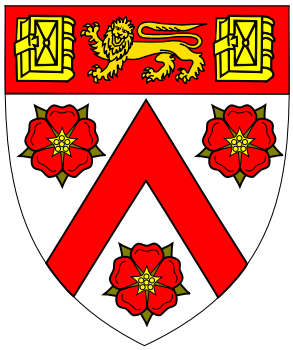
Breaking records

Language design with structural subtyping

Jakub Bachurski



Trinity College
University of Cambridge

June 2025

Submitted in partial fulfilment of the requirements for the
Computer Science Tripos, Part III

Breaking records:
Language design with structural subtyping

Jakub Bachurski

Abstract

Static type systems are the most widely deployed program verification technique. And yet, much of today's programming happens in *dynamically typed languages*, trading away the safety of static typing for flexibility. This thesis argues for designing languages with *structural subtyping*, showing that it mimics the flexibility of dynamically typed languages, while providing safety guarantees like traditional type systems.

In order to inform language design, reliable type inference is identified as key to programmer productivity. Based on Dolan's seminal invention of *algebraic subtyping*, this thesis develops a constraint-based type inference framework supporting structural subtyping. Its expressive power is displayed in the design and implementation of a functional language with extensible data types. Lastly, structural subtyping is shown to lead to a new middle ground between untyped and dependent type systems for array programming, with a novel design of a statically typed array calculus that admits type inference.

This research aims to steer language design towards structural subtyping, and to lead to improvements in optional static type systems for dynamically typed languages.

Acknowledgements

Firstly, I am deeply grateful to my supervisors – Prof Alan Mycroft and Dr Dominic Orchard – for their excellent guidance, constructive advice, and abundant patience.

I'd like to acknowledge the role of many people with whom I discussed ideas that culminated in this thesis – particularly Stephen Dolan, Neel Krishnaswami, Alistair O'Brien, Euan Ong, Jesse Tov, and Leo White – and anyone else who had the patience to listen to my explanations and offered their thoughts.

I am thankful to my parents for always supporting me, and all my friends for making my last year as an undergraduate at Cambridge so memorable.

Contents

1 Introduction

Dynamically typed languages form a large fraction of modern programming [68], and in many cases they are more flexible and enable more rapid development [65]. By leaving the programmer unrestricted by a static type system, dynamic languages are more **expressive**. They admit more programs – many of which are useful, despite their lack of statically-known types guaranteeing safety.

Furthermore, for many domains – such as array programming or data science, both of which are relevant to machine learning – it has become a commonplace assumption that static type systems are too restrictive and get in the way of the programmer. Hence, programming in these domains is virtually always done without static types. Dynamic typing has become not only a matter of preference for higher expressiveness at the cost of safety, but also a **convention**.

1.1 Strategy

The benefits of static typing are clear – they provide safety and a framework for building abstractions. Bestowing static typing on otherwise dynamically typed programs is a useful problem to resolve – as proven in practice by the success of both TypeScript (JavaScript's typed dialect) and Python's optional typing system. Obviously, not all dynamically typed programs can admit static typing under a reasonably complex type system. Hence, we must consider what **characteristics** of dynamically typed programs we are interested in capturing, and what **properties** we desire from our static type system.

Given these desiderata, we can determine an **approach** for statically typing a reasonable subset of programs with the target characteristics. Having found such an approach, we can then inform the design of a statically typed language. Deriving it by statically typing dynamically typed programs, we get the best of both worlds: a sliver of the expressiveness of dynamic languages with a healthy dose of statically ensured safety.

The design of a new language is not in itself useful, but it productively informs the evolution and extension of existing languages. By restricting ourselves to our approach to static typing, we can also identify novel approaches to programming in domains which are traditionally untyped – upending existing conventions for dynamic typing.

1.2 Overview

The thesis follows the strategy proposed in the previous section, and is split into several chapters. Each chapter offers contributions in different areas.

Chapter 2. Structural Subtyping: The Static Soul of Dynamic Languages.

I identify **duck typing** as a powerful pattern characteristically applied in dynamically typed programs. I propose **structural subtyping** as the mechanism for modelling duck typing statically.

In need of a good model for a dynamic language, I introduce **Featherweight Lua** (FL) – a simple λ calculus with extensible records, following the tradition of object calculi [1]. I then give it a static type system with structural subtyping, capturing a *well-behaved* subset of FL.

