types Polymorphic variants require *equi-recursive* types to represent recursive variants such as `'list` in the example below:

```
let rec length t =
  match t with
  | `Nil -> 0
  | `Cons (_, t) -> 1 + length t
```

has the type `[< `Nil | `Cons of 'a * 'list] as 'list -> int`.

Formally, a recursive type has the form $\mu\alpha.\tau$ where α may appear in τ and represents a recursive occurrence of the type. With equi-recursive types, the recursive type $\mu\alpha.\tau$ and its expansion (or unfolding) $\{\mu\alpha.\tau/\alpha\}\tau$ are *equal*; that is, the two types denote the *same type*. This allows one

to *fold* and *unfold* an equi-recursive type infinitely, making comparisons between equi-recursive types more difficult. Fortunately, by utilising sharing (Section 3.1.4), we may represent equi-recursive types using *directed cyclic graphs*. In practice, this is implemented by *removing the occurs-check* in unification (Section 3.2.2).

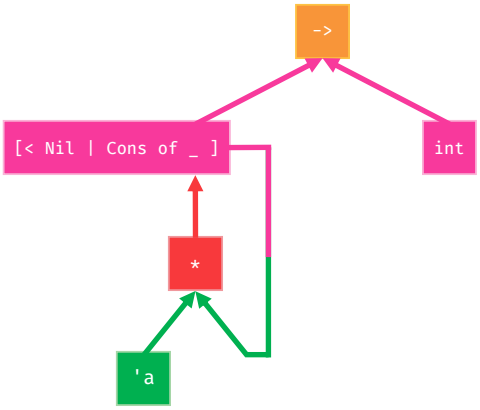
We extend Dromedary's types τ with aliases and recursive forms to encode the `as` construct, since formalising `as` directly is challenging:

$$\tau ::= \ldots \mid \tau \text{ where } \alpha = \tau \mid \mu\alpha.\tau\,,$$

where τ_1 where $\alpha = \tau_2$ defines the alias $\alpha = \tau_2$ in τ_1. We may encode the type for `length` as

$$\gamma \to \mathsf{int} \text{ where } \gamma = \mu\beta.[< \text{`Nil} \mid \text{`Cons of } \alpha \times \beta]\,.$$

The cyclic graph on the right depicts this type.



Typing Rules Expressions and patterns are extended with variants:

$$e ::= \ldots \mid \text{`}K\ [e]\,, \qquad\qquad p ::= \ldots \mid \text{`}K\ [p]\,.$$

We introduce *subtyping constraints* of the form $\tau \leq \overline{\text{`}K\ [\text{of } \tau]}$ and $\tau \geq \overline{\text{`}K\ [\text{of } \tau]}$ to reason about the *lower (<) and upper (>)* bounds of polymorphic variants (Section 2.2.6) using constraints. Constructing a variant `$K\ e$ (the nullary case being analogous) has a fairly elementary rule:

$$\frac{C \vDash \tau_2 \geq \text{`}K \text{ of } \tau_1 \qquad C \vdash e : \tau_1}{C \vdash \text{`}K\ e : \tau_2}\ \text{(Dromedary-exp-variant)}$$

The typing rules for match expressions and cases, given in Figure 3.3, are more involved, with several edge-cases. Dromedary-variant-match-closed implements *closed* pattern matching, where every variant constructor is explicitly handled in a case. For *open* pattern matching, we require a default case of the form: $\_ \to e$ in Dromedary-variant-match-open.

$$\frac{\begin{array}{l} C \vdash e : \tau_e \qquad C \vDash \tau_e \leq \overline{\text{`}K_i\ [\text{of } \tau_i]} \\ \forall 1 \leq i \leq n.\ C_i \vdash \text{`}K\ [p_i] \to e_i : \text{`}K_i\ [\text{of } \tau_i] \Rightarrow \tau \end{array}}{C \wedge \bigwedge_{i=1}^{n} C_i \vdash \text{match } e \text{ with } \overline{\text{`}K_i\ [p_i] \to e_i} : \tau}\ \text{(Dromedary-variant-match-closed)}$$

$$\frac{\begin{array}{l} C \vdash e : \tau_e \qquad C \vDash \tau_e \geq \overline{\text{`}K_i\ [\text{of } \tau_i]} \\ \forall 1 \leq i \leq n.\ C_i \vdash \text{`}K\ [p_i] \to e_i : \text{`}K_i\ [\text{of } \tau_i] \Rightarrow \tau \\ C_{n+1} \vdash e_{n+1} : \tau \end{array}}{C \wedge \bigwedge_{i=1}^{n+1} C_i \vdash \text{match } e \text{ with } (\overline{\text{`}K_i\ [p_i] \to e_i} \mid \_ \to e_{n+1}) : \tau}\ \text{(Dromedary-variant-match-open)}$$

$$\frac{[C_1 \vdash p : \tau_1 \rightsquigarrow \Delta] \qquad C_2 \vdash e : \tau_2}{[C_1 \wedge \text{def } \Delta \text{ in}]\ C_2 : \text{`}K\ [p] \to e : \text{`}K\ [\text{of } \tau_1] \Rightarrow \tau_2}\ \text{(Dromedary-variant-case)}$$

Figure 3.3: A selection of Dromedary's polymorphic variant typing rules for pattern matching.

We remark that Dromedary only type checks *shallow patterns* for polymorphic variants, namely patterns of the form `$K\ p$ where p does not include a polymorphic variant. Whereas OCaml

supports *deep patterns* by using *exhaustive pattern checking* [14] to determine whether the matched variant type τ_e is *closed* or *open*. Exhaustive checking in the presence of GADTs reduces to *proof search* [15], which is **outside the scope of this dissertation** due to its complexity; nonetheless, we foresee no difficulties incorporating exhaustive checking in our constraint-based approach.

3.1.6 Generalised Algebraic Data Types

Generalised algebraic data types extend algebraic data types in two ways. The first is an instance of Laüfer and Odersky's extension to ML with existential types [27]. The second extension *allows us to constrain the type for each constructor* (Section 2.2.3); we do so by permitting constructors to have *constrained* type schemes with *existential variables*:

$$\Psi \vdash K : \forall\overline{\alpha}.\exists\overline{\beta}.C \Rightarrow [\tau \to] \ \overline{\alpha}\ \mathsf{F}\,.$$

For example, one may write GADT α expr (Listing 2.6), using equality constraints, as shown in Figure 3.4 and Listing 3.2. The novelty of GADTs lies in the constraint C; in order to use

$$\begin{aligned}
\mathsf{Int} &: \forall\alpha.\alpha = \mathsf{int} \Rightarrow \mathsf{int} \to \alpha\ \mathsf{expr} \\
\mathsf{Pair} &: \forall\alpha.\exists\beta\gamma.\alpha = \beta \times \gamma \Rightarrow \beta\ \mathsf{expr} \times \gamma\ \mathsf{expr} \to \alpha\ \mathsf{expr} \\
\mathsf{Fst} &: \forall\alpha.\exists\beta\gamma.\alpha = \beta \Rightarrow (\beta \times \gamma)\ \mathsf{expr} \to \alpha\ \mathsf{expr} \\
\mathsf{Snd} &: \forall\alpha.\exists\beta\gamma.\alpha = \gamma \Rightarrow (\beta \times \gamma)\ \mathsf{expr} \to \alpha\ \mathsf{expr}
\end{aligned}$$

Figure 3.4: The formal definition of the type α expr, originally defined in Listing 2.6.

```
type 'a expr =
  | Int of int constraint 'a = int
  | Pair of 'b 'c. 'b expr * 'c expr constraint 'a = 'b * 'c
  | Fst of 'b 'c. ('b * 'c) expr constraint 'a = 'b
  | Snd of 'b 'c. ('b * 'c) expr constraint 'a = 'c
```

Listing 3.2: The type definition of the `'a expr` GADT in Dromedary – new syntax was introduced for existential variables and explicit constraints.

a constructor $K\ e$, e must have the type τ and the type variables $\overline{\alpha}, \overline{\beta}$ must be instantiated such that the constraint C is satisfied. Pattern matching now *binds local type variables and constraints*: If $K\ p$ matches a value of type $\overline{\alpha}\ \mathsf{F}$, then *there exists unknown types* $\overline{\beta}$ *that satisfy* C which may be bound in the fragment of p.

Ambivalent Types Dromedary's typing discipline for GADTs is based on Garrigue's and Rémy's *ambivalent types* [17]. Informally, an *ambivalent type* ζ is a *set of types* that are *equal under the local constraints*; they are used when the type is *ambiguous* – namely when $|\zeta| > 1$. An ambivalent type is said to have *leaked* if the set of types are no longer equal under the local constraints. To illustrate this, we consider:

```
let g (type a) (eq : (a, int) eq) (y : a) =
  match eq with Refl -> if y > 0 then y else 0
```

where the equality type `eq` is given in Listing 2.7. The `then` branch returns `y`, with type `a`, whereas the `else` branch returns a value of type `int`. The resultant type is the ambivalent

type $\zeta = \{\mathsf{a}, \mathsf{int}\}$, which represents a type that is *either* `a` or `int` When exiting the scope of `match` branch, ζ is *leaked* – since the local equality $\mathsf{a} = \mathsf{int}$ is no-longer present in the context!

Ambiguities are eliminated using *annotations* (Section 3.1.2); for an expression $(e : \tau)$, the expressions e and $(e : \tau)$ may have differing ambivalent types ζ_1, ζ_2, but τ must be included in both – to ensure soundness.

Ambivalent types rely on *sharing* (Section 3.1.4) to guarantee the inference of principal types. When instantiating a type scheme $\forall\alpha.\zeta$ *without* sharing, we lose the information that *all* copies of α must be *structurally equal* since types that are not structurally equal may be equated due to local equalities. Sharing recovers this information as each copy of α corresponds to the *same* node in the graph-based representation of ζ (Section 3.1.4).

Ambivalent Constraints We now present our novel constraint language, extended with ambivalent types:

$$\begin{aligned} C ::= {} & \mathsf{true} \mid \mathsf{false} \mid C \wedge C \mid \forall\alpha.C \mid \exists\zeta.C \\ & \mid \zeta = \zeta \mid \psi \subseteq \zeta \mid R \implies C \end{aligned}$$

$$\psi ::= \alpha \mid \overline{\zeta}\ \mathsf{F}\,,$$

where ζ is an *ambivalent type variable* and ψ is a *shallow type*, either consisting of a *shallow type former* $\overline{\zeta}\ \mathsf{F}$ or a rigid variable α. $R ::= \mathsf{true} \mid R \wedge R \mid \tau = \tau$ defines *rigid constraints*; constraints solely consisting of equalities between (rigid) types.

We briefly highlight the new constructs of our language. We introduce *existential quantifiers* $\exists\zeta.C$ *for ambivalent type variables*. We enforce *sharing* by preventing *(deep) types* τ from occurring in constraints. The $\zeta = \zeta$ constraint is used in lieu of $\tau = \tau$, providing a first-order equality constraint between ambivalent types; and the *subset constraint* $\psi \subseteq \zeta$, read as: *the ambivalent type* ζ *includes the type* ψ, is used to define explicitly shared types.

In Dromedary, we restrict the local constraints of GADT types to *rigid constraints*, hence type schemes for constructors are of the form:

$$\Psi \vdash K : \forall\overline{\alpha}.\exists\overline{\beta}.R \Rightarrow [\tau \to] \ \overline{\alpha}\ \mathsf{F}\,,$$

where variables $\overline{\beta}$ are considered *rigid*. This **mimics** OCaml's requirement to annotate GADT types with *rigid variables* [16]. We may introduce local rigid constraints using the new *implication constraint* $R \implies C$. Semantically, implication constraints also ensure no ambivalent types are leaked when exiting the *scope of the implication*.

In practice, our constraint language differs from our presentation here since we implement ambivalent types using *scoped abbreviations*, which provides an efficient (linear) consistency and leakage check. This is the approach used by OCaml (4.12.0). While we do not fully explain this, it seems important to acknowledge the difference.

Typing Rules We begin by extending the notion of a *fragment*, introduced in Section 3.1.1, to *generalised fragments*. A *generalised fragment* Θ is a triple, consisting of a context of existential variables $\overline{\beta}$, a rigid constraint R, and a fragment Δ, written as $\Theta ::= \exists\overline{\beta}.\Delta \Rightarrow R$. These generalised fragments describe *all typing information* gained from a pattern that includes GADTs.

The typing rules[2] (Figure 3.5) extend our presentation of algebraic data types (Section 3.1.1). Dromedary-pat-construct checks whether the constructor K has the type $\tau_1 \to \tau_2$, and binds

[2]The presented typing rules here differ from the ones given in Appendix C due to sharing, which is a (*trivial*) technical detail we omit.

local existential variables $\overline{\beta}$ and constraints R. The sub-pattern p checked to have the type τ_1, binding the fragment Θ. The novelty of GADTs require $\overline{\beta}$ and R to be bound in the fragment of $K\ p$, extending Θ, which we write as $\exists\overline{\beta}.\Theta \Rightarrow R$.

Dromedary-pat-tuple requires each pattern p_i in the tuple $(p_1, \ldots, p_n)$ of type $\tau_1 \times \cdots \times \tau_n$ to have the type τ_i. Each pattern produces a fragment Θ_i. The resultant fragment of $(p_1, \ldots, p_n)$, is the concatenation $\Theta_1 \times \cdots \times \Theta_n$ of the individual fragments.

In Dromedary-case, the pattern p is checked against the matched type τ_1, giving us the fragment $\exists\overline{\beta}.\Delta \Rightarrow R$. The case body e is then checked to have the type τ_2, under the assumptions of R, using an implication constraint. The local existential variables $\overline{\beta}$ are *universally quantified* since they represent *unknown local types* within C_2. We also note that the constraint C_1, which the pattern is checked under, also assumes the local constraints R – permitting local constraints to flow between patterns in tuples.

$$\frac{C \vDash K \leq \exists\overline{\beta}.\tau_1 \to \tau_2 \Rightarrow R \qquad C \vdash p : \tau_1 \rightsquigarrow \Theta}{C \vdash K\ p : \tau_2 \rightsquigarrow \exists\overline{\beta}.\Theta \Rightarrow R} \text{ (Dromedary-pat-construct)}$$

$$\frac{\forall 1 \leq i \leq n \qquad C_i \vdash p_i : \tau_i \rightsquigarrow \Theta_i}{\bigwedge_{i=1}^{n} C_i \vdash (p_1, \ldots, p_n) : \tau_1 \times \cdots \times \tau_n \rightsquigarrow \Theta_1 \times \cdots \times \Theta_n} \text{ (Dromedary-pat-tuple)}$$

$$\frac{C_1 \vdash p : \tau_1 \rightsquigarrow \exists\overline{\beta}.\Delta \Rightarrow R \qquad C_2 \vdash e : \tau_2}{\forall\overline{\beta}.R \implies C_1 \wedge \text{ def } \Delta \text{ in } C_2 \vdash p \to e : \tau_1 \Rightarrow \tau_2} \text{ (Dromedary-case)}$$

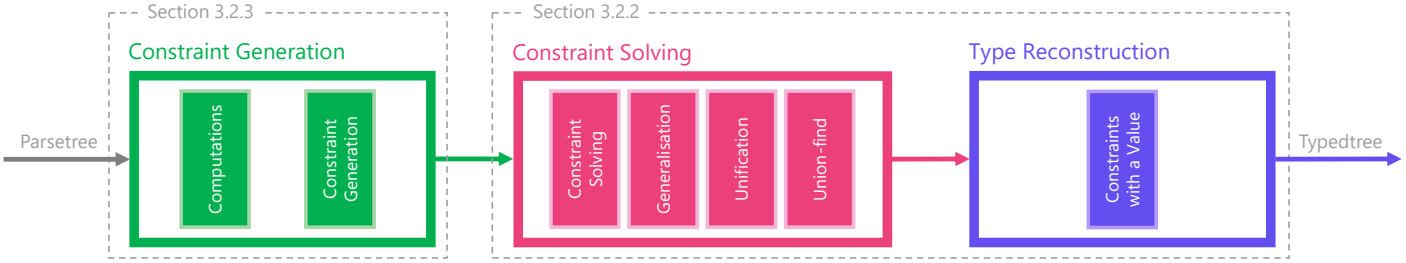
Figure 3.5: The relevant typing rules for GADTs from Dromedary's type system.

3.2 Inference Implementation

The OCaml compiler takes a program, parses it creating a *parsetree*, and performs type inference creating **an explicitly typed intermediate representation**, called the *typedtree*, used in the *backend* to generate bytecode.

Figure 3.6: A phase diagram of the OCaml compiler.

Dromedary re-implements the first **two stages**. The design of Dromedary embodies the *separation of concerns (SoC) principle* [26], separating these stages into more distinct modular phases using a *constraint-based approach*. Our implementation focuses on correctness and clarity over performance (unless specified otherwise).



Our explanation of Dromedary's type inference is organised according to the constraint pipeline (Figure 1.1), with the Sections 3.2.2 and 3.2.3 structured as illustrated above.

3.2.1 Repository Overview

The top-level project directory consists of the source code `src/`, tests `test/` and benchmarks `benchmark/`, with additional files for the Dune build system. Table 3.1 gives an overview of the repository structure. Within `src/`, Dromedary is split into a `parsing` library, a `constraints`

Library	Description	Lines
`src/constraints`	Implements the constraint language and constraint solver.	3022
`src/constraints/constraint`	Contains type definitions, pretty-printing and module types for the constraint language.	717
`src/constraints/solver`	Solves constraints, producing their values or an `error`.	2108
`src/parsing`	Contains type definitions and pretty-printing for *parsetree*. Contains lexing and parsing code.	1853
`src/typing`	Contains type definitions and pretty-printing for *typedtree*. Contains code for constraint generation and type reconstruction.	3324
`tests/`	Unit tests using Alcotest and Expect.	6853
`benchmarks/`	Benchmarks of Dromedary's inference.	637

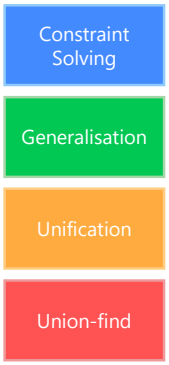
Table 3.1: Repository overview. All code is written in OCaml.

library (Section 3.2.2), and a `typing` library (Section 3.2.3). My project repository broadly follows the structure of the OCaml compiler, aiding in interoperability with the OCaml compiler in the future.