To show that FL remains adequately powerful when typed statically, I develop a **translation** from the well-known **Featherweight Java** (FJ) [35] into FL. I thus formally show the intuitive notion that we can obtain a structurally typed language by erasing type definitions from a nominally typed one. Indeed, the FJ-FL translation preserves well-typedness – showing that the statically typed fragment of FL is at least as expressive as FJ, even though FL needs none of FJ's explicit class (type) definitions.

To truly model a dynamic language, we need to rid programs of type annotations. This naturally leads to the question of providing *type inference* for FL – and generally, any language with a type system relying on structural subtyping – which I address in Chapter 3.

Chapter 3. Constraint-Based Algebraic Subtyping

By using the **algebraic subtyping** technique invented by Dolan [20], I develop a **type inference framework**, Warp, for languages with structural subtyping. The framework builds on algebraic subtyping – enabling ML-style type inference, combining subtyping with parametric polymorphism and principal types.

Furthermore, I show an extension of algebraic subtyping with **type algebra homomorphisms**. Using them, I show type inference for extensible records (as in FL) without the usual need for row polymorphism.

The description of the type inference approach is *independent* of specific expression and type languages. Instead, it is given in terms of a generic **constraint language** and **solver** and some requirements.

With this framework in hand, I can freely take advantage of its flexibility by designing a new language.

Chapter 4. Design and Implementation of Fabric

I present my **design** of a functional language with structural subtyping – **Fabric** – mainly building on the design of Dolan's MLsub. I focus on designing its type system to take advantage of Warp's algebraic subtyping – enabling type inference for non-trivial language features which mirror dynamic languages.

I also investigate Fabric's **implementation**. I describe my compiler for it – Weaver – which targets WebAssembly. As part of Weaver, I give a prototype implementation of Warp. It can be universally applied to any type language for which we can generate appropriate constraints.

Chapter 5. Structuring Arrays with Algebraic Shapes

Inspired by structural subtyping, I propose a novel statically typed calculus for array programming – **Star** – as an application of the findings of the prior chapters.

The standard in practical array programming has been to forgo types, with dependent type systems proposed as virtually the only options for static typing [47]. I instead propose a middle-ground: **structural types for array shapes and indices**. These types help the programmer build abstractions and avoid menial index arithmetic. At the same time, Star also admits type inference using Warp.

This chapter is based on a paper of the same name, written as part of work on my thesis and coauthored with my supervisors. It was accepted for publication in the Proceedings of the 11th ACM ARRAY Workshop.

Notes for the Reader

The reader may find it useful to consult the index of notation located at the end of this report. Appendices give additional technical details and examples.

2 Structural Subtyping: The Static Soul of Dynamic Languages

This chapter both builds the motivation of this thesis, and introduces many relevant concepts – many of which are covered by Pierce [50].

We begin by exploring characteristics of dynamically typed programs that we might like to type statically (Section 2.1). I propose for **duck typing** to be the key characteristic we should focus on, and argue that **structural subtyping** is a solid approach for statically modelling this pattern. Afterwards, via the introduction of the *FJ-FL translation* in Section 2.3 (between languages introduced in Section 2.2), I ground intuitions about structural subtyping and its expressivity.

2.1 Background

We first consider the meaning of *dynamic* and *static* typing.

Dynamic typing generally means any form of runtime type checking, while *static typing* is type checking at compile time. A dynamically typed language (or program) relies only on dynamic typing, and not static typing. Since when we talk about *typing* we generally mean static typing, we also call dynamically typed programs *untyped*.

Many statically typed languages possess some facilities for runtime type checking, blurring the line between static and dynamic languages:

Static can be dynamic Java (along with many other object-oriented languages) features *downcasts*, which coerce an object of some class into its subclass. This has to be checked at runtime to preserve safety. More glaringly, Java also has reflection facilities, allowing introspection of types of values at runtime.

Dynamic can be static Most dynamically typed languages *could* be given a trivial static type system – for example, where all well-formed expressions are just given some type $\star$.

Furthermore, the line between type checking and other kinds of checks is itself subjective. Verifying that an index into an array is an integer is obviously known to be type checking. On the other hand, determining whether the index is out-of-bounds is generally thought not to be a type check – even though it is handled as such in a dependent type system [5, 47].