3.2.2 Constraints and Type Reconstruction

The design of the `Constraints` library focuses on *modularity* and *efficiency*, preferring *mutable* data structures over *immutable* ones. This approach ensures Dromedary is **equally performant** as OCaml (Section 4.3). We note that **no global mutable state is exposed** since the constraint language provides an immutable interface.

Modules As mentioned in Section 2.2.1, Dromedary's constraints library heavily relies on OCaml's module system and *functors* to provide modularity, internally and externally. The library is split into four parameterised modules forming a *layered abstraction stack*: (a) at the bottom is Tarjan's efficient **union-find data structure** [47]; (b) above it is Huet's **unification algorithm** [21]; (c) then Rémy's efficient **rank-based generalisation** [41]; (d) at the top, the implementation of the **constraint language and solver**, implementing Pottier's *constraints with a value* [39].



Constraints with a Value The objective of the constraint solver is to determine whether the constraints are satisfiable or unsatisfiable (`true` or `false`). Unfortunately, this approach doesn't work with *type reconstruction* (or *elaboration*) – the process of constructing the *typedtree*.

Many languages using constraint-based inference, such as Haskell [37], resort to combining the phases of constraint solving and elaboration. However, this approach violates the SoC principle that Dromedary adheres to. In [39], Pottier proposes an alternative implementation of constraints that facilitates solving and elaboration in a modular fashion. To allow elaboration, Pottier extends constraints to not only return information of satisfiability but also *values*.

This gives rise to the notion of an "α-constraints", a constraint which (if satisfiable) produces a result of type α. In Dromedary, these constraints are represented as **generalised algebraic datatype** `'a Constraint.t`:

```
type _ t =
  | True : unit t
  | Conj : 'a t * 'b t -> ('a * 'b) t
  | Eq : variable * variable -> unit t
  | ...
  | Def : def_binding list * 'a t -> 'a t
  | Let :
    'a let_binding list * 'b t
    -> ('a term_let_binding list * 'b) t
  | Return : 'a -> 'a t
  | Map : 'a t * ('a -> 'b) -> 'b t
  | Decode : variable -> Decoded.Type.t t
```

For example, the conjunction constraint `Conj` (C_1, C_2) returns a pair of values (v_1, v_2) composed of the values returned by C_1, C_2 respectively. Following Pottier, we also extend our constraints language with a `Map` (C, f) construct which evaluates C to some value v (if satisfiable), and returns the value $f\ v$. We refer the reader to [39] for the complete formal semantics of constraints with a value. This approach allows Dromedary to express *constraint generation* and *type reconstruction* using the constraint language – in the same place!

The constraints library (Listing 3.4) is parameterised by the notion of an *algebra*. Informally, an `Algebra` specifies the *term variables* embedded in the constraint language and the structure of Dromedary's *types*. The constraints library provides an *abstract type* `'a t` for constraints that produce a value of type `'a`, a number of *combinators* for constructing constraints, and a *solve* function that solves the constraint, either returning a value of type `'a` or an `error`.

The constraints language is equipped with `return`, `both ( &~ )` and `map` combinators, forming an *applicative functor* (Section 2.2.2). Dromedary makes extensive use of this abstraction with Jane Street's `ppx_let` [4], which provides syntactic sugar for working with applicatives (and monads):

```
(exp1 &~ exp2)
>>| fun (exp1, exp2) ->
Texp_app (exp1, exp2)
```

```
let%map exp1 = exp1
and exp2 = exp2 in
Texp_app (exp1, exp2)
```

Listing 3.3: Desugared (left) verses `ppx_let` syntax (right) for applicatives (and monads).

Constraints, like any intermediate representation, may be optimised. Dromedary uses *smart constructors* to perform *peephole optimisations* on constraints. For instance, the equivalence $\forall\overline{\alpha}.\forall\overline{\beta}.C \simeq \forall\overline{\alpha},\overline{\beta}.C$ may be used to reduce the number of (expensive) *generalisation operations* performed. Such optimisations are **not possible** in OCaml's type checker.

```
module Make (Algebra : Algebra) : sig
  (** Abstract type for ['a Constraint.t] *)
  type 'a t

  (** Constraints form an applicative functor *)
  include Applicative.S with type 'a t := 'a t
  include Applicative.Let_syntax with type 'a t := 'a t

  (** Combinators for constructing constraints *)
  val ( &~ ) : 'a t -> 'b t -> ('a * 'b) t
  ...

  val solve : 'a t -> ('a, [> Solver.error ]) Result.t
end
```

Listing 3.4: A snippet of the `Constraints` library interface.

```
(** The type ['a t] denotes a node within a given disjoint set.
    ['a] is the type of the value (descriptor) of the node. *)
type 'a t

val make : 'a -> 'a t
val find : 'a t -> 'a
val union : 'a t -> 'a t -> f:('a -> 'a -> 'a) -> unit
```

Listing 3.5: The module signature for Dromedary's implementation of the union-find data structure.

Union find Unification is the process of solving equations of the form:

$$U ::= \mathsf{true} \mid U \wedge U \mid \exists \alpha . U \mid \tau = \tau \,.$$

Tarjan's union-find data structure (Listing 3.5) implements a family of *disjoint sets* (equivalence classes of types), each set associated with a *descriptor* (the representative type); with the following operations: `find t` returns the descriptor of set `t`; `union t1 t2 ~f` computes the union of the sets `t1`, `t2` merging their descriptors using `f`.

Dromedary implements a *forest-based* structure, consisting of a collection of trees, each tree representing a disjoint set:

```
type 'a t = 'a node ref
and 'a node =
  | Root of { rank : int; desc : 'a }
  | Link of 'a t
```

A `'a node` represents a node in a tree (a *set*): which is either the *root* of the graph, containing the *descriptor* of the set, or an *internal node* with no data and a parent node, known as a *link*. For **quasi-linear complexity** in time for `find` and `union`, we implement *path compression* and *union by rank* [47], the latter not being implemented in OCaml's type checker.

Unification and Structures Dromedary extends first-order unification with several non-trivial extensions: (a) *under a mixed prefix* [32], unification in the presence of existential *and* universal quantifiers (Section 3.1.2); (b) the addition of *unscoped* equational context $\mathcal{A}$ for type

abbreviations; (c) *scopes* and *scoped* equational contexts for ambivalence (Section 3.1.6); (d) rows (Section 3.1.5).

Each extension to unification is independent and thus may be implemented *modularly*, using the notion of a *structure*, which describes the *descriptor* attached to equivalence classes in unification. The interface for a structure is given in Listing 3.6, consisting of: (a) an abstract type for structures `'a t` which contains children of type `'a`; (b) a function `merge`, which is used to equate two structures `t1, t2` of type `'a t`, within some *context* `ctx` of type `'a ctx`, returning the resultant *merged* structure or raising the exception `Cannot_merge` if the structures are not compatible; (c) *functorial* functions (Section 2.2.2) such as `map`, `iter`, and `fold` used to traverse the structure performing various element-wise operations.

```
module type Structure = sig
  type 'a t

  type 'a ctx
  exception Cannot_merge
  val merge
    :  ctx:'a ctx -> equate:('a -> 'a -> unit)
    -> 'a t -> 'a t -> 'a t

  val map : 'a t -> f:('a -> 'b) -> 'b t
  val iter : 'a t -> f:('a -> unit) -> unit
  val fold : 'a t -> f:('a -> 'b -> 'b) -> init:'b -> 'b
end
```

Listing 3.6: The module signature for first-order unification structures.

Structures may be composed and extended using *functors* (Section 2.2.1). To illustrate this, we may define a structure called `First_order` (Listing 3.7) which extends an arbitrary structure `S` with *variables*.

```
module First_order (S : Structure) : sig
  type 'a t =
    | Var
    | Structure of 'a S.t

  include S with type 'a t := 'a t and type 'a ctx = 'a S.ctx
end
```

Listing 3.7: An example of a composable unification structure using OCaml's functors – the structure `First_order` extends a structure `S` adding (uni-sorted) variables.

The `Unifier` module, which implements *types* and unification, is parameterised by a `Structure`. Types are defined as *cyclic directed graphs* where each node contains a *structure*:

```
type t = desc Union_find.t
(** Arbitrary [id] field, used for printing & total ordering *)
and desc = { id : int; structure : t Structure.t }
```

This *graphical* definition permits *equi-recursive* types (Section 3.1.5) and **sharing** (Section 3.1.4), which is key for efficient unification.

Generalisation In the context of constraint solving, generalisation is the process of simplifying *constrained type schemes* $\forall\overline{\alpha}.C \Rightarrow \tau$ to *type schemes* $\forall\overline{\beta}.\tau'$, which is performed when solving let constraints.

For **approximately linear time** generalisation and instantiation, we implement Rémy's efficient *rank*-based scheme. Each type variable in the constraint is annotated with an integer *level* (or *rank*), which is used to determine the scope of the variable (in constant time): variables with level l are bound in the l^{th} nested $\forall$-quantifier in constrained type schemes of let constraints, with 0 being the outermost level. For example, the following depicts the levels within the generated constraint of expression let id = fun x → x in id:

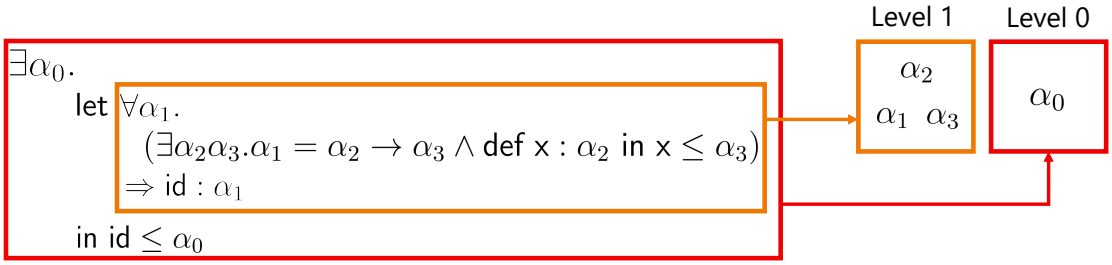


Figure 3.7: A visualisation of rank-based generalisation [41] for the generated constraint of let id = fun x → x in id.

The essential observation is that we cannot generalise variables bound in the enclosing scope, such as α_0, since they may be equated after we exit the current scope – namely in the $\mathsf{in}\ \mathsf{id} \leq \alpha_0$ portion of the above constraint.

When equating two type variables during unification, we reduce their level to the lowest of their levels (outermost scope). Thus, when generalising (exiting l^{th} level), we only generalise variables whose level is greater than or equal to l – variables that are not bound in an enclosing scope.

3.2.3 Typing and Constraint Generation

The `Typing` library implements constraint generation and elaboration for the *typedtree*, using the `Constraints` library. Unlike OCaml's current `Typing` library, Dromedary's focuses on clarity and correctness relying on abstractions such as monads and explicit error types over side-effecting operations, at a potential performance cost.

Computations One of Dromedary's fundamental (and novel) abstractions in constraint generation is the idea of a *computation*. Concretely, computations (and *binders*) are defined by the following *domain-specific language*:

$\tau ::= \tau \to \tau \mid \tau\ \mathsf{computation} \mid \tau\ \mathsf{binder} \mid \tau\ \mathsf{constraint} \mid \ldots$	Types
$e ::= x \mid \mathsf{fun}\ x \to e \mid e\ e \mid \{\ t\ \} \mid [\ u\ ] \mid C \mid \ldots$	Expressions
$t ::= \mathsf{let}\ x = t;\ t \mid \mathsf{return}\ e \mid \mathsf{bind}\ x = u;\ t \mid \ldots$	Computation commands
$u ::= \mathsf{let}\ x = u;\ u \mid \mathsf{return}\ e \mid \mathsf{sub}\ x = t;\ u \mid \mathsf{exists} \mid \mathsf{forall} \mid \ldots$	Binder commands

Figure 3.8: The formal syntax of computations and binders.

A τ computation computes a value of type τ within the context required for constraint generation. Within computations, one may define the notion of a *binder*, which represents a context for binding variables within constraints; for example, $\exists\alpha.[\cdot]$ is a binding context for the variable

α with a 'hole' (represented by the binding command exists). Computations and binders both form *monads* (Section 2.2.2) with `bind` (let commands) and `return` operations.

The computation bind $x = u$; t applies the binder u of type τ, binding x to its value, and fills u's 'hole' with the constraint returned by the computation t. The binder sub $x = t$; u computes the computation t of type τ, binding its value to x and returns the binder u. The DSL is shallowly embedded in OCaml using `ppx_let` and OCaml's let-binding operators for `bind` and `sub` (using `let@` and `let&`, respectively). See Appendix D for the complete embedding.

Constraint Generation Dromedary's constraint generation utilises *α-constraints* and *computations* to express constraint generation *and* type reconstruction *together*, resulting in concise, compositional, and maintainable code that naturally reflects the formal constraint mapping $[\![e : \tau]\!]$ (Figure 2.3).

For example, the following snippet generates constraints for the application exp1 exp2 (following the definition in Figure 2.3) *and* constructs the respective typedtree fragment `Texp_app (exp1, exp2)` in the *same* code segment:

```
| Pexp_app (exp1, exp2) ->
  (* bind [var] existentially *)
  let@ var = exists () in
  (* check [exp1] has type [var -> exp_type];
     and [exp2] has type [var] *)
  let%bind exp1 = lift (infer_exp exp1) (var @-> exp_type) in
  let%bind exp2 = infer_exp exp2 var in
  return
    (let%map exp1 = exp1
     and exp2 = exp2 in
     Texp_app (exp1, exp2))
```

Listing 3.8: A snippet of Dromedary's constraint generation illustrating the usage of constraints, computations, and binders for clear, compositional, and maintainable code.

3.3 Summary

This chapter began by introducing Dromedary's type system, the **first** unified presentation of OCaml's type system in a constraint-based setting, which we believe to be (a) *more natural* than other presentations for certain features, such as GADTs; (b) better suited to correctness proofs and formal verification of the type checker, an ongoing field of research [11, 6]. We discussed various *advanced type system features* of Dromedary and their constraint-based formalisation, requiring many **novel extensions** to the constraints language. The author wishes to emphasise that Dromedary implements **additional features not covered in this section**[3], including *abstract types*, *side-effecting primitives*, *type abbreviations*, *extensible variants*, and *structures*.

Having discussed the theoretical aspects of Dromedary's type system and its features, we explored the practical implementation of Dromedary's type inference algorithm – focusing on mechanisms that allow **Dromedary to implement SoC**. Dromedary's `constraints` library is **fundamentally modular, while implementing quasi-linear constraint solving in time**[4] provided type schemes have bounded size [30]. The `typing` library, which implements Dromedary's constraint-based inference, was designed to focus on clarity and correctness; permitting the effortless description of constraints and type reconstruction using *computations*.

[3]Due to the page limit.
[4]Not formally analysed.

4 Evaluation

In this section, I will evaluate whether the implementation of Dromedary fulfilled the success criteria outlined in the project proposal (Appendix E). In Section 4.1, I demonstrate that Dromedary far exceeds the success criteria. Following this, I explore the permissiveness of Dromedary's type system in comparison to OCaml's, *empirically* showing that **Dromedary is as permissive as OCaml**. Finally, in Section 4.3, I show that **Dromedary outperforms OCaml** in our benchmarks.

4.1 Project Requirements and Success Criteria

Analysing the requirements stated in the preparation chapter (Section 2.3), I have achieved **all my must-have, should-have and could-have requirements** (Table 2.1). Considering the original success criteria detailed in the proposal (Appendix E):

Design Dromedary's type system; supporting ML with GADTs Dromedary's type system far exceeds the minimum requirements of ML with GADTs, implementing all the features from Core ML and support for semi-explicit first-class polymorphism, polymorphic variants, type abbreviations, polymorphic recursion, extensible variants, and structures.

Design the constraint language for Dromedary I successfully designed a constraint language capable of expressing all of Dromedary's features (Section 3.1); often requiring **novel extensions** on previous work (Section 2.1.2).

Implement a constraint-based inference algorithm for Dromedary Not only did I implement a constraint-based inference algorithm for Dromedary; but one that was fundamentally more modular **and** more performant than OCaml's inference algorithm!

Evaluate the permissiveness and efficiency of Dromedary I use the Jane Street Expect Test and Core Bench library to evaluate the permissiveness and performance of Dromedary's inference algorithm, performing **427 experiments**.

4.2 Permissiveness of Dromedary

In this section, we discuss the permissiveness of Dromedary's type system. Since Dromedary implements a subset of OCaml, we aim to show that **Dromedary is equally permissive to OCaml**; that is to say that everything Dromedary successfully type checks, OCaml type checks and vice versa[1].

Methodology We split Dromedary's type system into its main constituent features: Core ML features, semi-explicit first-class polymorphism, polymorphic recursion, GADTs, and polymorphic variants.

We use the selection of relevant programs used in the OCaml compiler test suite [51] for each feature. If insufficient programs are available from the said test suite, then we use a carefully crafted corpus of programs taken from various academic papers and textbooks.

[1] In the implemented features.

Results We completed a total of **412 tests**, summarised in Table 4.1. Each test we performed concluded that OCaml and Dromedary **are equally permissive in the implemented features**. We briefly discuss our tests and results in two categories: tests from the OCaml test suite, and tests using examples from other sources:

OCaml testsuite: Of the **631 relevant tests** for semi-explicit first-class polymorphism, GADTs and polymorphic recursion in the OCaml test suite, Dromedary was able to implement **283** of them.

All tests that we were unable to implement were due to features not supported by Dromedary:

- 16% (57) were due to interactions with the module system,
- 47% (159) relied on objects and classes,
- 37% (132) involved other miscellaneous features of OCaml that are not supported in Dromedary.

Other sources: Since OCaml's test suite lacked representative tests for Core ML features and polymorphic variants, we relied on other sources for examples.

For Core ML features, we curated a corpus of **111 programs** using examples from Whitington's 'OCaml from the very beginning' [40], Paulson's 'ML for the working programmer' [23] and the foundations of computer science lecture notes [1]. Similarly, for polymorphic variants, we used examples from 'Real-world OCaml' [34] and various papers on polymorphic variants [12, 42], resulting in **18 additional programs**.

Dromedary was able to correctly type check **all** of these programs.

In practice, we found that translating programs between Dromedary and OCaml only requires *minor syntactic changes* where the syntax differs – for example, scoped annotations (Section 3.1.2). In total, we approximately translated **4100 lines** of OCaml to Dromedary.

Since Dromedary successfully passes **all tests** relevant to its type system features, we conclude that *Dromedary is equally permissive as OCaml*. These are encouraging results, *suggesting* that the constraint-based approach used in Dromedary could be integrated with OCaml *without significant backwards compatibility issues*, demonstrating the practicality of our work.

Given that we performed 412 tests, we also view these results as *empirical evidence* for the **correctness of Dromedary's type system and its implementation**.

4.3 Benchmarks

In this section, we discuss the efficiency and asymptotic behaviour of Dromedary; substantiating our claim that Dromedary implements *quasi-linear constraint solving* and showing that **Dromedary is more performant than OCaml**.

Methodology OCaml programs are type-checked using the OCaml compiler. I ensure the benchmarks are comparative by modifying OCaml's inference algorithm to ensure it only type checks relevant features – for example, disabling inference for objects/classes and modules. This was achieved by forking the implementation of the OCaml compiler; removing many unnecessary libraries and modules, and replacing certain functions within the implementation with stubs. Since Dromedary infers principal types and permits equi-recursive types (Section 3.1.5), OCaml's `-principal` and `-rec-types` compiler flags are enabled.

Feature	Testsuite Files	Tests	
		OCaml	**Dromedary**
Core ML:			
	`whitington.ml`	51	51
	`paulson.ml`	5	5
	`focs.ml`	22	22
	`infer_core.ml`	33	33
Semi-explicit First-class Polymorphism:			
	`poly.ml`	141	9
	`pr7636.ml`	3	2
	`pr9603.ml`	2	0
	`error_messages.ml`	10	0
Polymorphic Recursion:			
	`poly.ml`	5	5
GADTs:			
	`ambiguity.ml`	16	13
	`ambivalent_apply.ml`	3	3
	`didier.ml`	7	5
	`dynamic_frisch.ml`	24	24
	`gadthead.ml`	2	0
	`name_existentials.ml`	12	12
	`nested_equations.ml`	8	2
	`omega07.ml`	56	56
	`or_patterns.ml`	58	0
	`term_conv.ml`	5	5
	`unify_mb.ml`	14	14
	`principality_and_gadts.ml`	38	19
	`return_type.ml`	3	0
	`yallop_bugs.ml`	4	0
	`unexpected_existentials.ml`	16	2
	`test.ml`	84	56
	`pr*.ml`	120	56
Polymorphic Variants:			
	`docs.ml`	4	4
	`garrigue.ml`	5	5
	`real_world_ocaml.ml`	7	7
	`remy.ml`	2	2

Table 4.1: A summary of tests in each file for Dromedary and OCaml – consisting of **412 tests**.

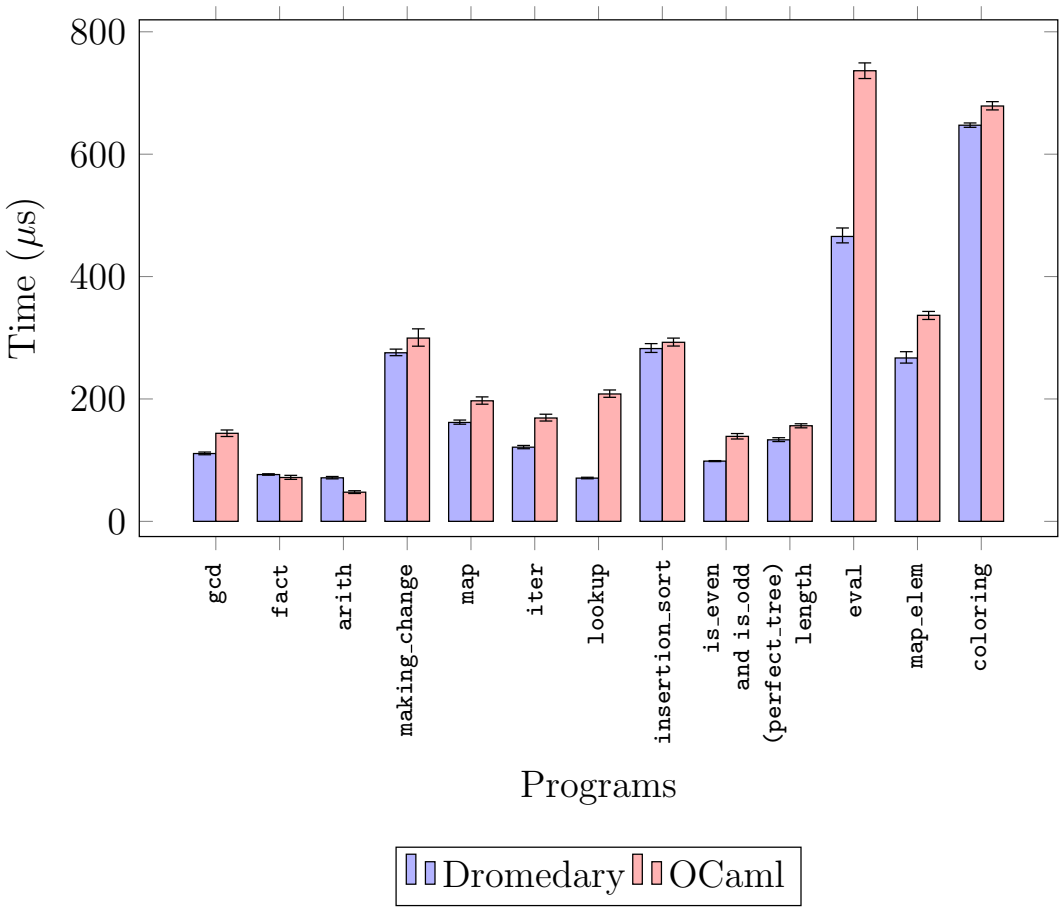
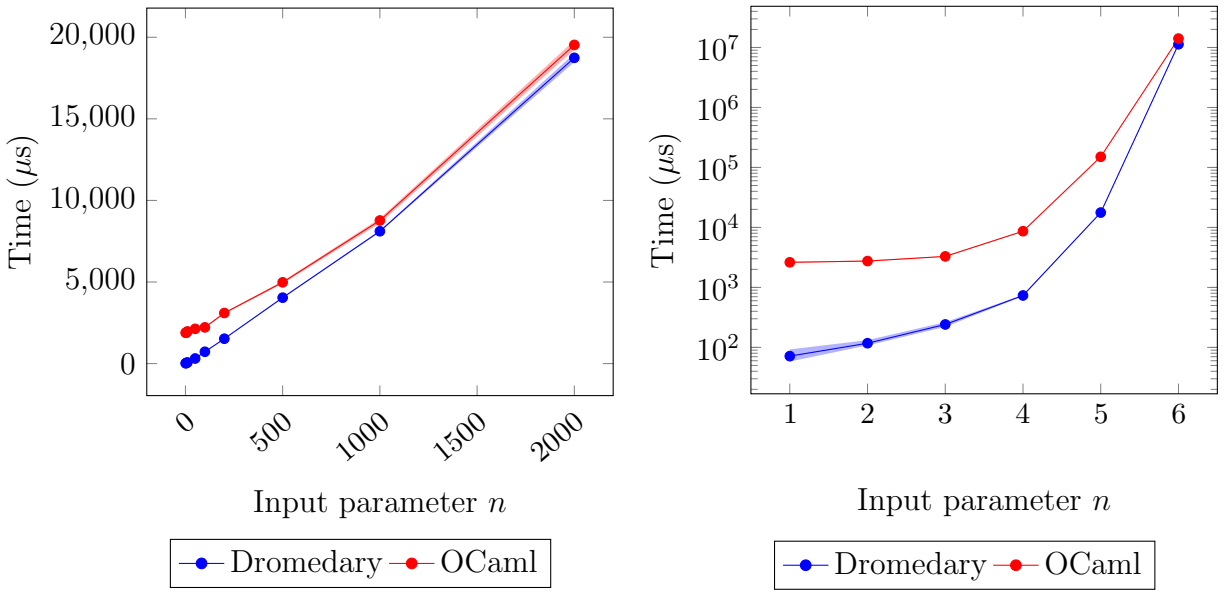


Figure 4.1: Benchmarks of various programs using 10000 trials. A subset from the corpus is used for permissiveness testing. Error bars represent $\pm 2\sigma$.



(a) Inference with exponentially sized types

(b) Inference with exponentially sized type schemes

Figure 4.2: Benchmarks comparing Dromedary and OCaml's asymptotic behaviour in classical exponential cases for ML inference. Shaded areas represent the 95% confidence interval ($\pm 2\sigma$). 10000 trials for (a), 200 trials for (b).

For the benchmark of each feature, we selected random programs from our permissiveness tests. We used programs of our devising to examine the asymptotic behaviour of Dromedary and OCaml.

The benchmarks are automated using the `Core_bench` micro-benchmarking library [3]. Measurements are split into samples, performing linear regression to predict the execution time. The primary source of non-determinism in benchmarks are the effects of garbage collection (GC), which we minimise by ensuring the GC is stabilised between each benchmark. We use a *bootstrapping phase* consisting of 10% of the trials to achieve tight error bounds. Measurements were collected using my personal computer with the following specification:

Processor Intel Core i7-8700 3.20GHz

Memory 16 GB 2133 MHz DDR4 RAM

OS Windows 10 Pro Version 10.0.19043

OCaml Version 4.12.0

Results Figure 4.1 compares the inference times of various programs in Dromedary and OCaml – taken from our permissiveness experiments. Dromedary is usually more performant than OCaml, however, only marginally. Two factors can explain this:

- Dromedary translates programs into *constraints*. Like any intermediate representation, we can optimise constraints – Dromedary employs a variety of *peephole optimisations* on constraints (Section 3.2.2) that aim to minimise costly operations such as *generalisation*.

  This, we believe, explains the significant difference in the timing of `eval`, since ambivalent types heavily rely on generalisation for consistency checking.

  These kinds of optimisations are **not possible** with OCaml's existing type checker; they are a fundamental advantage of a constraint-based approach.

- Dromedary is better optimised due to its more modular approach, specifically in unification and generalisation; using the *union-by-rank* optimisation (Section 3.2.2) and more compact and efficient data structures for generalisation.

  Some of these optimisations are possible with OCaml's existing approach, nevertheless, implementing them would be a technically demanding task owing to the fragility and complexity of the type checker in its present state.

Figure 4.2 compares the asymptotic behaviour of OCaml and Dromedary. *Dromedary consistently outperforms OCaml* in these benchmarks, more noticeably on smaller input parameters of n. However, one may remark that asymptotically they behave comparably. Benchmark (a) measures the inference of the expression:

$$\texttt{let id = fun x -> x in } \underbrace{\texttt{id id } \cdots \texttt{id}}_{n \text{ times}}$$

This expression yields types of exponentially increasing sizes within the *typedtree* representation. However, Dromedary and OCaml both type check the expression in *quasi-linear* time, as seen in Figure 4.2 (a), owing to their use of *sharing* (Section 3.2.2). This corroborates our claim that Dromedary solves constraints in *quasi-linear time*[2].

In benchmark (b), we experiment with exponentially sized *type schemes*, which results in exponential complexity in time, using the expression:

[2] Under certain conditions [30].

```
let pair x f = f x x in
let f0 x = pair x in
let f1 x = f0 (f0 x) in
⋮
let fn x = fn−1 (fn−1 x) in
fun z -> fn (fun x -> x) z
```

Demonstrating that Dromedary and OCaml suffer from the exponential complexity of ML inference [24] when type schemes are unbounded, which no amount of optimisations can prevent.

4.4 Summary

Dromedary **exceeded** all success criteria, achieving all core requirements and extensions listed in Section 2.3. In our benchmarks, **Dromedary outperformed OCaml**, demonstrating the practicality of a constraint-based approach. Additionally, our findings indicate that Dromedary and OCaml share the same asymptotic quasi-linear time complexity for inference[2].

Comparing the permissiveness of Dromedary and OCaml, it was clear from our results that **Dromedary's type system offered equal expressivity** in the implemented features. OCaml and Dromedary programs only differed on minor syntactic features, with all OCaml programs successfully translated into Dromedary programs. Notably, this *suggests* that our type system and constraint-based approach for inference *could be backwards-compatible* with the existing OCaml type checker; however, this is not formally proved. Our experiments into permissiveness also provided empirical evidence towards the correctness of Dromedary's type system and its type checker.

5 Conclusions

The project was a resounding success, surpassing all core success criteria and completing many of the planned extensions.

This project set out to develop a type inference algorithm for a subset of OCaml using a constraint-based approach, designed to address the fragility and unnecessary complexity of the current OCaml type checker.

I introduced Dromedary, a substantial subset of OCaml, whose type system I designed (Section 3.1) based on the PCB type system. I developed an ergonomic constraints language capable of expressing many advanced type system features in OCaml, with *modular* constraint solving and elaboration (Section 3.2).

Dromedary's implementation was designed with *the separation of concerns principle* in mind, which we believe improves clarity, modularity, extensibility and maintainability over the existing OCaml type checker – an original aim of the project.

I established, *experimentally*, that Dromedary is **equally permissive** to OCaml in the implemented features (Section 4.2). Additionally, I demonstrated that **Dromedary outperforms OCaml** (Section 4.3), proving the practicality of a constraint-based approach.

5.1 Future Work

Dromedary and its type system provide several potential avenues for future work. Dromedary could be extended, adding objects [43], modules [28], and other features present in OCaml, such as PPX. Once extended, Dromedary's type checker could be integrated with OCaml's compiler pipeline.

In this dissertation, we formally presented Dromedary and its type system; however, we did not explore any of its metatheoretic properties. Formal proofs of properties, such as the soundness and completeness of constraint generation, are necessary to ensure Dromedary's type system and its inference algorithm are *theoretically* correct.

Additionally, a formal proof does not ensure the correctness of the implementation. Extra work could be done to verify Dromedary's implementation using mechanised proof assistants such as Coq or Agda.

5.2 Lessons Learnt

The timetable proposed in my original project proposal was somewhat optimistic, often underestimating additional term work such as supervisions or unit of assessment examinations. As a result, certain milestones were delayed. Fortunately, I allocated sufficient slack time. However, an improved timetable would have benefited the author.

The early unit tests required explicitly writing the *parsetree* of Dromedary programs. In retrospect, I feel that prioritising the implementation of a lexer and parser would have aided the initial unit testing, thereby advancing the evolution of Dromedary's test suite.

Despite my initial exuberance, adding GADTs to Dromedary proved to be the most challenging milestone to complete, requiring three attempts before our ambivalent types implementation succeeded. On reflection, I believe it would have benefited the project to leave this feature as an extension, replacing it with another, less ambitious, extension in the core deliverable's requirements.

Bibliography

[1] Jeremy Yallop Anil Madhavapeddy. *Foundations of Computer Science (2021-2022) Course Notes*. URL: https://www.cl.cam.ac.uk/teaching/2122/FoundsCS/focs-202122-v1.3.pdf.

[2] Anton Bachin. *The Bisect_ppx code coverage tool*. 2022. URL: https://github.com/aantron/bisect_ppx.

[3] Jane Street Capital. *Core_bench micro-benchmarking framework*. 2022. URL: https://github.com/janestreet/core_bench.

[4] Jane Street Capital. *ppx_let preprocessor*. 2022. URL: https://github.com/janestreet/ppx_let.

[5] Dai Clegg and Richard Barker. *CASE method fast-track - a RAD approach*. Addison-Wesley, 1994. ISBN: 978-0-201-62432-8.

[6] *COCTI: Certificable OCaml Type Inference*. 2022. URL: https://www.math.nagoya-u.ac.jp/~garrigue/cocti/.

[7] *Coveralls.io*. 2022. URL: https://coveralls.io/.

[8] Simon Cruanes. *The QCheck testing framework*. 2022. URL: https://github.com/c-cube/qcheck.

[9] Jana Dunfield and Neelakantan R. Krishnaswami. "Complete and easy bidirectional typechecking for higher-rank polymorphism". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 429–442. DOI: 10.1145/2500365.2500582. URL: https://doi.org/10.1145/2500365.2500582.

[10] Dirk Dussart, Fritz Henglein, and Christian Mossin. "Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time". In: *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings*. Ed. by Alan Mycroft. Vol. 983. Lecture Notes in Computer Science. Springer, 1995, pp. 118–135. DOI: 10.1007/3-540-60360-3_36. URL: https://doi.org/10.1007/3-540-60360-3%5C_36.

[11] Jacques Garrigue. "A certified implementation of ML with structural polymorphism and recursive types". In: *Math. Struct. Comput. Sci.* 25.4 (2015), pp. 867–891. DOI: 10.1017/S0960129513000066. URL: https://doi.org/10.1017/S0960129513000066.

[12] Jacques Garrigue. "Programming with polymorphic variants". In: *ML Workshop*. Vol. 13. 7. Baltimore. 1998.

[13] Jacques Garrigue. "Simple Type Inference for Structural Polymorphism". In: *The Second Asian Workshop on Programming Languages and Systems, APLAS'01, Korea Advanced Institute of Science and Technology, Daejeon, Korea, December 17-18, 2001, Proceedings*. 2001, pp. 329–343.

[14] Jacques Garrigue. "Typing deep pattern-matching in presence of polymorphic variants". In: *JSSST Workshop on Programming and Programming Languages*. Citeseer. 2004. URL: https://caml.inria.fr/pub/papers/garrigue-deep-variants-2004.pdf.

[15] Jacques Garrigue and Jacques Le Normand. "GADTs and Exhaustiveness: Looking for the Impossible". In: *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015*. Ed. by Jeremy Yallop and Damien Doligez. Vol. 241. EPTCS. 2015, pp. 23–35. DOI: 10.4204/EPTCS.241.2. URL: https://doi.org/10.4204/EPTCS.241.2.

[16] Jacques Garrigue and JL Normand. "Adding GADTs to OCaml: the direct approach". In: *Workshop on ML*. 2011.