Hence, it is difficult to say what dynamic language patterns we should attempt to type statically, and what type safety properties we should ensure. It is necessary to make some assumptions in order to proceed further. After all, the space of dynamically typed programs is extremely large – but there are only so many interesting patterns hiding within.

2.1.1 Taming runtime type checking

Let us distinguish two kinds of runtime type checking – with the goal of identifying a well-behaving subset:

Types as data A value's type can be read and itself operated on as a value *unique to the type*. For example, both Python and Lua feature a `type` built-in function. Since the type becomes a runtime value, it can directly affect data and control flow. Branching on the type (e.g. Python's `isinstance`) – commonly written $(e \in \tau)\ ?\ e : e$ in literature on *set-theoretic types* [15] – treats the type as data.

I exclude the principle of types as data, as it has the following consequences:

Lack of *parametricity* – accessing the type of a value is always legal, i.e. it is of type $\forall\alpha.\ \alpha \to \mathsf{Type}$. However, since the type directly impacts the result of the computation, *parametricity* no longer holds: it is no longer the case a parametrically-polymorphic function performs the same computation in any type instantiation, impeding useful properties (like theorems for free [72]).

Lack of *predictability* – Subjectively, code using types as data is more complicated and confusing to follow. This is reflected in attempts at static analysis – even when we limit ourselves to branching on types, we are computationally limited by *backtracking* [15, 45, 48] (unions, arising in branches $e \in \tau, e \notin \tau$, are type-checked independently).

Types as runtime guardrails A value's type could instead be consulted only when we perform an operation on it, checking whether it is legal – with an error raised otherwise – similarly to how types are used in statically typed languages.

For example, consider a *record* (or *object*) data type, which stores some list of labelled fields, as commonly featured in dynamic languages. We usually consider the labels of fields present to be part of the record's type. Accessing a field of an object requires a (dynamic *or* static) type check for whether it is present in the object.

Viewing types as guardrails – sources of merely runtime type errors, and not information about a value – follows the **duck typing** pattern [41] of untyped programs. If we expect an object to `quack()`, then we worry about nothing else but for our program to `quack()`. This embodies the principle *'ask for forgiveness, not permission'*. Duck typing does not suffer from the loss of parametricity.

I chose to design an approach to static typing which admits **duck-typed** programs – motivated by preservation of parametricity. We now have to find a way to statically model duck typing.

2.1.2 Modelling duck typing

Having identified duck typing as a crucial well-behaved pattern in dynamically-typed programs, it remains to find its *static soul* – a method which admits a corresponding pattern, but can be statically typed.

Structural subtyping

Duck typing leads us to a natural notion of **subtyping**: if a duck a of type A quacks, and another duck b of type B not only quacks but also walks – then clearly b can be used in any place a can. A substitution principle holds: the type B supports more operations than A, and thus B is a subtype of A [50] – we denote this $B \leqslant A$. When speaking of subtyping, we usually refer to its *implicit* variety – the language does not require an *explicit* annotation every time we treat a type as its supertype.

```
(* record *)
type point = { x : int; y : int }
(* variant *)
type opt = None | Some of int
```

(a) Nominally typed records and variants.

```
(* object *)
type point = < x : int; y : int >
(* polymorphic variant *)
type opt = [ `None | `Some of int ]
```

(b) Structurally typed objects and polymorphic variants.

Figure 2.1: Examples of nominal and structural type definitions in OCaml. The same syntax is used for both nominal *type definitions* (Figure 2.1a) and structural *type abbreviations* (Figure 2.1b), even though the two behave differently in OCaml's type system.

This is a good time to introduce a distinction between **nominal** and **structural** (static) typing [50]:

Nominal Most static type systems in popular programming languages are nominal: types are introduced via a type definition, where they are *named* – the type is then identified with (e.g. compared by) this name. For example, the type systems in Java, C/C++, and Haskell are predominantly nominal.

Structural On the other hand, a structural type system does not pose the requirement for a type to have a name nor a definition. A popular example of a structurally typed language would be TypeScript – the statically typed dialect of JavaScript.

Similarly to static and dynamic, nominal and structural typing also lie on a spectrum: few languages feature solely nominal or structural types. For example, OCaml mixes nominal and structural typing (Figure 2.1): its record and variant datatypes can only be introduced in a *nominal* type definition, though they have structural versions introduced in *structural* type abbreviations (objects and polymorphic variants). OCaml modules are structurally typed, too.