[17] Jacques Garrigue and Didier Rémy. "Ambivalent Types for Principal Type Inference with GADTs". In: *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. Ed. by Chung-chieh Shan. Vol. 8301. Lecture Notes in Computer Science. Springer, 2013, pp. 257–272. DOI: 10.1007/978-3-319-03542-0_19. URL: https://doi.org/10.1007/978-3-319-03542-0%5C_19.

[18] Jacques Garrigue and Didier Rémy. "Semi-Explicit First-Class Polymorphism for ML". In: *Inf. Comput.* 155.1-2 (1999), pp. 134–169. DOI: 10.1006/inco.1999.2830. URL: https://doi.org/10.1006/inco.1999.2830.

[19] GitHub. *GitHub project board*. 2022. URL: https://docs.github.com/en/issues/organizing-your-work-with-project-boards/managing-project-boards/about-project-boards.

[20] Fritz Henglein. "Type Inference with Polymorphic Recursion". In: *ACM Trans. Program. Lang. Syst.* 15.2 (1993), pp. 253–289. DOI: 10.1145/169701.169692. URL: https://doi.org/10.1145/169701.169692.

[21] Gérard P. Huet. "A Unification Algorithm for Typed lambda-Calculus". In: *Theor. Comput. Sci.* 1.1 (1975), pp. 27–57. DOI: 10.1016/0304-3975(75)90011-0. URL: https://doi.org/10.1016/0304-3975(75)90011-0.

[22] The Open Source Initiative. *The MIT licence*. URL: https://opensource.org/licenses/MIT.

[23] Barry L. Ives. "ML for the Working Programmer by L. C. Paulson (Cambridge University Press, 1996)". In: *ACM SIGSOFT Softw. Eng. Notes* 22.4 (1997), p. 114. DOI: 10.1145/263244.773584. URL: https://doi.org/10.1145/263244.773584.

[24] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. "ML Typability is DEXTIME-Complete". In: *CAAP '90, 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*. Ed. by André Arnold. Vol. 431. Lecture Notes in Computer Science. Springer, 1990, pp. 206–220. DOI: 10.1007/3-540-52590-4_50. URL: https://doi.org/10.1007/3-540-52590-4%5C_50.

[25] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. "Type Reconstruction in the Presence of Polymorphic Recursion". In: *ACM Trans. Program. Lang. Syst.* 15.2 (1993), pp. 290–311. DOI: 10.1145/169701.169687. URL: https://doi.org/10.1145/169701.169687.

[26] Philip A Laplante. *What every engineer should know about software engineering*. CRC Press, 2007.

[27] Konstantin Läufer and Martin Odersky. "Polymorphic Type Inference and Abstract Data Types". In: *ACM Trans. Program. Lang. Syst.* 16.5 (1994), pp. 1411–1430. DOI: 10.1145/186025.186031. URL: https://doi.org/10.1145/186025.186031.

[28] Xavier Leroy. "A modular module system". In: *J. Funct. Program.* 10.3 (2000), pp. 269–303. URL: http://journals.cambridge.org/action/displayAbstract?aid=54525.

[29] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990. URL: https://xavierleroy.org/publi/ZINC.pdf.

[30] David A. McAllester. "Joint RTA-TLCA Invited Talk: A Logical Algorithm for ML Type Inference". In: *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*. Ed. by Robert Nieuwenhuis. Vol. 2706. Lecture Notes in Computer Science. Springer, 2003, pp. 436–451. DOI: 10.1007/3-540-44881-0_31. URL: https://doi.org/10.1007/3-540-44881-0%5C_31.

[31] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *J. Funct. Program.* 18.1 (2008), pp. 1–13. DOI: 10.1017/S0956796807006326. URL: https://doi.org/10.1017/S0956796807006326.

[32] Dale Miller. "Unification Under a Mixed Prefix". In: *J. Symb. Comput.* 14.4 (1992), pp. 321–358. DOI: 10.1016/0747-7171(92)90011-R. URL: https://doi.org/10.1016/0747-7171(92)90011-R.

[33] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: https://doi.org/10.1016/0022-0000(78)90014-4.

[34] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml - Functional Programming for the Masses*. O'Reilly, 2013. ISBN: 978-1-4493-2391-2. URL: http://shop.oreilly.com/product/0636920024743.do%5C#tab%5C_04%5C_2.

[35] Alan Mycroft. "Polymorphic Type Schemes and Recursive Definitions". In: *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 217–228. DOI: 10.1007/3-540-12925-1_41. URL: https://doi.org/10.1007/3-540-12925-1%5C_41.

[36] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. ISBN: 978-0-521-66350-2.

[37] Simon Peyton Jones. *Type inference as constraint solving: how GHC's type inference engine actually works*. Zurihac keynote talk. June 2019. URL: https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/.

[38] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. 2005.

[39] François Pottier. "Hindley-milner elaboration in applicative style: functional pearl". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 203–212. DOI: 10.1145/2628136.2628145. URL: https://doi.org/10.1145/2628136.2628145.

[40] Prabhakar Ragde. "OCaml from the Very Beginning, by John Whitington, Coherent Press, 2013. ISBN-10: 0957671105 (paperback), 204 pp". In: *J. Funct. Program.* 23.3 (2013), pp. 352–354. DOI: 10.1017/S0956796813000087. URL: https://doi.org/10.1017/S0956796813000087.

[41] Didier Rémy. *Extending ML Type System with a Sorted Equational Theory*. Research Report 1766. Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France: Institut National de Recherche en Informatique et Automatisme, 1992. URL: http://gallium.inria.fr/~remy/ftp/eq-theory-on-types.pdf.

[42] Didier Rémy. "Typechecking Records and Variants in a Natural Extension of ML". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 1989, pp. 77–88. DOI: 10.1145/75277.75284. URL: https://doi.org/10.1145/75277.75284.

[43] Didier Rémy and Jerome Vouillon. "Objective ML: An Effective Object-Oriented Extension to ML". In: *Theory Pract. Object Syst.* 4.1 (1998), pp. 27–50.

[44] Didier Rémy and Boris Yakobowski. "From ML to ML^F: graphic type constraints with efficient type inference". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by James Hook and Peter Thiemann. ACM, 2008, pp. 63–74. DOI: 10.1145/1411204.1411216. URL: https://doi.org/10.1145/1411204.1411216.

[45] W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques". In: *Proceedings, 9th International Conference on Software Engineering, Monterey, California, USA, March 30 - April 2, 1987*. Ed. by William E. Riddle, Robert M. Balzer, and Kouichi Kishida. ACM Press, 1987, pp. 328–339. URL: http://dl.acm.org/citation.cfm?id=41801.

[46] Martin Sulzmann et al. *Type inference for GADTs via Herbrand constraint abduction*. Tech. rep. Jan. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.4392.

[47] Robert Endre Tarjan. "Efficiency of a Good But Not Linear Set Union Algorithm". In: *J. ACM* 22.2 (1975), pp. 215–225. DOI: 10.1145/321879.321884. URL: https://doi.org/10.1145/321879.321884.

[48] The Dune Team. *OCaml Dune build system*. 2022. URL: https://dune.build/.

[49] The LexiFi Team. *Landmarks profiling framework*. 2022. URL: https://github.com/LexiFi/landmarks.

[50] The Mirage Team. *Alcotest testing framework*. 2022. URL: https://github.com/mirage/alcotest.

[51] The OCaml Team. *The OCaml Compiler Testsuite*. URL: https://github.com/ocaml/ocaml/tree/trunk/testsuite.

[52] The OCaml Team. *TODO for the OCaml type-checker implementation*. 2020. URL: https://github.com/ocaml/ocaml/blob/4.12.0/typing/TODO.md.

[53] The OPAM Team. *OCaml package manager OPAM*. 2022. URL: https://opam.ocaml.org/.

[54] Mads Tofte and Jean-Pierre Talpin. "Implementation of the Typed Call-by-Value lambda-Calculus using a Stack of Regions". In: *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. Ed. by Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin. ACM Press, 1994, pp. 188–201. DOI: 10.1145/174675.177855. URL: https://doi.org/10.1145/174675.177855.

[55] Philip Wadler. "Comprehending Monads". In: *Math. Struct. Comput. Sci.* 2.4 (1992), pp. 461–493. DOI: 10.1017/S0960129500001560. URL: https://doi.org/10.1017/S0960129500001560.

[56] Hongwei Xi, Chiyan Chen, and Gang Chen. “Guarded recursive datatype constructors”. In: *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. Ed. by Alex Aiken and Greg Morrisett. ACM, 2003, pp. 224–235. DOI: 10.1145/604131.604150. URL: https://doi.org/10.1145/604131.604150.

A Untyped Syntax

Dromedary is a representative[1] subset of OCaml defined by the following grammar:

str	$::= \overline{str\_item;;}$	Structure
str_item	$::=$	Structure item
	$\mid vd$	Value definition
	$\mid$ type td and ... and td	Type declaration
	$\mid$ type $\overline{\alpha}$ T $+= constr\_decl \mid \ldots \mid constr\_decl$	Type extension
	$\mid$ external $x : \sigma =$ string literal prefixed by %	Primitive declaration
	$\mid$ exception K [of τ]	Exception declaration
vb	$::=$ (type $\overline{\alpha}$) $p = e$	Value binding
vd	$::=$ let [rec] vb and ... and vb	Value definition
td	$::= \overline{\alpha}$ T td_kind	Type declaration
td_kind	$::=$	Type declaration kind
	$\mid \varepsilon$	Abstract type
	$\mid \ldots$	Open type
	$\mid = \tau$	Type alias
	$\mid = constr\_decl \mid \ldots \mid constr\_decl$	Variant type
	$\mid =$ { $label\_decl$; ... ; $label\_decl$ }	Record type
$constr\_decl$	$::=$	Constructor declaration
	$\mid K$ [of $\overline{\beta}.\tau$] [constraint E]	
$label\_decl$	$::=$	Label declaration
	$\mid \ell : \overline{\beta}.\tau$	
E	$::=$	Equations
	$\mid \tau = \tau$ and ... and $\tau = \tau$	
τ	$::=$	Type
	$\mid \alpha$	Type variable
	$\mid \tau \to \tau$	Function type
	$\mid \overline{\tau}$ T	Applied type constructor
	$\mid \tau \times \ldots \times \tau$	Tuple type
	$\mid$ [ρ]	Polymorphic variant type
	$\mid (\tau)$	Parenthesis
ρ	$::=$	Rows
	$\mid (<\mid \varepsilon \mid >)$ $\overline{\text{`}K \text{ [of } \tau]}$	

[1] The source syntax differs only notationally

σ ::=		Scheme
	$\mid \overline{\alpha}.\tau$	
c ::=		Constant
	$\mid$ true	
	$\mid$ false	
	$\mid$ ()	Unit
	$\mid$ string literal, e.g. "Hello World"	
	$\mid$ float literal, e.g. 3.14 or .25	
	$\mid$ int literal, e.g. 42 or -12	
e ::=		Expression
	$\mid x$	Variable
	$\mid c$	Constant
	$\mid$ fun $p \to e$	Function
	$\mid e\ e$	Function application
	$\mid vd$ in e	Let binding
	$\mid (e)$	Parenthesis
	$\mid uop\ e$	Unary operator primitive
	$\mid e\ bop\ e$	Binary operator primitive
	$\mid$ if e then e else e	If
	$\mid$ forall (type $\overline{\alpha}$) $\to e$	Universal quantifier
	$\mid$ exists (type $\overline{\alpha}$) $\to e$	Existential quantifier
	$\mid (e : \tau)$	Annotation
	$\mid \{\ \ell = e\ ;\ \ \ldots\ ;\ \ell = e\ \}$	Record
	$\mid e.\ell$	Record field access
	$\mid (e, \ldots, e)$	Tuple
	$\mid K\ [e]$	Constructor
	$\mid$ match e with $(h \mid \ldots \mid h)$	Match
	$\mid$ try e with $(h \mid \ldots \mid h)$	Try
	$\mid e;\ e$	Sequence
	$\mid$ for $p = e$ (to $\mid$ downto) e do e done	For loop
	$\mid$ while e do e done	While loop
	$\mid$ ‘$K\ [e]$	Variant
uop ::=		Unary operator
	$\mid$ -	Negation
	$\mid$!	Dereference
	$\mid$ ref	Reference creation
bop ::=		Binary operator
	$\mid +$	Integer addition
	$\mid -$	Integer subtraction
	$\mid \times$	Integer multiplication

$\mid /$	Integer division
$\mid :=$	Assignment

$p ::=$	Pattern
$\mid \_$	Wildcard
$\mid c$	Constant
$\mid x$	Variable
$\mid K\ [p]$	Constructor
$\mid (p, \ldots, p)$	Tuple
$\mid (p : \tau)$	Annotation
$\mid \text{`}K\ [p]$	Variant
$\mid (p)$	Parenthesis
$h ::=$	Case
$\mid p \rightarrow e$	

where:

K	Constructor, e.g. `Nil, Cons`
x	Variable, e.g. `type_decl, exp'`
α, β	Type variable, e.g. `'a, 'b`
T	Type constructor, e.g. `int`, `list`

B Constraints

The complete constraints language is defined by the following grammar:

$C ::=$	Constraint
$\mid \mathsf{true}$	Truth
$\mid \mathsf{false}$	Falsehood
$\mid C \wedge C$	Conjunction
$\mid \exists \alpha.C$	Existential quantification
$\mid \forall \alpha.C$	Universal quantification
$\mid \exists \zeta.C$	Existential ambivalent quantification
$\mid \zeta = \zeta$	Equality
$\mid \psi \subseteq \zeta$	Subset
$\mid \zeta :> \tau$	Ambivalent coercion
$\mid R \implies C$	Rigid implication
$\mid \mathsf{def}\ \Gamma\ \mathsf{in}\ C$	Explicit substitution
$\mid \mathsf{let}\ \Gamma\ \mathsf{in}\ C$	Let binding
$\mid x \leq \zeta$	Variable Instantiation
$\mid \sigma \leq \zeta$	Scheme Instantiation
$\mid \mathsf{def\ rec}\ \Pi\ \mathsf{in}\ C$	Recursive def binding
$\mid \mathsf{let\ rec}\ \Pi\ \mathsf{in}\ C$	Recursive let binding

$R ::=$	Rigid constraint
$\mid \mathsf{true}$	Truth
$\mid R \wedge R$	Conjunction
$\mid \tau = \tau$	Equality

where

$\sigma ::= \forall \overline{\alpha}, \overline{\zeta}.C \Rightarrow \tau$	Constrainted type scheme
$\Delta ::= \cdot \mid \Delta, x : \zeta$	Fragment
$\Gamma ::= \forall \overline{\alpha}, \overline{\zeta}.C \Rightarrow \Delta$	Constrained context
$\pi ::=$	Recursive binding
$\mid x : \forall \overline{\alpha}, \overline{\zeta}.C \Rightarrow \zeta$	Inferred binding
$\mid x : \forall \overline{\alpha}.C \Leftarrow \tau$	Checked binding
$\Pi ::=$	Recursive context
$\mid \cdot$	Empty context
$\mid \Pi, \pi$	Context snoc

$\psi ::=$	Shallow type
$\mid \alpha$	Type variable
$\mid \overline{\zeta}\ \mathsf{F}$	Shallow type former

The Algebra of Types

In this section, we formally discuss the *multi-sorted algebra of types* and their semantic interpretation in Dromedary's constraint language.

The language of types is *sorted*, where the form of types τ is constrained by their *sort* s – for example, the type $\Sigma\ \mathsf{int}$ is invalid since Σ expects a *row type*. The grammar of *sorts* s (or *kinds*) is given by:

$s ::=$	Sorts
$\mid \star$	Type sort
$\mid \mathsf{row}(L)$	Row sort

where $\mathcal{L}$ is the enumerable set of labels and $L \subseteq \mathcal{L}$. The sort $\star$ is for *basic types*, such as int, and $\mathsf{row}(L)$ for *rows types not containing labels in* L.

The grammar of the multi-sorted algebra of types τ and type formers F is defined as:

$\tau ::=$	Types
$\mid \alpha$	Type variable
$\mid \overline{\tau}\ \mathsf{F}$	Applied type former
$\mid \ell^L : \tau :: \tau$	Row cons
$\mid \partial^L \tau$	Row uniform
$\mid \mu\alpha.\tau$	Equi-recursive type

$\mathsf{F} ::=$	Type former
$\mid \mathsf{T}^s$	Constructor former
$\mid \Sigma\cdot$	Variant former

where T denotes a *basic type constructor* – in the dissertation we do not explicitly distinguish *type formers* from *type constructors* as they are often treated the same in many contexts.

Let $\mathcal{S}$ be a *signature* for basic type constructors T, defining an arity function $\mathsf{arity}_{\mathcal{S}}$ mapping type constructors T to their arity $n \in \mathbb{N}$. A *sorting context* Γ is a sequence of bindings of type variables α to sorts s.

Ill-sorted types are prevented using *sorting judgements* of the form $\mathcal{S}; \Gamma \vdash \tau :: s$, read as: *the type* τ *has the sort* s *in the context* Γ *and signature* $\mathcal{S}$. Like *typing rules* they are defined inductively,

as shown below:

$$\frac{}{\mathcal{S};\Gamma \vdash \alpha :: \Gamma(\alpha)}\ (\textsf{Type-var})$$

$$\frac{\mathsf{arity}_{\mathcal{S}}(T) = n \qquad \forall 1 \leq i \leq n.\mathcal{S};\Gamma \vdash \tau_i :: s}{\mathcal{S};\Gamma \vdash \overline{\tau}\ \mathsf{T}^s :: s}\ (\textsf{Type-former-constr})$$

$$\frac{\mathcal{S};\Gamma \vdash \tau :: \mathsf{row}(\emptyset)}{\mathcal{S};\Gamma \vdash \Sigma\ \tau :: \star}\ (\textsf{Type-former-variant})$$

$$\frac{\mathcal{S};\Gamma \vdash \tau_1 : \star \qquad \mathcal{S};\Gamma \vdash \tau_2 :: \mathsf{row}(L \cup \{\ell\}) \qquad L \subseteq_{\text{fin}} \mathcal{L} \setminus \{\ell\}}{\mathcal{S};\Gamma \vdash (\ell^L : \tau_1 :: \tau_2) :: \mathsf{row}(L)}\ (\textsf{Type-row-cons})$$

$$\frac{\mathcal{S};\Gamma \vdash \tau :: \star}{\mathcal{S};\Gamma \vdash \partial^L\ \tau :: \mathsf{row}(L)}\ (\textsf{Type-row-uniform})$$

$$\frac{\mathcal{S};\Gamma, \alpha : s \vdash \tau :: s}{\mathcal{S};\Gamma \vdash \mu\alpha.\tau :: s}\ (\textsf{Type-mu})$$

The superscripts in the algebra of types ensure that symbols are not overloaded and that each symbol has a unique sort or signature; however, we often omit these superscripts for clarity (as in Appendix C).

Our algebra is associated with an equaltional theory E, defined by the following set of axioms:

Commutativity Labels within *row cons* and *row uniform* types may be permuted. That is, for all labels $\ell_1, \ell_2 \in \mathcal{L}$, finite subset of labels $L \subseteq_{\text{fin}} \mathcal{L} \setminus \{\ell_1, \ell_2\}$, and types τ_1, τ_2, τ_3, the following axioms hold:

$$\frac{}{\ell_1^L : \tau_1 :: (\ell_2^{L \cup \{\ell_1\}} : \tau_2 :: \tau_3) = \ell_2^L : \tau_2 :: (\ell_1^{L \cup \{\ell_2\}} : \tau_1 :: \tau_3)}\ (\textsf{Type-eq-comm-row-cons})$$

$$\frac{}{\partial^L\ \tau_1 = \ell_1^L : \tau_1 :: \partial^{L \cup \{\ell_1\}}\ \tau_1}\ (\textsf{Type-eq-comm-row-uniform})$$

Distributivity Basic type constructors T may be *lifted* – for instance $\rho_1 \to \rho_2$ (where ρ_1, ρ_2 are row types) is interpreted as the row type obtained by applying the constructor $\cdot \to \cdot$ point-wise to the row types ρ_1, ρ_2.

As a result of this property, the equational theory has the following axiom:

$$\frac{}{(\ell^L : \tau_1 :: \rho_1, \ldots, \ell^L : \tau_n :: \rho_n)\ \mathsf{T}^{\mathsf{row}(L)} = \ell^L : (\tau_1, \ldots, \tau_n)\ T^\star :: (\rho_1, \ldots, \rho_n)\ \mathsf{T}^{\mathsf{row}(L \cup \{\ell\})}}\ (\textsf{Type-eq-distrib})$$

for any $\ell \in \mathcal{L}$, and $L \subseteq_{\text{fin}} \mathcal{L} \setminus \{\ell\}$.

Equi-recursive equivalences Equi-recursive types may be *folded* and *unfolded* infinitely:

$$\frac{}{\mu\alpha.\tau = \{\mu\alpha.\tau/\alpha\}\tau}\ (\textsf{Type-eq-equi-fold/unfold})$$

$$\frac{\tau_1 = \{\tau_1/\alpha\}\tau \qquad \tau_2 = \{\tau_2/\alpha\}\tau}{\tau_1 = \tau_2}\ (\textsf{Type-eq-equi-uniqueness})$$

Semantics We now formally define the semantic interpretation of types. Informally, the model consists of graphical ground types generated by the grammar. However, the inclusion of rows and equi-recursive types complicates matters.

We describe our graphical types using the notion of paths. A path π is a sequence of integers or labels. The empty path is denoted as ϵ, and the concatenation of the path π_1 followed by π_2 is written $\pi_1 \cdot \pi_2$.

A graphical term $\mathbf{t}$ over a signature $\mathcal{S}$ is defined as a non-empty partial function from paths to $\mathcal{S}$ that is prefix-closed and well-sorted. The *subterm* of $\mathbf{t}$ rooted at π, written $\mathbf{t} \setminus \pi$, is the function $\pi' \mapsto \mathbf{t}(\pi \cdot \pi')$. The signature of Dromedary's *graphical types* $\mathcal{S}_{\text{drom}}$ is given by:

Symbol	**Signature / Sort**
T	$\star^{\mathsf{arity}(\mathsf{T})} \to \star$
Σ	$\mathsf{row}(\emptyset) \to \star$
L	$\star^{\lvert\mathcal{L}\setminus L\rvert} \to \mathsf{row}(L)$, where $L \subseteq_{\text{fin}} \mathcal{L}$

Thus, we define a *graphical type* $\boldsymbol{t}$ as a graphical term over the signature $\mathcal{S}_{\text{drom}}$ *with* a finite number of *distinct* subterms[1]. We write $\mathbb{T}$ for the set of graphical types. The set of graphical types of sort s is defined as $\mathbb{T}_s = \{\mathbf{t} \in \mathbb{T} : \mathcal{S}_{\text{drom}} \vdash \mathbf{t} :: s\}$.

The interpretation of a type τ of sort s, under the ground assignment φ (Section 2.1.2), written $\varphi(\tau^s)$ is defined as follows:

$$
\begin{aligned}
\varphi(\alpha^s) &= \varphi(\alpha) \\
\varphi(\overline{\tau}\ \mathsf{T}^\star) &= \mathbf{t} \in \mathbb{T}_\star \\
&\qquad \text{s.t } \mathbf{t}(\epsilon) = \mathsf{T} \\
&\qquad \wedge \forall 1 \le i \le \mathsf{arity}_{\mathcal{S}}(\mathsf{T}).\mathbf{t} \setminus i = \varphi(\tau_i^\star) \\
\varphi\left(\overline{\tau}\ \mathsf{T}^{\mathsf{row}(L)}\right) &= \mathbf{t} \in \mathbb{T}_{\mathsf{row}(L)} \\
&\qquad \text{s.t } \mathbf{t}(\epsilon) = L \\
&\qquad \wedge \forall \ell \in \mathcal{L} \setminus L.\mathbf{t}(\epsilon) = \mathsf{T} \\
&\qquad \wedge \forall \ell \in \mathcal{L} \setminus L, 1 \le i \le \mathsf{arity}_{\mathcal{S}}(\mathsf{T}).\mathbf{t} \setminus (\ell \cdot i) = \varphi(\tau_i^\star) \\
\varphi\left((\Sigma\ \tau)^\star\right) &= \mathbf{t} \in \mathbb{T}_\star \\
&\qquad \text{s.t } \mathbf{t}(\epsilon) = \Sigma \\
&\qquad \wedge \mathbf{t} \setminus 1 = \varphi(\tau^{\mathsf{row}(\emptyset)}) \\
\varphi\left((\partial^L\ \tau)^{\mathsf{row}(L)}\right) &= \mathbf{t} \in \mathbb{T}_{\mathsf{row}(L)} \\
&\qquad \text{s.t } \forall \ell \in \mathcal{L} \setminus L.\mathbf{t} \setminus \ell = \varphi(\tau^\star) \\
\varphi\left((\ell^L : \tau_1 :: \tau_2)^{\mathsf{row}(L)}\right) &= \mathbf{t} \in \mathbb{T}_{\mathsf{row}(L)} \\
&\qquad \text{s.t } \mathbf{t}(\epsilon) = L \\
&\qquad \wedge \mathbf{t} \setminus \ell = \varphi(\tau_1^\star) \\
&\qquad \wedge \forall \ell' \in \mathcal{L} \setminus (L \cup \{\ell\}).\mathbf{t} \setminus \ell' = \varphi\left(\tau_2^{\mathsf{row}(L \cup \{\ell\})}\right) \setminus \ell' \\
\varphi\left((\mu\alpha.\tau)^s\right) &= \mathbf{t} \in \mathbb{T}_s \\
&\qquad \text{s.t } \mathbf{t} = (\varphi, \alpha \mapsto \mathbf{t})(\tau^s)
\end{aligned}
$$

[1]This permits cyclic types with a finite encoding.

Type Abbreviations

A *type abbreviation* is a type constructor T with a isomorphism $\overline{\alpha}\ \mathsf{T} \cong \tau_{\mathsf{T}}$, where $\overline{\alpha}$ is a tuple of (disjoint) type variables such that $\mathsf{fv}(\tau_{\mathsf{T}}) \subseteq \overline{\alpha}$.

To reason about equalities in the presence of type abbreviations, we seek to develop rewriting strategies that carry out the 'expansions' of abbreviations.

Head expansion The type abbreviation $\overline{\alpha}\ \mathsf{T} \cong \tau_{\mathsf{T}}$ defines a *rewriting rule* $\mathbf{t}_1 \rhd_{\mathsf{T}} \mathbf{t}_2$ between graphical types, given by:

$$\frac{\mathbf{t}_1(\epsilon) = \mathsf{T} \qquad \mathbf{t}_2 = \{\mathbf{t}_1(i)/\alpha_i : 1 \leq i \leq \mathsf{arity}_{\mathcal{S}}(\mathsf{T})\}(\tau^{\star}_{\mathsf{T}})}{\mathbf{t}_1 \rhd_{\mathsf{T}} \mathbf{t}_2}\ (\textsf{Abbrev-head})$$

Intuitively, the rule defines the expansion of the *head* of $\mathbf{t}_1$, yielding the type $\mathbf{t}_2$.

Contextual expansion Similarly, the rewriting rule $\mathbf{t}_1 \rightsquigarrow_{\mathsf{T}} \mathbf{t}_2$ for the abbreviation $\overline{\alpha}\ \mathsf{T} \cong \tau_{\mathsf{T}}$ is defined as:

$$\frac{\mathbf{t}_1 \setminus \pi \rhd_{\mathsf{T}} \mathbf{t}_2 \setminus \pi}{\mathbf{t}_1 \rightsquigarrow_{\mathsf{T}} \mathbf{t}_2}\ (\textsf{Abbrev-expand})$$

This rewriting rule applies head expansion in some context (or path π) within $\mathbf{t}_1$, resulting in the expansion $\mathbf{t}_2$.

The reflexive transitive closure of $\rightsquigarrow_{\mathsf{T}}$ is denoted $\rightsquigarrow^{*}_{\mathsf{T}}$ and we define the *complete expansion* $\rightsquigarrow^{\infty}_{\mathsf{T}}$ relation by the equivalence: $\mathbf{t}_1 \rightsquigarrow^{\infty}_{\mathsf{T}} \mathbf{t_2}$ if and only if $\mathbf{t}_1 \rightsquigarrow^{*}_{\mathsf{T}} \mathbf{t}_2 \not\rightsquigarrow_{\mathsf{T}}$.

An *abbreviation context* $\mathcal{A}$ is defined as a sequence of type abbreviations $\mathcal{A} ::= \cdot \mid \mathcal{A}, \overline{\alpha}\ \mathsf{T} \cong \tau_{\mathsf{T}}$. We extend our rewriting rules $\rhd, \rightsquigarrow$ to expand any abbreviation in the context $\mathcal{A}$, resulting the relations $\rhd_{\mathcal{A}}$ and $\rightsquigarrow_{\mathcal{A}}$.

Since abbreviations introduce new equivalences, these must be taken into account when resolving equalities between types in constraints. Thus, with the abbreviation context $\mathcal{A}$, equality $=_{\mathcal{A}}$ corresponds to structural equality *modulo* the equivalence relation induced by the expansion of abbreviations in $\mathcal{A}$, that is:

$$\frac{\mathbf{t}_1 \rightsquigarrow^{\infty}_{\mathcal{A}} \mathbf{t} \qquad \mathbf{t}_2 \rightsquigarrow^{\infty}_{\mathcal{A}} \mathbf{t}}{\mathbf{t}_1 =_{\mathcal{A}} \mathbf{t}_2}\ (\textsf{Abbrev-eq})$$

Semantics

Semantically, constraints are interpreted in the *model* $\mathcal{M}$ consisting of:

(i) The set of *graphical types* (henceforth referred to as *ground types*) $\mathbf{t}$ for types τ defined in the above section.

(ii) The set of *ground ambivalent types* $\mathbf{z}$ for ambivalent types, defined as *sets of ground types*:

$$\mathbf{z} ::= \{\mathbf{t}_1, \ldots, \mathbf{t}_n\}$$

Constraints are also interpreted under an implicit *abbreviation context* $\mathcal{A}$.

A *ground assignment* φ is a partial mapping from type variables α to ground types. Similarly, an *ambivalent ground assignment* ϑ is a partial mapping from ambivalent type variables ζ to

ground ambivalent types $\mathbf{z}$. An *environment* ρ is a partial function from term variables x to *sets of ground ambivalent types*.

Implications introduce equalities that must be taken into account when checking the consistency of ground ambivalent types – using an equational context E. A *ground equational context* $E ::= \cdot \mid E, \mathbf{t} = \mathbf{t}$ is a collection of assumed equations between ground types. We write $E \Vdash \mathbf{t}_1 =_{\mathcal{A}} \mathbf{t}_2$, if $\mathbf{t}_1, \mathbf{t}_2$ are contextually equal under the equational context E. Consistency of ambivalent types $E \Vdash \mathbf{z}$ is simply defined as pairwise equality under the equational context:

$$\forall 1 \leq i, j \leq |\mathbf{z}|. E \Vdash \mathbf{t}_i =_{\mathcal{A}} \mathbf{t}_j$$

Coercions $\mathbf{z} :> \mathbf{t}$ are semantically defined by the axiom:

$$\frac{\mathbf{z} =_{\mathcal{A}} \{\mathbf{t}\}}{\mathbf{z} :> \mathbf{t}} \text{ (Coercion)}$$

Intuitively, the axiom states that $\mathbf{z} :> \mathbf{t}$ holds if the ambivalent type $\mathbf{z}$ is *the non-ambiguous type* $\mathbf{t}$. This allows us to *coerce* ambivalent types to types, which is required when embedding ambivalent variables in rigid constraints R during pattern matching.

Satisfiability judgements, defined inductively, take the form $E; \vartheta; \varphi; \rho \Vdash C$, read as: *in the environment ρ, under the equational context E, the assignments ϑ, φ satisfy C*:

$$\frac{}{E; \vartheta; \varphi; \rho \Vdash \mathsf{true}} \text{ (Truth)} \qquad \frac{\forall i \qquad E; \vartheta; \varphi; \rho \Vdash C_i}{E; \vartheta; \varphi; \rho \Vdash C_1 \wedge C_2} \text{ (Conj)}$$

$$\frac{E; \vartheta; \varphi, \alpha \mapsto \mathbf{t}; \rho \Vdash C}{E; \vartheta; \varphi; \rho \Vdash \exists \alpha. C} \text{ (Exists)} \qquad \frac{\forall \mathbf{t} \qquad E; \vartheta; \varphi, \alpha \mapsto \mathbf{t}; \rho \Vdash C}{E; \vartheta; \varphi; \rho \Vdash \forall \alpha. C} \text{ (Forall)}$$

$$\frac{E; \vartheta, \zeta \mapsto \mathbf{z}; \varphi; \rho \Vdash C \qquad E \Vdash \mathbf{z}}{E; \vartheta; \varphi; \rho \Vdash \exists \zeta. C} \text{ (Exists)} \qquad \frac{E, \varphi(R); \vartheta; \varphi; \rho \Vdash C}{E; \vartheta; \varphi; \rho \Vdash R \implies C} \text{ (Implication)}$$

$$\frac{(\vartheta; \varphi)(\psi) \subseteq \vartheta(\zeta)}{E; \vartheta; \varphi; \rho \Vdash \psi \subseteq \zeta} \text{ (Subset)} \qquad \frac{\vartheta(\zeta_1) =_{\mathcal{A}} \vartheta(\zeta_2)}{E; \vartheta; \varphi; \rho \Vdash \zeta_1 = \zeta_2} \text{ (Eq)} \qquad \frac{\vartheta(\zeta) :> \varphi(\tau)}{E; \vartheta; \varphi; \rho \Vdash \zeta :> \tau} \text{ (Coerce)}$$

$$\frac{\vartheta(\zeta) \in \rho(x)}{E; \vartheta; \varphi; \rho \Vdash x \leq \zeta} \text{ (Inst}_0\text{)} \qquad \frac{\vartheta(\zeta) \in (E; \vartheta; \varphi; \rho)(\sigma)}{E; \vartheta; \varphi; \rho \Vdash \sigma \leq \zeta} \text{ (Inst}_1\text{)}$$

$$\frac{E; \vartheta; \varphi; \rho, (E; \vartheta; \varphi; \rho)(\Gamma) \Vdash C}{E; \vartheta; \varphi; \rho \Vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C} \text{ (Def)}$$

$$\frac{E; \vartheta; \varphi; \rho \Vdash \exists \Gamma \qquad E; \vartheta; \varphi; \rho, (E; \vartheta; \varphi; \rho)(\Gamma) \Vdash C}{E; \vartheta; \varphi; \rho \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C} \text{ (Let)}$$

$$\frac{E; \vartheta; \varphi; \rho, (E; \vartheta; \varphi; \rho)(\Pi) \Vdash C}{E; \vartheta; \varphi; \rho \Vdash \mathsf{def\ rec}\ \Pi\ \mathsf{in}\ C} \text{ (Def-rec)}$$

$$\frac{E; \vartheta; \varphi; \rho \Vdash \exists \Pi \qquad E; \vartheta; \varphi; \rho, (E; \vartheta; \varphi; \rho)(\Pi) \Vdash C}{E; \vartheta; \varphi; \rho \Vdash \mathsf{let\ rec}\ \Pi\ \mathsf{in}\ C} \text{ (Let-rec)}$$

where the interpretation of constrained contexts and recursive contexts are given by:

$$(E;\vartheta;\varphi;\rho)(\forall\overline{\alpha},\overline{\zeta}.C \Rightarrow \zeta) = \left\{\vartheta'(\zeta) : \varphi =_{\backslash\overline{\alpha}} \varphi' \wedge \vartheta =_{\backslash\overline{\zeta}} \vartheta' \wedge E;\vartheta';\varphi';\rho \Vdash C\right\}$$

$$(E;\vartheta;\varphi;\rho)(\forall\overline{\alpha},\overline{\zeta}.C \Rightarrow \overline{x_i : \zeta_i}) = \overline{x_i \mapsto (E;\vartheta;\varphi;\rho)(\forall\overline{\alpha},\overline{\zeta}.C \Rightarrow \zeta_i)}$$

$$\exists(\forall\overline{\alpha},\overline{\zeta}.C \Rightarrow \Delta) \simeq \forall\overline{\alpha}.\exists\overline{\zeta}.C$$

$$(E;\vartheta;\varphi;\rho)(x : \forall\overline{\alpha}.C \Leftarrow \tau) = \rho' \quad \text{s.t } \rho'(x) = \{\varphi'(\tau) : \varphi' =_{\backslash\overline{\alpha}} \varphi \wedge E;\vartheta;\varphi;\rho,\rho'[x \mapsto \varphi(\forall\overline{\alpha}.\tau)] \Vdash C\}$$

$$(E;\vartheta;\varphi;\rho)(\overline{x : \forall\overline{\alpha},\overline{\zeta}.C \Rightarrow \zeta}) = \rho' \quad \text{s.t } \rho'(x_i) = (E;\vartheta;\varphi;\rho,\rho')(\forall\overline{\alpha},\overline{\zeta}.\mathsf{def}\ \Delta\ \mathsf{in}\ \bigwedge_i C_i \Rightarrow \Delta)(x_i) \quad \text{where } \Delta = \overline{x_i : \zeta_i}$$

$$\exists(\overline{x : \forall\overline{\alpha}.C \Leftarrow \tau}, \overline{x : \forall\overline{\beta},\overline{\zeta}.D \Rightarrow \zeta}) \simeq \forall\overline{\beta}.\exists\overline{\zeta}.\mathsf{def}\ \overline{x : \forall\overline{\alpha}.\tau}, \overline{x : \zeta}\ \mathsf{in}\ \bigwedge_i C_i \wedge \bigwedge_j D_j$$

Intuitively $\exists\Gamma$ checks whether the constraint C in Γ is satisfiable for all rigid variables $\overline{\alpha}$. Similarly, $\exists\Pi$ checks that all constraints within Π are satisfiable within the recursive context.