Intuitively, a nominally typed program can be transformed into a structurally typed one, and as such structurally typed programs are inherently more flexible – we explore this in Section 2.3. On the other hand, a translation the other way (structural to nominal) duplicates code or introduces explicit coercions.

We also introduce a similar distinction between nominal and structural **subtyping**: while in a language like Java subtyping is defined through the inheritance hierarchy (`class C extends C' {...}`) – on types with specific *names* – structural subtyping is instead defined in terms of the structure of the type. Structural subtyping naturally arises on records (and, similarly, variants) through the following two rules:[1]

Width subtyping A record is a subtype of another if it has more fields and the other are compatible, e.g.:

$$\{\text{foo} : \text{int}, \text{bar} : \text{string}\} \leqslant \{\text{foo} : \text{int}\}$$

Depth subtyping A record is a subtype of another if its respective fields are subtypes, e.g.:

$$\{\text{foo} : \text{nat}, \text{bar} : \text{string}\} \leqslant \{\text{foo} : \text{int}, \text{bar} : \text{string}\}$$

We formalise this in the type system for the record calculus – Featherweight Lua – in Section 2.2.1.

[1] A traditional formal treatment is given by Pierce [50]. In this dissertation – to express it more conveniently for algebraic subtyping in Chapter 3 – we use a slightly different approach with *field types*.

```
let f = function
      | `None -> `Unit
      | `Some x when x mod 2 = 0 -> `Pair (x / 2, x / 2)
      | `Some x -> `Single x
(*
val f : [< `None | `Some of int ]
     -> [> `Pair of int * int | `Single of int | `Unit ]
*)
```

Figure 2.2: Example of row polymorphism in OCaml using polymorphic variants, annotated with its (inferred) most general type. < and > stand for row type variables in closed and open variants, respectively. The argument type can be unified against any variant with at most the listed cases, while the result with at least those.

Clearly, **structural subtyping** is most relevant for modelling duck typing: most dynamically typed programs do not feature type definitions.[2] I am mainly motivated by modelling objects in dynamic languages, following the tradition of Cardelli [9].

Note that we use the name structural subtyping in the sense of subtyping in a structurally typed setting, like Dolan [20] or Cardelli [11] – and not in the sense of the non-structural and structural split (based on type constructor arity) explored by e.g. Su, Aiken, Niehren, Priesnitz and Treinen [69].

I chose structural subtyping as the direction for my thesis. I motivate this choice mainly by the recency – relative to the plethora of work on subtyping from the 90s – of the seminal work of Dolan and Mycroft [21] on *algebraic subtyping*, as it is directly applicable to languages with structural subtyping. It is thus the topic of Chapter 3 and the basis for type systems of languages designed in Chapters 4 and 5. I thus aim to reduce the long-standing distrust in designing languages with implicit subtyping [6].

We briefly consider two possible alternatives. These represent the main areas of related work.

Alternative 1: Row polymorphism

There is a common folklore trick for replacing subtype polymorphism (as for objects/records above) by an appropriate form of parametric polymorphism [70]. For example, a function of type $\top \to \text{int}$ – where $\top$ is top, the supertype of any type – could equivalently be given the type scheme $\forall\alpha.\ \alpha \to \text{int}$, since α can be instantiated to any argument type. **Row polymorphism** [57], introduced by Wand [73] (more generally *structural polymorphism* [26]) – can be seen as an application of this trick to structural record and variant types: we introduce *row type variables* that stand for 'the rest of the record'.

A technical explanation of row polymorphism is outside the scope of this thesis, so we focus on the practical example of OCaml's objects [58] and polymorphic variants [25] – as exemplified in Figure 2.1b. OCaml uses *implicit* row variables [58], which can cause some awkward limitations [16]. It also supports *explicit* (super)type coercions anyhow, adding a form of explicit structural subtyping – managing the limitations of the row polymorphism design. An example application of row polymorphism in OCaml is given in Figure 2.2.

The relative power of structural subtyping and row polymorphism (as extensions to parametric polymorphism) is unclear. There is some recent work exploring the relationship between the two [70, 77]. However, systems with row polymorphism tend to grow more complex than ones with subtyping [16]. Nonetheless, they have historically been used as a common extension of ML-like type systems.