C Type System

In this appendix we present the entirety of Dromedary's type system. We begin by formally defining the complete multi-sorted algebra of types τ and type formers F:

$\tau ::=$	Type
$\mid \alpha$	Type variable
$\mid \zeta$	Ambivalent type variable
$\mid \tau \to \tau$	Function type
$\mid \overline{\tau}\ \mathsf{T}$	Applied type constructor
$\mid \tau \times \cdots \times \tau$	Tuple type
$\mid \Sigma\ \tau$	Polymorphic variant type
$\mid \ell : \tau :: \tau$	Row cons
$\mid \partial\tau$	Row uniform
$\mid \mu\alpha.\tau$	Equi-recursive type
$\mid \tau$ where $\alpha = \tau$	Explicit type substitution

$\mathsf{F} ::=$	Type former
$\mid \cdot \to \cdot$	Arrow former
$\mid \mathsf{T}$	Constructor former
$\mid \cdot \times \cdot \times \cdots \times \cdot$	Tuple former
$\mid \Sigma\cdot$	Variant former

where ℓ denotes a *label* and T denotes a *type constructor*. In the context of Dromedary's polymorphic variants, we define labels as $\ell ::= K$. For more details regarding the multi-sorted algebra of types, we refer the reader to Appendix B.

Split types For the translation of types τ into *shallow types* used in constraints, we require the notion of *split types*. Split types ς are a pair $\Xi \rhd \zeta$, where the (deep) type may be reconstructed from the subset constraints in Ξ and variable ζ.

More formally, the grammar of split types ς is given by:

$$\varsigma ::= \Xi \rhd \zeta \qquad \Xi ::= \exists \overline{\zeta}.\Omega \qquad \Omega ::= \cdot \mid \Omega, \zeta \supseteq \psi$$

where ψ is an *shallow type*, defined in Appendix B. As a notational convenience, we write $\Xi \rhd \psi$ for the split type $\exists \zeta.\Xi, \zeta \supseteq \psi \rhd \zeta$. Here is formal translations between split and deep types:

$$\begin{aligned}
\lfloor \alpha \rfloor &= \cdot \rhd \alpha \\
\lfloor \zeta \rfloor &= \exists \cdot . \cdot \rhd \zeta \\
\lfloor \tau_1 \to \tau_2 \rfloor &= \Xi_1 \times \Xi_2 \rhd \zeta_1 \to \zeta_2 && \text{where } \lfloor \tau_i \rfloor = \Xi_i \rhd \zeta_i \\
\lfloor \overline{\tau}\ \mathsf{T} \rfloor &= \Xi_1 \times \cdots \times \Xi_n \rhd \overline{\zeta}\ \mathsf{T} && \text{where } \lfloor \tau_i \rfloor = \Xi_i \rhd \zeta_i \\
\lfloor \tau_1 \times \cdots \times \tau_n \rfloor &= \Xi_1 \times \cdots \times \Xi_n \rhd \zeta_1 \times \cdots \times \zeta_n && \text{where } \lfloor \tau_i \rfloor = \Xi_i \rhd \zeta_i \\
\lfloor \Sigma\ \tau \rfloor &= \Xi \rhd \Sigma\ \zeta && \text{where } \lfloor \tau \rfloor = \Xi \rhd \zeta
\end{aligned}$$

$$\begin{aligned}
\lfloor \ell : \tau_1 :: \tau_2 \rfloor &= \Xi_1, \Xi_2 \rhd \ell : \zeta_1 :: \zeta_2 && \text{where } \lfloor \tau_i \rfloor = \Xi_i \rhd \zeta_i \\
\lfloor \partial\tau \rfloor &= \Xi \rhd \partial\zeta && \text{where } \lfloor \tau \rfloor = \Xi \rhd \zeta \\
\lfloor \mu\alpha.\tau \rfloor &= \exists\zeta.\Xi, \zeta \supseteq \psi \rhd \zeta && \text{where } \lfloor \{\zeta/\alpha\}\tau \rfloor = \Xi \rhd \psi \\
\lfloor \tau_1 \text{ where } \alpha = \tau_2 \rfloor &= \Xi_1 \times \Xi_2 \rhd \zeta_1 && \text{where } \lfloor \tau_2 \rfloor = \Xi_2 \rhd \zeta_2, \lfloor \{\zeta_2/\alpha\}\tau_1 \rfloor = \Xi_1 \rhd \zeta_1
\end{aligned}$$

We extend constraints with a subset constraint for types $\tau \subseteq \zeta$ using *shallow type translations*, such that the following equivalence holds:

$$\tau \subseteq \zeta \;\simeq\; \exists\overline{\zeta}.\bigwedge \Omega \wedge \zeta = \zeta' \qquad \text{where } \lfloor \tau \rfloor = \exists\overline{\zeta}.\Omega \rhd \zeta'$$

Typing Rules

In this section, we present *all* of Dromedary's typing rules.

Structures A structural context Ψ is a sequence of label and constructor bindings, that is:

$$\begin{aligned}
\Psi ::= & && \text{Structural Context} \\
& \mid \cdot && \text{Empty context} \\
& \mid \Psi, K : \forall\overline{\alpha}.\exists\overline{\beta}.R \Rightarrow [\tau \to] \; \overline{\alpha} \; \mathsf{T} && \text{Constructor binding} \\
& \mid \Psi, \ell : \forall\overline{\alpha}.(\forall\overline{\beta}.\tau) \to \overline{\alpha} \; \mathsf{T} && \text{Label binding} \\
& \mid \Psi, \overline{\alpha} \; \mathsf{T} = \tau && \text{Alias}
\end{aligned}$$

We write $\lfloor \Psi \rfloor$ as the abbreviation context $\mathcal{A}$ consisting of abbreviations (or *aliases*) in Ψ. The following table specifies the judgements for structural contexts:

Judgement	**Interpretation**
$\Psi \vdash K :: \forall\overline{\alpha}.\exists\overline{\beta}.R \Rightarrow [\tau \to] \; \overline{\alpha} \; \mathsf{T}$	In Ψ, the constructor K has the associated constructor type scheme $\forall\overline{\alpha}.\exists\overline{\beta}.R \Rightarrow [\tau \to] \; \overline{\alpha} \; \mathsf{T}$ for the type constructor T.
$\Psi \vdash \ell : \forall\overline{\alpha}.(\forall\overline{\beta}.\tau) \to \overline{\alpha} \; \mathsf{T}$	In Ψ, the label ℓ has the associated label type scheme $\forall\overline{\alpha}.(\forall\overline{\beta}.\tau) \to \overline{\alpha} \; \mathsf{T}$ for the type constructor T
$\Psi \vdash \mathsf{T} \; \{ \; \ell_1 \; ; \; \ldots \; ; \; \ell_n \; \}$	In Ψ, the type constructor T is a record type with labels $\ell_1, \ldots, \ell_n$

For a constraint-based formalisation of structures we extend the constraint language with a notion of structural constraints S and definitions D, defined by the grammar:

$$S ::= \overline{D;}; \qquad \text{Structural constraint}$$

$$\begin{aligned}
D ::= & && \text{Definition} \\
& \mid \cdot && \text{Empty definition} \\
& \mid D, D && \text{Conj definition} \\
& \mid \mathsf{def} \; \Gamma && \text{Def binding} \\
& \mid \mathsf{let} \; \Gamma && \text{Let binding}
\end{aligned}$$

$$
\begin{array}{lr}
\mid \mathsf{def\ rec}\ \Pi & \text{Recursive def binding} \\
\mid \mathsf{let\ rec}\ \Pi & \text{Recursive let binding}
\end{array}
$$

$$
\begin{array}{lr}
\Upsilon\ ::= & \text{Multi-context binding} \\
\quad \mid \forall\overline{\alpha}.\exists\overline{\zeta}.C \wedge \mathsf{def}\ \Gamma & \text{Def binding} \\
\quad \mid \mathsf{let}\ \Gamma & \text{Let binding}
\end{array}
$$

where multi-context bindings Υ are required for the *value restriction*.

Using structural contexts and constraints, our structural judgements are of the form $\Psi; S \vdash str$ read as: *under the structural context Ψ and satisfiable structural constraint S (under $\lfloor\Psi\rfloor$), str is well-formed.*

Similarly, for structure items, judgements are of the form $\Psi; D \vdash str\_item \rightsquigarrow \Psi'$ read as: *under structural context Ψ and satisfiable definition D (under $\lfloor\Psi\rfloor$), str_item is well-formed, binding a new structural context Ψ',* given by:

$$\frac{}{\Psi; \cdot \vdash \cdot}\ (\textsf{Dromedary-str-nil})$$

$$\frac{\Psi_0; D \vdash str\_item \rightsquigarrow \Psi_1 \qquad \Psi_1; S \vdash str}{\Psi_0; (D;;S) \vdash str\_item;;str}\ (\textsf{Dromedary-str-cons})$$

$$\frac{\Psi; \overline{\Upsilon} \vdash \overline{vb}}{\Psi; \overline{\Upsilon} \vdash \mathsf{let}\ \overline{vb} \rightsquigarrow \Psi}\ (\textsf{Dromedary-str-item-let})$$

$$\frac{\Psi; \Pi \vdash \overline{vb}}{\Psi; \mathsf{let\ rec}\ \Pi \vdash \mathsf{let\ rec}\ \overline{vb} \rightsquigarrow \Psi}\ (\textsf{Dromedary-str-item-let-rec})$$

$$\frac{}{\Psi; \cdot \vdash \mathsf{type}\ td_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ td_n \rightsquigarrow \Psi, \overline{\overline{td}}}\ (\textsf{Dromedary-str-item-type})$$

$$\frac{\Psi' = \Psi, \overline{K : \forall\overline{\alpha}.[\exists\overline{\beta}.] \bigwedge E \Rightarrow [\tau \rightarrow]\ \overline{\alpha}\ \mathsf{T}}}{\Psi; \cdot \vdash \mathsf{type}\ \overline{\alpha}\ \mathsf{T} \mathrel{+}= \overline{K\ [\mathsf{of}\ \overline{\beta}.\tau]\ [\mathsf{constraint}\ E]} \rightsquigarrow \Psi'}\ (\textsf{Dromedary-str-item-type-ext})$$

$$\frac{}{\Psi; \mathsf{def}\ x : \sigma \vdash \mathsf{external}\ x : \sigma = \text{"\%..."} \rightsquigarrow \Psi}\ (\textsf{Dromedary-str-item-external})$$

$$\frac{}{\Psi; \cdot \vdash \mathsf{exception}\ K\ [\mathsf{of}\ \tau] \rightsquigarrow \Psi, K : \forall\cdot.\exists\cdot.\mathsf{true} \Rightarrow [\tau \rightarrow]\ \mathsf{exn}}\ (\textsf{Dromedary-str-item-exception})$$

Expressions Expression judgements are of the form $C \vdash e : \zeta$, read as: *under the satisfiable assumptions C, the expression e has the ambivalent type ζ.* As in the dissertation (section 3.1.1), we leave the structural context Ψ used within the judgements implicit.

The restriction to ambivalent type variables in the judgement leads to a restricted and explicit type system, thus for a more natural presentation, we permit judgements of the form $C \vdash e : \tau$

and $C \vdash e : \psi$, given by:

$$\frac{C \vdash e : \zeta \qquad \zeta \# \tau}{\exists \zeta . C \wedge \tau \subseteq \zeta \vdash e : \tau} \text{ (Dromedary-exp-tau)}$$

$$\frac{C \vdash e : \zeta \qquad \zeta \# \psi}{\exists \zeta . C \wedge \psi \subseteq \zeta \vdash e : \psi} \text{ (Dromedary-exp-shallow)}$$

For various features in Dromedary's type system discussed in Section 3.1, we expand the constraint language with the following constructs:

$$\Sigma ::= \exists \overline{\alpha} . \forall \overline{\beta} . R \implies \Gamma \qquad \text{Generalized Constrained Context}$$

$$\mathsf{def}\ \exists \overline{\alpha} . \forall \overline{\beta} . R \implies \Gamma\ \mathsf{in}\ C \simeq \exists \overline{\alpha} . \forall \overline{\beta} . R \implies \mathsf{def}\ \Gamma\ \mathsf{in}\ C$$

$$\mathsf{let}\ \exists \overline{\alpha} . \forall \overline{\beta} . R \implies \Gamma\ \mathsf{in}\ C \simeq \exists \overline{\alpha} . \forall \overline{\beta} . R \implies \mathsf{let}\ \Gamma\ \mathsf{in}\ C$$

$$\begin{array}{rll} \Upsilon & ::= & \text{Multi-context binding} \\ & \mid \exists \overline{\alpha} . \forall \overline{\beta} . R \implies \forall \overline{\alpha} . \exists \overline{\zeta} . C \wedge \mathsf{def}\ \Sigma & \text{Def binding} \\ & \mid \mathsf{let}\ \Sigma & \text{Let binding} \end{array}$$

$$uop \leq \zeta \to \zeta \qquad \text{Unary operator instantiation}$$

$$\begin{array}{rl} - \leq \zeta_1 \to \zeta_2 \simeq & \mathsf{int} \subseteq \zeta_1 \wedge \mathsf{int} \subseteq \zeta_2 \\ ! \leq \zeta_1 \to \zeta_2 \simeq & \exists \zeta . \zeta \mathsf{ref} \subseteq \zeta_1 \wedge \zeta = \zeta_2 \\ \mathsf{ref} \leq \zeta_1 \to \zeta_2 \simeq & \exists \zeta . \zeta = \zeta_1 \wedge \zeta\ \mathsf{ref} \subseteq \zeta_2 \end{array}$$

$$bop \leq \zeta \to \zeta \to \zeta \qquad \text{Binary operator instantiation}$$

$$\begin{array}{rl} (+ \mid - \mid / \mid \times) \leq \zeta_1 \to \zeta_2 \to \zeta_3 \simeq & \mathsf{int} \subseteq \zeta_1 \wedge \mathsf{int} \subseteq \zeta_2 \wedge \mathsf{int} \subseteq \zeta_3 \\ := \quad \leq \zeta_1 \to \zeta_2 \to \zeta_3 \simeq & \exists \zeta . \zeta\ \mathsf{ref} \subseteq \zeta_1 \wedge \zeta_2 = \zeta \wedge \mathsf{unit} \subseteq \zeta_3 \end{array}$$

Constructor instantiation

$$K \leq [\zeta_1 \to]\ \zeta_2 \simeq \exists \overline{\zeta_\alpha}, \overline{\zeta_\beta} . \theta R \wedge \overline{\zeta_\alpha}\ \mathsf{T} \subseteq \zeta_2\ [\wedge\ \theta\tau \subseteq \zeta_1] \qquad \text{if } \Psi \vdash K : \forall \overline{\alpha} . \exists \overline{\beta} . R \Rightarrow [\tau \to]\ \overline{\alpha}\ \mathsf{T}$$
$$\text{where } \theta = \{\overline{\zeta_\alpha}/\overline{\alpha}, \overline{\zeta_\beta}/\overline{\beta}\}$$

Label constraints

$$\ell \leq \zeta_1 \to \zeta_2 \simeq \exists \overline{\zeta_\alpha}, \overline{\zeta_\beta} . \{\overline{\zeta_\alpha}/\overline{\alpha}, \overline{\zeta_\beta}/\overline{\beta}\}\tau \subseteq \zeta_1 \wedge \overline{\zeta_\alpha}\ \mathsf{T} \subseteq \zeta_2 \qquad \text{if } \Psi \vdash \ell : \forall \overline{\alpha} . (\forall \overline{\beta} . \tau) \to \overline{\alpha}\ \mathsf{T}$$

$$\ell : (\forall \overline{\beta} . C \Rightarrow \zeta_1) \to \zeta_2 \simeq \exists \overline{\zeta_\alpha} . \overline{\zeta_\alpha}\ \mathsf{T} \subseteq \tau_2 \wedge \forall \overline{\beta} . \{\overline{\zeta_\alpha}/\overline{\alpha}\}\tau \subseteq \zeta_1 \wedge C \qquad \text{if } \Psi \vdash \ell : \forall \overline{\alpha} . (\forall \overline{\beta} . \tau) \to \overline{\alpha}\ \mathsf{T}$$

Variant constraints

$$\zeta \leq \text{`}K_1[\mathsf{of}\ \zeta_1] \mid \ldots \mid \text{`}K_n[\mathsf{of}\ \zeta_n] \simeq \zeta \supseteq \text{`}K_1 : [\zeta_1] :: \ldots :: \text{`}K_n : [\zeta_n] :: \partial\ \mathsf{absent}$$

$$\zeta \geq \text{`}K_1[\mathsf{of}\ \zeta_1] \mid \ldots \mid \text{`}K_n[\mathsf{of}\ \zeta_n] \simeq \exists \zeta_\rho . \zeta \supseteq \text{`}K_1 : [\zeta_1] :: \ldots :: \text{`}K_n : [\zeta_n] :: \zeta_\rho$$

The typing rules are now given by:

$$\frac{C \vDash x \leq \zeta}{C \vdash x : \zeta} \text{ (Dromedary-exp-var)}$$

$$\frac{C \vDash c \leq \zeta}{C \vdash c : \zeta} \text{ (Dromedary-exp-const)}$$

$$\frac{C \vdash p \to e : \zeta_1 \Rightarrow \zeta_2}{C \vdash \mathsf{fun}\ p \to e : \zeta_1 \to \zeta_2} \text{ (Dromedary-exp-fun)}$$

$$\frac{C_1 \vdash e_1 : \zeta_1 \to \zeta_2 \qquad C_2 \vdash e_2 : \zeta_1}{C_1 \wedge C_2 \vdash e_1\ e_2 : \zeta_2} \text{ (Dromedary-exp-app)}$$

$$\frac{\overline{\Upsilon} \vdash \overline{vb} \qquad C \vdash e : \zeta}{\overline{\Upsilon}\ \mathsf{in}\ C \vdash \mathsf{let}\ \overline{vb}\ \mathsf{in}\ e : \zeta} \text{ (Dromedary-exp-let)}$$

$$\frac{\Pi \vdash \overline{vb} \qquad C \vdash e : \zeta}{\mathsf{let\ rec}\ \Pi\ \mathsf{in}\ C \vdash \mathsf{let}\ \overline{vb}\ \mathsf{in}\ e : \zeta} \text{ (Dromedary-exp-let-rec)}$$

$$\frac{C_1 \vdash e : \zeta_1 \qquad C_2 \vDash uop \leq \zeta_1 \to \zeta_2}{C_1 \wedge C_2 \vdash uop\ e : \zeta_2} \text{ (Dromedary-exp-uop)}$$

$$\frac{C_1 \vdash e_1 : \zeta_1 \qquad C_2 \vdash e_2 : \zeta_2 \qquad C_3 \vDash bop \leq \zeta_1 \to \zeta_2 \to \zeta_3}{C_1 \wedge C_2 \wedge C_3 \vdash e_1\ bop\ e_2 : \zeta_3} \text{ (Dromedary-exp-bop)}$$

$$\frac{C_1 \vdash e_1 : \mathsf{bool} \qquad C_2 \vdash e_2 : \zeta \qquad C_3 \vdash e_3 : \zeta}{C_1 \wedge C_2 \wedge C_3 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \zeta} \text{ (Dromedary-exp-ifthenelse)}$$

$$\frac{C_1 \vdash e : \zeta_1 \qquad C_2 \vDash x \leq \zeta_2}{\mathsf{let}\ \forall \overline{\alpha}, \zeta . C_1 \Rightarrow x : \zeta_2\ \mathsf{in}\ C_2 \vdash \mathsf{forall}\ (\mathsf{type}\ \overline{\alpha}) \to e : \zeta_1} \text{ (Dromedary-exp-forall)}$$

$$\frac{C \vdash \{\overline{\zeta}/\overline{\alpha}\}e : \zeta}{\exists \overline{\zeta} . C \vdash \mathsf{exists}\ (\mathsf{type}\ \overline{\alpha}) \to e : \zeta} \text{ (Dromedary-exp-exists)}$$

$$\frac{C \vDash \tau \subseteq \zeta_1 \qquad C \vDash \tau \subseteq \zeta_2 \qquad C \vdash e : \zeta_2}{C \vdash (e : \tau) : \zeta_1} \text{ (Dromedary-exp-annot)}$$

$$\frac{\forall 1 \leq i \leq n. \qquad C_i \vdash \ell = e : \zeta \qquad \Psi \vdash \mathsf{T}\ \{\ \ell_1\ ; \ldots\ ;\ \ell_n\ \}}{\bigwedge_{i=1}^{n} C_i \vdash \{\ \ell_1 = e_1\ ;\ \ldots\ ;\ \ell_n = e_n\ \} : \zeta} \text{ (Dromedary-exp-record)}$$

$$\frac{C \vdash e : \zeta_1}{\ell : \forall \overline{\beta} . C \Rightarrow \zeta_1 \to \zeta_2 \vdash \ell = e : \zeta_2} \text{ (Dromedary-exp-record-field)}$$

$$\frac{C \vDash \ell \leq \zeta_1 \to \zeta_2 \qquad C \vdash e : \zeta_2}{C \vdash e.\ell : \zeta_1} \text{ (Dromedary-exp-field)}$$

$$\frac{\forall 1 \leq i \leq n. \qquad C_i \vdash e_i : \zeta_i}{\bigwedge_{i=1}^{n} C_i \vdash (e_1, \ldots, e_n) : \zeta_1 \times \cdots \times \zeta_n} \text{ (Dromedary-exp-tuple)}$$

$$\frac{C \vDash K \leq [\zeta_1 \to] \ \zeta_2 \qquad [C \vdash e : \zeta_1]}{C \vdash K \ [e] : \zeta_2} \text{ (Dromedary-exp-construct)}$$

$$\frac{C_e \vdash e : \zeta_e \qquad \forall 1 \leq i \leq n. C_i \vdash h : \zeta_e \Rightarrow \zeta}{C_e \wedge \bigwedge_{i=1}^{n} C_i \vdash \mathsf{match}\ e\ \mathsf{with}\ (h_1 \mid \ldots \mid h_n) : \zeta} \text{ (Dromedary-exp-match)}$$

$$\frac{C_e \vdash e : \zeta \qquad \forall 1 \leq i \leq n. C_i \vdash h : \mathsf{exn} \Rightarrow \zeta}{C_e \wedge \bigwedge_{i=1}^{n} C_i \vdash \mathsf{try}\ e\ \mathsf{with}\ (h_1 \mid \ldots \mid h_n) : \zeta} \text{ (Dromedary-exp-try)}$$

$$\frac{C_1 \vdash e_1 : \mathsf{unit} \qquad C_2 \vdash e_2 : \zeta}{C_1 \wedge C_2 \vdash e_1;\ e_2 : \zeta} \text{ (Dromedary-exp-seq)}$$

$$\frac{C_1 \vdash e_1 : \mathsf{int} \qquad C_2 \vdash e_2 : \mathsf{int} \qquad C_3 \vdash e_3 : \mathsf{unit}}{C_1 \wedge C_2 \wedge \mathsf{def}\ i : \mathsf{int}\ \mathsf{in}\ C_3 \vdash \mathsf{for}\ i = e_1\ (\mathsf{to} \mid \mathsf{downto})\ e_2\ \mathsf{do}\ e_3\ \mathsf{done} : \mathsf{unit}} \text{ (Dromedary-exp-for)}$$

$$\frac{C_1 \vdash e_1 : \mathsf{bool} \qquad C_2 \vdash e_2 : \mathsf{unit}}{C_1 \wedge C_2 \vdash \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done} : \mathsf{unit}} \text{ (Dromedary-exp-while)}$$

$$\frac{[C \vdash e : \zeta'] \qquad C \vDash \zeta \geq \text{`}K\ [\mathsf{of}\ \zeta']}{C \vdash \text{`}K\ [e] : \zeta} \text{ (Dromedary-exp-variant)}$$

$$\frac{\begin{array}{l} C \vdash e : \zeta_e \qquad C \vDash \zeta_e \leq \overline{\text{`}K_i\ [\mathsf{of}\ \zeta_i]} \\ \forall 1 \leq i \leq n. C_i \vdash \text{`}K\ [p_i] \to e_i : \text{`}K_i\ [\mathsf{of}\ \zeta_i] \Rightarrow \zeta \end{array}}{C \wedge \bigwedge_{i=1}^{n} C_i \vdash \mathsf{match}\ e\ \mathsf{with}\ \overline{\text{`}K_i\ [p_i] \to e_i} : \zeta} \text{ (Dromedary-exp-var-match-closed)}$$

$$\frac{\begin{array}{l} C \vdash e : \zeta_e \qquad C \vDash \zeta_e \geq \overline{\text{`}K_i\ [\mathsf{of}\ \zeta_i]} \\ \forall 1 \leq i \leq n. C_i \vdash \text{`}K\ [p_i] \to e_i : \text{`}K_i\ [\mathsf{of}\ \zeta_i] \Rightarrow \zeta \\ C_{n+1} \vdash e_{n+1} : \zeta \end{array}}{C \wedge \bigwedge_{i=1}^{n+1} C_i \vdash \mathsf{match}\ e\ \mathsf{with}\ (\overline{\text{`}K_i\ [p_i] \to e_i} \mid \_ \to e_{n+1}) : \zeta} \text{ (Dromedary-exp-var-match-open)}$$

$$\frac{C \vdash e : \zeta_2}{C \wedge \zeta_1 = \zeta_2 \vdash e : \zeta_1} \text{ (Dromedary-exp-eq)}$$

$$\frac{C \vdash e : \zeta_2 \qquad \zeta_1 \neq \zeta_2}{\exists \zeta_1. C \vdash e : \zeta_2} \text{ (Dromedary-exp-exist)}$$

Judgements for non-recursive and recursive value bindings are of form $\Upsilon \vdash vb$ and $\pi \vdash vb$, respectively.

$$\frac{C \vdash e : \zeta}{x : \forall \overline{\alpha}, \mathsf{fav}(C), \zeta. C \Rightarrow \zeta \vdash (\mathsf{type}\ \overline{\alpha})\ x = e} \text{ (Dromedary-vb-rec-mono)}$$

$$\frac{C \vdash e : \tau}{x : \forall\overline{\alpha}.C \Leftarrow \tau \vdash (\mathsf{type}\ \overline{\alpha})\ x : \tau = e}\ (\textsf{Dromedary-vb-rec-poly})$$

$$\frac{C_p \vdash p : \zeta \rightsquigarrow \exists\overline{\alpha}, \overline{\beta}.\Delta \Rightarrow R \qquad C_e \vdash v : \zeta}{\mathsf{let}\ \exists\overline{\alpha}.\forall\overline{\beta}.R \implies \forall\overline{\gamma}, \mathsf{fav}(C_p, C_e, \Delta).C_p \wedge C_e \Rightarrow \Delta \vdash (\mathsf{type}\ \overline{\gamma})\ p = v}\ (\textsf{Dromedary-vb-val})$$

$$\frac{C_p \vdash p : \zeta \rightsquigarrow \exists\overline{\alpha}, \overline{\beta}.\Delta \Rightarrow R \qquad C_e \vdash e : \zeta}{\exists\overline{\alpha}.\forall\overline{\beta}.R \implies \forall\overline{\gamma}.\exists\mathsf{fav}(C_e, C_p, \Delta).C_e \wedge C_p \wedge \mathsf{def}\ \Delta \vdash (\mathsf{type}\ \overline{\gamma})\ p = e}\ (\textsf{Dromedary-vb-nonval})$$

Patterns Judgements for patterns and cases are of the form: $C \vdash p : \tau \rightsquigarrow \Theta$ and $C \vdash p \to e : \zeta_1 \Rightarrow \zeta_2$; interpreted as: *under the satisfiable assumptions C, the pattern p has the type ζ, binding variables in the generalized fragment Θ* and *under the satisfiable assumptions C, the case $p \to e$ matches values of type ζ_1 returning values of type ζ_2*, respectively.

A *generalized fragment* Θ is a tuple, consisting of a context of flexibly bound variables $\overline{\alpha}$ in rigid constraints, existential variables $\overline{\beta}$, a rigid constraint R, and a fragment Δ, written as $\Theta ::= \exists\overline{\alpha}, \overline{\beta}.\Delta \Rightarrow R$.

The addition of flexibly bound (non-ambivalent) variables $\overline{\alpha}$ in Θ is for propagation of type information from instantiation constraints defined below. These constraints involve *coercion constraints* $\zeta :> \alpha$, which ensure our ambivalent types can be coerced to non-ambiguous types.

We redefine constructor instantiation constraints for patterns, since constructor instantiation for patterns semantically differs to instantiation in expressions, using the following equivalences:

$K \leq \exists\overline{\alpha}.\zeta \Rightarrow R$ — Nullary data constructor instantiation

$K \leq \exists\overline{\alpha}.\zeta \Rightarrow R \simeq \exists\overline{\zeta}.\overline{\zeta}\ \mathsf{T} \subseteq \zeta \wedge \bigwedge_i \zeta_i :> \alpha_i$ if $\Psi \vdash K : \forall\overline{\alpha}'.R \Rightarrow \overline{\alpha}'\ \mathsf{T}$

where $\overline{\alpha} = \mathsf{fv}(R)$

$K \leq \exists\overline{\alpha}, \overline{\beta}.\zeta \to \zeta \Rightarrow R$ — Unary data constructor instantiation

$K \leq \exists\overline{\alpha}, \overline{\beta}.\zeta_1 \to \zeta_2 \Rightarrow R \simeq \exists\overline{\zeta}.\{\overline{\zeta}/\overline{\alpha}\}\tau \subseteq \zeta_1 \wedge \overline{\zeta}\ \mathsf{T} \subseteq \zeta_2 \wedge \bigwedge_i \zeta_i :> \alpha_i$ if $\Psi \vdash K : \forall\overline{\alpha}'.\exists\overline{\beta}.R \Rightarrow \tau \to \overline{\alpha}'\ \mathsf{T}$

where $\overline{\alpha} = \mathsf{fv}(R)$

As with expressions, we permit judgements involving types and ambivalent structures, yielding the analogous rules Dromedary-pat-tau and Dromedary-pat-shallow. The typing rules are given by:

$$\frac{C \vdash p \to e : \zeta_1 \Rightarrow \zeta_2}{C : {\text{`}}K\ [p] \to e : {\text{`}}K\ \mathsf{of}\ \zeta_1 \Rightarrow \zeta_2}\ (\textsf{Dromedary-var-case}_1)$$

$$\frac{C \vdash p : \zeta}{C : {\text{`}}K \to e : {\text{`}}K \Rightarrow \zeta}\ (\textsf{Dromedary-var-case}_2)$$

$$\frac{C_1 \vdash p : \zeta_1 \rightsquigarrow \exists \overline{\beta}.\Delta \Rightarrow R \qquad C_2 \vdash e : \zeta_2}{\exists \overline{\alpha}.\forall \overline{\beta}.R \implies \mathsf{let}\ \forall \mathsf{fav}(C_1, \Delta).C_1 \Rightarrow \Delta\ \mathsf{in}\ C_2 \vdash p \to e : \zeta_1 \Rightarrow \zeta_2}\ (\text{Dromedary-case})$$

$$\frac{}{C \vdash \_ : \zeta \rightsquigarrow \exists \cdot .\cdot \Rightarrow \mathsf{true}}\ (\text{Dromedary-pat-wild})$$

$$\frac{}{C \vdash x : \zeta \rightsquigarrow \exists \cdot .x : \zeta \Rightarrow \mathsf{true}}\ (\text{Dromedary-pat-var})$$

$$\frac{C \vDash c \leq \zeta}{C \vdash c : \zeta \rightsquigarrow \exists \cdot .\cdot \Rightarrow \mathsf{true}}\ (\text{Dromedary-pat-const})$$

$$\frac{C \vDash K \leq \exists \overline{\alpha}.\zeta \Rightarrow R}{C \vdash K : \zeta \rightsquigarrow \exists \overline{\alpha}.\cdot \Rightarrow R}\ (\text{Dromedary-pat-construct}_0)$$

$$\frac{C \vDash K \leq \exists \overline{\alpha}, \overline{\beta}.\zeta_1 \to \zeta_2 \Rightarrow R \qquad C \vdash p : \zeta_1 \rightsquigarrow \Theta}{C \vdash K : \zeta_2 \rightsquigarrow \exists \overline{\alpha}, \overline{\beta}.\Theta \Rightarrow R}\ (\text{Dromedary-pat-construct}_1)$$

$$\frac{\forall 1 \leq i \leq n. \qquad C_i \vdash p_i : \zeta_i \rightsquigarrow \Theta_i}{\bigwedge_{i=1}^{n} C_i \vdash (p_1, \ldots, p_n) : \zeta_1 \times \cdots \times \zeta_n \rightsquigarrow \Theta_1 \times \cdots \times \Theta_n}\ (\text{Dromedary-pat-tuple})$$

$$\frac{C \vdash e : \zeta_2 \rightsquigarrow \Theta}{C \wedge \zeta_1 = \zeta_2 \vdash e : \zeta_1 \rightsquigarrow \Theta}\ (\text{Dromedary-pat-eq})$$

$$\frac{C \vdash e : \zeta_2 \rightsquigarrow \Theta \qquad \zeta_1 \neq \zeta_2}{\exists \zeta_1.C \vdash p : \zeta_2 \rightsquigarrow \Theta}\ (\text{Dromedary-pat-exist})$$

D Computations

The *domain-specific language* for *computations* (Section 3.2.3) is embedded with the following signature:

```
module type S = sig
  (** A computation ['a Computation.t] represents a
      monadic computation that produces a value of type ['a].

      Computations are designed for computating (or generating)
      ['a Constraint.t]'s, thus it's syntax provided by [ppx_let]
      is altered (from standard Monadic Let_syntax) for this.

      Computations are bound using the [let%bind] syntax:
      {[
        val comp1 : Typedtree.pattern Constraint.t Computation.t

        let%bind pat1 = comp1 in
        let%bind pat2 = comp1 in
        ...
      ]}
  *)
  type 'a t
  include Monad.S with type 'a t := 'a t

  (** [const x] creates a computation that returns a
      constraint ['a Constraint.t] that evaluates to [x]. *)
  val const : 'a -> 'a Constraint.t t

  (** [fail err] raises the error [err]. *)
  val fail : Sexp.t -> 'a t

  (** [of_result result ~on_error] lifts the result [result]
      into a computation, using [on_error] to compute the error
      message for the computation. *)
  val of_result
    :  ('a, 'err) Result.t
    -> on_error:('err -> Sexp.t)
    -> 'a t

  module Binder : sig
    type 'a computation := 'a t

    (** A ['a Binder.t] represents a monadic binding context for a
        ['b Constraint.t Computation.t]. They are designed to provide
        an intuitive notion of "compositional" binding.

        Computations are bound using the let-op [let&].
```