[2] A note-worthy exception would be OOP-style class definitions, present in e.g. Python.

Alternative 2: Set-theoretic types

With **set-theoretic types**, we extend the type language with operators corresponding to set operations:

$$\tau ::= \cdots \mid \tau \cup \tau \mid \tau \cap \tau \mid \neg\tau$$

These operators admit a set-theoretic interpretation – they are compatible with the sets of expressions (values) of a given type. A notion of subtyping is naturally given by set-inclusion. Set-theoretic types are elegant, expressive, and practical [14, 62]. However, they are difficult to efficiently infer and simplify, usually lacking *principality* [15, 16, 48] enjoyed by ML-style type inference – algebraic subtyping included [15, 45].

In algebraic subtyping, we rely on a *type lattice* with *meets* $\wedge$ (least upper bounds) and *joins* $\vee$ (greatest lower bounds) similar to intersections $\cap$ and unions $\cup$. However, they are subtly different [45]: while $(\mathrm{int} \to \mathrm{int}) \cup (\mathrm{nat} \to \mathrm{nat})$ is a unique type, in the type lattice of MLsub [21] & MLstruct [45] we have:

$$(\mathrm{int} \to \mathrm{int}) \vee (\mathrm{nat} \to \mathrm{nat}) = (\mathrm{int} \wedge \mathrm{nat} \to \mathrm{int} \vee \mathrm{nat}) = \mathrm{nat} \to \mathrm{int}$$

While $\cup$ gives us a function type which preserves its argument's type, $\vee$ only grants that the function must accept nat and only returns int (cf. subtyping of functions). Hence, types no longer have a direct set-theoretic interpretation.[3] Thus, set-theoretic types seem to be at least as expressive as algebraic subtyping, where types are constrained to just the ones in the defined lattice. To the author's knowledge, no formal relationship between the two has been established.

2.2 Languages

Having determined my approach to static typing – **structural subtyping** – I propose a pair of model languages to reason about its properties concretely.

Firstly, I construct a simple calculus, which I dub **Featherweight Lua** (FL), modelling a dynamically typed language, and with a statically typed fragment featuring structural subtyping. FL admits a simple embedding into its namesake, the popular dynamically typed language Lua. Secondly, I recall the established **Featherweight Java** (FJ) calculus [35], and use it as a model for a nominally typed OOP language.

2.2.1 Featherweight Lua

I construct FL as the simply-typed lambda calculus extended with extensible record types. Expressions and types in FL are given in Figure 2.3. We consider key aspects in the following subsections.

Expressions

Since FL's functions and bindings are entirely standard, we consider its records and coercions:

Records Records can be constructed from a list of individual assignments, and individual fields can be projected. We have a record extension operator, which, for an existing record lacking a given field, returns a new record with that field set.

Coercion FL features a type coercion operator, which is a no-op at runtime. It is included for completeness of the FJ-FL translation later.

The operational semantics is specified in Appendix A.

[3] Note that type lattices exist where the equation does not hold, and that Dolan's system admits some set-theoretic-esque types like $\{\} \wedge (\top \to \top)$ (though they can only be used as $\top$) where necessary for *extensibility*.

$$
\begin{array}{rll}
e & ::= x & \text{(variable)} \\
& \mid \text{let } x = e \text{ in } e & \text{(let-binding)} \\
& \mid \lambda x.\, e & \text{(function)} \\
& \mid e\, e & \text{(application)} \\
& \mid \{\overline{\ell = e}\} & \text{(record construction)} \\
& \mid \{\ell = e \mid e\} & \text{(record extension)} \\
& \mid e.\ell & \text{(record projection)} \\
& \mid e \rhd \tau & \text{(subtype coercion)}
\end{array}
$$

$$
\begin{array}{rll}
v & ::= \lambda x.\, e & \text{(function)} \\
& \mid \{\ell = v\} & \text{(record)}
\end{array}
$$

(a) Expressions e and values v in Featherweight Lua.

$$
\begin{array}{rll}
\tau & ::= \top & \text{(top)} \\
& \mid \bot & \text{(bottom)} \\
& \mid \tau \to \tau & \text{(function)} \\
& \mid \{\overline{\ell : \phi} \mid \phi\} & \text{(record)}
\end{array}
$$

$$
\begin{array}{rll}
\phi & ::= \mathbb{T} & \text{(top)} \\
& \mid \perp\!\!\!\perp & \text{(bottom)} \\
& \mid \boxed{\tau} & \text{(present)} \\
& \mid \square & \text{(absent)}
\end{array}
$$

$$\{\overline{\ell : \phi_\ell}\} \triangleq \{\overline{\ell : \phi_\ell} \mid \mathbb{T}\}$$

(b) Types τ (def. coinductively [16]) and field types ϕ. For brevity, we sometimes write $\boxed{\tau}$ as τ.