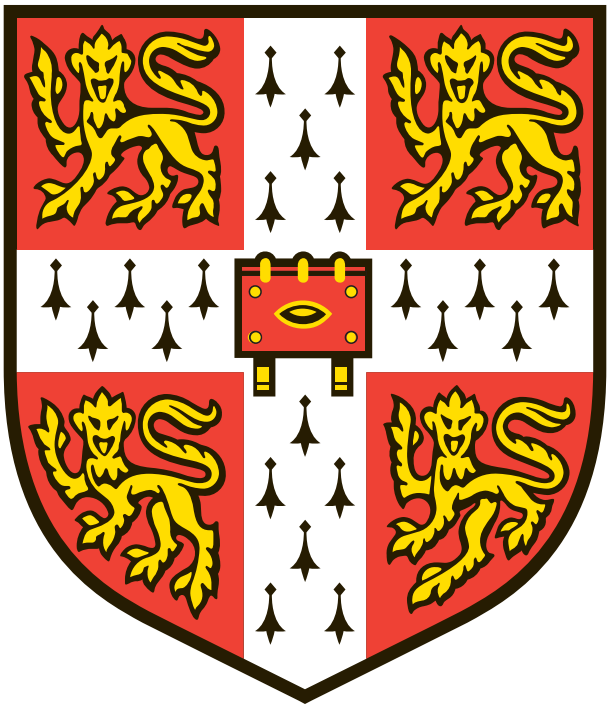
```
  *)
  type 'a t
  include Monad.S with type 'a t := 'a t

  val exists : unit -> Constraint.variable t
  val forall : unit -> Constraint.variable t
  val exists_vars : Constraint.variable list -> unit t
  val forall_ctx : ctx:Constraint.universal_context -> unit t
  val exists_ctx : ctx:Constraint.existential_context -> unit t
  val of_type : Constraint.Type.t -> Constraint.variable t

  module Let_syntax : sig
    val return : 'a -> 'a t
    val ( let& ) : 'a computation -> ('a -> 'b t) -> 'b t
    val ( >>| ) : 'a Constraint.t -> ('a -> 'b) -> 'b Constraint.t

    val ( <*> )
      :  ('a -> 'b) Constraint.t
      -> 'a Constraint.t
      -> 'b Constraint.t

    module Let_syntax : sig
      val return : 'a -> 'a t
      val map : 'a Constraint.t -> f:('a -> 'b) -> 'b Constraint.t
      val both
        :  'a Constraint.t
        -> 'b Constraint.t
        -> ('a * 'b) Constraint.t

      val bind : 'a t -> f:('a -> 'b t) -> 'b t
    end
  end
end

(** [Let_syntax] does not follow the conventional [Let_syntax]
    signature for a Monad. Instead we have standard [return]
    and [bind], however, the [map] and [both] are used
    for constructing constraints.

    This allows the pattern for constructing constraints:
    {[
      let%bind p1 = comp1 in
      let%bind p2 = comp2 in
      return
        (let%map () = var1 =~ var2 in
        ...)
    ]}
    Binders are bound using the let-op [let@].
*)
```

```
  module Let_syntax : sig
    val return : 'a -> 'a t
    val ( let@ )
      :  'a Binder.t
      -> ('a -> 'b Constraint.t t)
      -> 'b Constraint.t t

    val ( >>| )
      :  'a Constraint.t
      -> ('a -> 'b)
      -> 'b Constraint.t

    val ( <*> )
      :  ('a -> 'b) Constraint.t
      -> 'a Constraint.t
      -> 'b Constraint.t

    module Let_syntax : sig
      val return : 'a -> 'a t
      val map : 'a Constraint.t -> f:('a -> 'b) -> 'b Constraint.t
      val both
        :  'a Constraint.t
        -> 'b Constraint.t
        -> ('a * 'b) Constraint.t

      val bind : 'a t -> f:('a -> 'b t) -> 'b t
    end
  end
end
```

E Proposal

Typing OCaml in OCaml: A Constraint-Based Approach

Part II Project Proposal



2377E

Computer Science Tripos

October 18, 2021

Project Originator: 2377E

Project Supervisors: Mistral Contrastin and Dr. Jeremy Yallop
Signatures:

Project Overseers: Prof. Andrew Moore and Andreas Vlachos
Signatures:

Director of Studies:
Signatures:

Introduction

Objective Caml (OCaml) introduced by X. Leroy [7] is a popular and advanced functional programming language based on the ML language – a simple calculus defined by R. Milner [8] offering a restricted form of polymorphism, known as *let-based polymorphism*, with *decidable type inference*.

The *core* language (referred to as Core ML) extends ML with the following features: mutually recursive let-bindings, algebraic data types, patterns, constants, records, mutable references (and the value restriction), exceptions and type annotations. OCaml's major extensions on Core ML consist of first-class and recursive modules, classes and objects, polymorphic variants, semi-explicit first-class polymorphism, generalized algebraic data types (GADTs), the relaxed value restriction, type abbreviations, and labels.

In this project, we will implement a constraint-based type inference algorithm for a subset of the OCaml, provisionally dubbed *Dromedary*, consisting of ML with mutually recursive let-bindings, records, type annotations (a subset of Core ML, provisionally dubbed *Procaml*[1]) and GADTs.

OCaml's inference algorithm is based on algorithm $\mathcal{W}$ [8] with D. Remy's [11] efficient *rank-based generalization* and modifications for the above extensions. While efficient, it has become difficult to maintain and evolve [14]. *Dromedary*'s solution is to re-implement OCaml's type inference using a *constraint-based approach*.

A constraint-based approach would provide a modular implementation of type inference, with separate constraint generation, constraint solving, and type reconstruction phases, using a *small independent* constraint language. Additional advantages include: combining existing constraint-based approaches with OCaml's approaches to increase permissiveness and applications in OCaml's ecosystem.

Previous work to improve OCaml's inference algorithm focuses on incremental changes to the current implementation [14]. Whereas our work is more ambitious and aims to provide the foundation for a complete rewrite – which we believe to be worthwhile.

Despite *Dromedary* being seemingly simple, its inference will suffer from the many challenging issues of GADT type inference, with previous work highlighting that:

- Type Systems with GADTs lack the *principal ("most general") type* property. M. Sulzmann et al. [13] show that programs with GADTs have infinitely many *maximal* types. Hence a *complete* (*unrestricted*) inference algorithm must consider all of these types, adding significant complexity.
- GADT pattern matching introduces local typing constraints, that may result in different branch types. Reconciling these types is difficult.
- GADT programs extensively rely on A. Mycroft's polymorphic recursion [9]. However, F. Henglein [4] and A. J. Kforuy et al. [6] proved that inference with polymorphic recursion is undecidable.

Dromedary addresses these issues via a novel combination of Haskell's OutsideIn [12] and OCaml's ambivalent types [2]. Constraint propagation and ambivalent types equip *Dromedary* with sufficient expressiveness to reconcile differing branch types. *Dromedary* will require type annotations for polymorphic recursion, guaranteeing the decidability of inference.

We will evaluate *Dromedary*'s inference algorithm against OCaml's (with respect to the implemented features), considering aspects such as permissiveness and efficiency.

[1] After *Procamelus*, an extinct genus of camel

Starting Point

I'm familiar with types, having studied Semantics of Programming Languages. I have no previous experience in type inference beyond ML's classical inference algorithms [8]. I have a basic knowledge of constraint solving having studied Prolog and Logic and Proof.

Prior to starting, I have read literature on OCaml's type system to investigate the feasibility of the project. I have practical experience writing OCaml programs from Foundations of Computer Science and extra-circular study. I have practical experience extending the OCaml type checker.

Structure of the Project

The aim of this project is to implement a constraint-based inference algorithm for a subset of OCaml, called *Dromedary*.

A number of design choices have already been made in order to make a concrete plan.

1. *Dromedary*'s type system will be formally defined, using concepts from Semantics of Programming Languages. Its operational semantics is given by a subset of OCaml's semantics.

   GADTs will use a novel combination of Haskell's OutsideIn [12] and OCaml's ambivalent types [2], designed to increase permissiveness.

2. We will design a (first-order) *constraint language* for *Dromedary*. We will then define a mapping (known as *constraint generation* mapping) from candidate typing judgements (e.g. $e : \tau$) to constraints.

3. A constraint solver for constraints C will be defined.

4. Several properties of *Dromedary* will be stated but not proved. These include principal types, decidability, soundness and completeness of inference. We will verify these properties empirically, using tests from the OCaml type checker test suite.

5. *Dromedary*'s inference algorithm will extend F. Pottier's framework [10] for modular and efficient constraint generation, constraint solving, and type reconstruction, implemented in OCaml.

   The first-order unification algorithm for constraint solving will follow Huet [5], using an efficient *union-find* data structure.

This project will follow an incremental structure, focusing on *Procaml* followed by GADTs, at each stage, extending the type system, semantics, constraint language, and constraint solver.

Thus the structure of the project is as follows:

1. An in-depth study in OCaml's type system to ensure I have the correct details before starting work, focusing on [2].

2. An in-depth study of Haskell's OutsideIn [12].

3. Defining *Dromedary*'s type system for *Procaml*.

4. Defining and implement *Dromedary*'s constraints and constraint solving for *Procaml*.

5. Implementing *Dromedary*'s GADTs.
6. Verifying the correctness of *Dromedary*'s type inference algorithm via tests.
7. Performing a qualitative study into the permissiveness of *Dromedary*'s inference algorithm.
8. Benchmarking *Dromedary* against OCaml's current (4.12.0) implementation.

Success Criteria

For the project to be deemed a success, the following must be successfully completed:

1. Design the type system of *Dromedary*. This should support ML with GADTs.
2. Design the constraint language and constraint generation for *Dromedary*.
3. Implement a constraint-based inference algorithm for *Procaml*.
4. Implement constraint-based inference for GADTs.

   We note that this criterion is not restricted by the design choices from Section 3. Hence will be considered a success provided *any* GADT inference is implemented.
5. Evaluate the permissiveness and efficiency of *Dromedary*'s inference against OCaml's current (4.12.0) implementation.

Evaluation

The following is a list of possible extensions to the project:

1. Adding mutable references, exceptions and the value restriction to *Dromedary*.
2. Adding polymorphic variants [1] to *Dromedary*. The implementation will require the notion of subtyping constraints.
3. Adding semi-explicit first-class polymorphism [3] to *Dromedary*. This will use an efficient rank-based approach [11].
4. Proving properties about *Dromedary*'s semantics and inference, including progress, preservation, principal types, and the soundness and completeness of inference.

Timetable and Milestones

Weeks 1 to 2 (7th Oct – 20th Oct)

Proposal Submitted.

Read ahead in the Types course about System F. Read up on advanced constraint-based type inference of ML.

Read and make notes on the following papers [2] and [12].

Formalize the type system for the *Procaml* subset of *Dromedary*.

Milestone: Formalized type system of *Procaml*.

Weeks 3 to 4 (21st Oct – 3rd Nov)

Define the constraint language and its semantics.

Implement *Dromedary*'s constraint solver as a set of constraint rewriting rules. Write some example constraints and verify the solver solves them correctly.

Milestone: Implemented constraint solver for *Dromedary*.

Weeks 5 to 6 (4th Nov – 17th Nov)

Define the mapping from candidate typing judgments to constraints for *Procaml*, the *constraint generation* mapping.

Implement constraint generation and type reconstruction for *Procaml*. Write some *Procaml* programs and verify that their types are correctly inferred.

Milestone: Implemented type inference for *Procaml*.

Weeks 7 to 8 (18th Nov – 1st Dec)

Formalize the semantics of GADTs. Define constraint generation for GADTs.

Implement *Dromedary*'s inference for GADTs.

Milestone: Formalized *Dromedary*'s GADTs!

Milestone: Finish core project implementation.

Weeks 9 to 10 (2nd Dec – 15th Dec)

End of Michaelmas term – start of Christmas holidays.

Slack time to finish off any implementation.

Prepare test cases to evaluate the permissiveness and efficiency of *Dromedary*.

Qualitatively evaluate the permissiveness of *Dromedary*'s inference algorithm.

Weeks 11 to 12 (16th Dec – 29th Dec)

Take time off for Christmas.

Weeks 13 to 14 (30th Dec – 12th Jan)

Take time off for New Year's.

Weeks 15 to 16 (13th Jan – 26th Jan)

End of Christmas holidays – start of Lent term.

Benchmark *Dromedary*'s inference algorithm against the current OCaml (4.12.0) implementation.

Draft the Progress report and presentation and discuss with supervisor ahead of deadline.

Milestone: Finish core project evaluation.

Weeks 17 to 18 (27th Jan – 9th Feb)

Progress report deadline and presentation.

Start work on possible project extensions if time permits. Focus on extension (1) - adding polymorphic variants to *Dromedary*.

Add additional test cases for work completed on extension (1).

Milestone: Finish implementation of extension (1).

Milestone: Complete progress report and presentation.

Weeks 19 to 20 (10th Feb – 23rd Feb)

Extend benchmarking to include polymorphic variants, from extension (1).

If additional time, implement extension (2) - adding semi-explicit first-class polymorphism to *Dromedary*.

Add additional test cases for work completed on extension (2).

Milestone: Finish implementation of extension (2).

Weeks 21 to 22 (24th Feb – 9th Mar)

Extend benchmarking to include semi-explicit first-class polymorphism, from extension (2).

Start writing drafts for Introduction and Preparation chapters.

Milestone: Complete draft of Introduction and Preparation chapters.

Weeks 23 to 24 (10th Mar – 23rd Mar)

End of Lent term – start of Easter holidays.

Begin writing draft Implementation chapter.

Milestone: Complete draft of Implementation chapter.

Weeks 25 to 26 (24th Mar – 6th Apr)

Write-up draft Evaluation chapter, discuss with supervisor.

Additional time allocated to improve evaluation based on feedback.

Finish Conclusions chapter.

Milestone: Complete first draft of dissertation.

Weeks 27 to 28 (7th Apr – 20th Apr)

Revise for Part II exams while awaiting supervisor feedback.

Weeks 29 to 30 (21st Apr – 4th May)

End of Easter holidays – start of Easter term.

Incorporate feedback from supervisor and submit a new draft to supervisor and director of studies.

Slack time for improving code quality, focusing on documentation and code style.

Week 31 to 32 (5th May – 13th May)

Revise for Part II exams.

Milestone (13th May): Submit Dissertation!

Resource Declaration

I will be using my personal computer (3.20GHz i7-8700, 16GB RAM, 1TB SSD) as my primary machine for software development. *I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.*

As a backup, I will use my personal laptop (Razer Blade Stealth 2017 – 1.80 GHz i7-8500U, 16GB RAM, 1TB SSD) and the Computing Service's MCS. I will periodically backup the dissertation and project implementation to Git version control (GitHub).

References

[1] Jacques Garrigue. "Simple Type Inference for Structural Polymorphism". In: *The Second Asian Workshop on Programming Languages and Systems, APLAS'01, Korea Advanced Institute of Science and Technology, Daejeon, Korea, December 17-18, 2001, Proceedings*. 2001, pp. 329–343.

[2] Jacques Garrigue and Didier Rémy. "Ambivalent Types for Principal Type Inference with GADTs". In: *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. Ed. by Chung-chieh Shan. Vol. 8301. Lecture Notes in Computer Science. Springer, 2013, pp. 257–272. DOI: 10.1007/978-3-319-03542-0_19. URL: https://doi.org/10.1007/978-3-319-03542-0%5C_19.

[3] Jacques Garrigue and Didier Rémy. "Semi-Explicit First-Class Polymorphism for ML". In: *Inf. Comput.* 155.1-2 (1999), pp. 134–169. DOI: 10.1006/inco.1999.2830. URL: https://doi.org/10.1006/inco.1999.2830.

[4] Fritz Henglein. "Type Inference with Polymorphic Recursion". In: *ACM Trans. Program. Lang. Syst.* 15.2 (1993), pp. 253–289. DOI: 10.1145/169701.169692. URL: https://doi.org/10.1145/169701.169692.

[5] Gérard P. Huet. "A Unification Algorithm for Typed lambda-Calculus". In: *Theor. Comput. Sci.* 1.1 (1975), pp. 27–57. DOI: 10.1016/0304-3975(75)90011-0. URL: https://doi.org/10.1016/0304-3975(75)90011-0.

[6] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. "Type Reconstruction in the Presence of Polymorphic Recursion". In: *ACM Trans. Program. Lang. Syst.* 15.2 (1993), pp. 290–311. DOI: 10.1145/169701.169687. URL: https://doi.org/10.1145/169701.169687.

[7] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA, 1990. URL: https://xavierleroy.org/publi/ZINC.pdf.

[8] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: https://doi.org/10.1016/0022-0000(78)90014-4.

[9] Alan Mycroft. "Polymorphic Type Schemes and Recursive Definitions". In: *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 217–228. DOI: 10.1007/3-540-12925-1_41. URL: https://doi.org/10.1007/3-540-12925-1%5C_41.

[10] François Pottier. "Hindley-milner elaboration in applicative style: functional pearl". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 203–212. DOI: 10.1145/2628136.2628145. URL: https://doi.org/10.1145/2628136.2628145.

[11] Didier Rémy. *Extending ML Type System with a Sorted Equational Theory*. Research Report 1766. Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France: Institut National de Recherche en Informatique et Automatisme, 1992. URL: http://gallium.inria.fr/~remy/ftp/eq-theory-on-types.pdf.

[12] Tom Schrijvers et al. "Complete and decidable type inference for GADTs". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 2009, pp. 341–352. DOI: 10.1145/1596550.1596599. URL: https://doi.org/10.1145/1596550.1596599.

[13] Martin Sulzmann et al. *Type inference for GADTs via Herbrand constraint abduction*. Tech. rep. Jan. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.4392.

[14] The OCaml Team. *TODO for the OCaml type-checker implementation*. 2020. URL: https://github.com/ocaml/ocaml/blob/4.12.0/typing/TODO.md.