Figure 2.3: Syntax of Featherweight Lua.

Typing

We give the declarative definition of FL's subtyping in Figure 2.4, and typing rules in Figure 2.5. FL has only two type constructors – usual functions and slightly-unusual records. We consider some details.

Recursive types Types τ in FL are generated coinductively, so that we have regular (equi)recursive types (everything else generated inductively as usual). This later allows us to statically type (object-oriented) open recursion – a method calling the object's other methods and accessing its attributes.

Record field types Instead of the syntax for record types, we consider record types $\{\overline{\ell : \phi} \mid \phi'\}$ as functions from labels ℓ into *field types* ϕ, unequal to the *default* ϕ' at finitely many points, as given by the list $\overline{\ell : \phi}$. For example, $\phi = \mathbb{T}$ is the type of a field that might be either absent (fillable by extension; denoted $\square$) or present ($\boxed{\tau}$). This approach is convenient algebraically and easy to generalise to other field types. It is trivially compatible with the usual definition taking just present and top field types.

Record operations A freshly constructed record has all its other fields absent by default, and projection does not constrain the rest of the record (it allows $\mathbb{T}$ fields). Note no record value has a $\perp\!\!\!\perp$ field.

Embedding in Lua

Featherweight Lua is, notionally, a subset of a well-known dynamically typed language: Lua [34]. Lua's core data structure is the *table*, which we replicate – though with strictly less power (without dynamic key access, Lua's 'metatables', etc.) – as FL's records. Lua also features first-class functions. A key distinction between FL and Lua is the former's lack of mutability.

To show the compatibility of FL with its namesake, I sketch a semantics-preserving compositional embedding of FL into Lua in Figure 2.6.

$$\boxed{\tau.\ell = \phi}$$

$$\tau.\ell \triangleq \begin{cases} \phi_\ell, & \ell \text{ occurs in } \tau \\ \phi, & \ell \text{ does not occur in } \tau \end{cases} \qquad \text{where } \tau = \left\{\overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}$$

$$\boxed{\tau \leqslant \tau}$$

$$\textsc{SubFun}\ \frac{\tau_1 \geqslant \tau_2 \quad \tau_1' \leqslant \tau_2'}{\tau_1 \to \tau_1' \leqslant \tau_2 \to \tau_2'} \qquad \textsc{SubRec}\ \frac{\forall \ell.\, \tau_1.\ell \leqslant \tau_2.\ell}{\tau_1 \leqslant \tau_2}$$

$$\boxed{\phi \leqslant \phi}$$

$$\bot\!\!\bot \leqslant \tau \leqslant \top \qquad \frac{\tau' \leqslant \tau''}{\boxed{\tau'} \leqslant \boxed{\tau''}} \qquad \bot\!\!\bot \leqslant \square \leqslant \top$$

Figure 2.4: Declarative definition of the subtyping relation $\leqslant$ in Featherweight Lua, including on field types ϕ. We define projection on record types, $\tau.\ell = \phi$, which we use when deciding record subtyping – algorithmically, we check the occurring labels and defaults.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

$$\textsc{Sub}\ \frac{\Gamma \vdash e : \tau \quad \tau \leqslant \tau'}{\Gamma \vdash e : \tau'} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash v : \tau} \qquad \textsc{Cast}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \rhd \tau' : \tau'}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau'} \qquad \textsc{Fun}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'} \qquad \textsc{Apply}\ \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}$$

$$\textsc{Cons}\ \frac{\overline{\Gamma \vdash e_\ell : \tau_\ell}}{\Gamma \vdash \left\{\overline{\ell = e_\ell}\right\} : \left\{\overline{\ell : \tau_\ell} \mid \square\right\}} \qquad \textsc{Ext}\ \frac{\Gamma \vdash e : \left\{\ell' : \square, \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\} \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \left\{\ell' = e' \mid e\right\} : \left\{\ell' : \tau', \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}} \qquad \textsc{Proj}\ \frac{\Gamma \vdash e : \{\ell : \tau\}}{\Gamma \vdash e : \tau}$$

Figure 2.5: Typing rules for the statically typed fragment of Featherweight Lua.

$$\boxed{[\![e]\!] = \mathsf{e}}$$

$$\begin{aligned}
[\![x]\!] &= x \\
[\![\mathrm{let}\, x = e \,\mathrm{in}\, e]\!] &= x = [\![e]\!];\ [\![e]\!] \\
[\![\lambda x.\, e]\!] &= \mathsf{function}\ (x)\ \mathsf{return}\ [\![e]\!]\ \mathsf{end} \\
[\![e\, e]\!] &= [\![e]\!](e) \\
[\![\{\overline{\ell = e}\}]\!] &= \{\overline{\ell = [\![e]\!]}\} \\
[\![\{\ell = e \mid e'\}]\!] &= \mathsf{update}(\ell, e, e') \\
[\![e.\ell]\!] &= [\![e]\!].\ell \\
[\![e \rhd \tau]\!] &= [\![e]\!]
\end{aligned}$$

```
function update(l, e, t)
    local target = {}
    for k, v in pairs(t) do target[k] = v end
    target[l] = e
end
```

Figure 2.6: Embedding of Featherweight Lua expressions e into Lua programs e. The embedding makes use the use of the functional update for tables, defined in Lua code as part of a preamble.
This is only a sketch of the translation – for instance, any assignments resulting from let-bindings should be extracted as separate statements and sequenced with ';', as Lua does not have binding expressions.

```
class Pair extends Object {
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) {
        super(); this.fst = fst; this.snd = snd;
    }
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }
}
```

Figure 2.7: Example Featherweight Java program implementing a `Pair` class, given by Igarashi, Pierce and Wadler [35]

Untyped Featherweight Lua

By writing FL, we refer to its statically typed part. We also have the dynamically typed (untyped) fragment, dubbed Untyped Featherweight Lua, which admits more potentially useful programs. The following expression e_{untyped} is not well-typed, but evaluates to a value setting e_{cond} to both $\lambda x.\ \lambda y.\ x$ and $\lambda x.\ \lambda y.\ y$:

$$e_{\text{untyped}} = \text{let } x = e_{\text{cond}}\ \{a = \{\}\}\ \{b = \{\}\} \text{ in } e_{\text{cond}}\ (\lambda y.\ y.a)\ (\lambda y.\ y.b)\ x$$

Reasoning about correctness is more difficult than in the well-typed case: it is the fact the same branch is taken by e_{cond} that leads to successful evaluation.

2.2.2 Featherweight Java

Igarashi, Pierce and Wadler [35] introduced Featherweight Java: the *core functional calculus* of Java. It contains only the key features of Java – **classes** and **objects** with **fields** and **methods** – without mutation. To us, FJ is a nominally (statically) typed calculus with object-oriented features and subtype polymorphism. The cited paper offers the formal details. An example FJ program is in Figure 2.7.

Considering structural subtyping, we use FJ in the following thought experiment: what if we *erased* all the type names (FJ classes) in a nominally typed program, inlining their definitions? This process of **anonymising types** intuitively works – and shows that nominally typed programs have within them a structurally typed heart. We focus on the case of languages with subtype polymorphism (FJ and FL) – more complex cases (e.g. like Haskell's higher-kinded polymorphism) can be difficult to combine with structural typing (cf. structural subtyping of ML modules in White, Bour and Yallop [76]). Via the FJ-FL translation in Section 2.3 we will see that, despite its rich classes and objects, FJ is no more expressive than FL.

2.3 Anonymising types: FJ-FL translation

To serve as groundwork for this thesis, I now build a formal understanding of the relationship between the nominal, structural, and dynamic typing disciplines. To this end, we consider a translation between the languages above – from the nominally typed Featherweight Java into the structurally (statically) typed fragment of Featherweight Lua.

The translation follows long-known relationships between object-oriented programming and lambda calculus with records (summarised by e.g. Pierce [51]) – but focusing on nominal and structural typing, rather than semantics.

$$\boxed{(\!|\texttt{K}|\!) = \tau} \qquad \text{(constructor typing)}$$

$$\left(\!\left|\texttt{C(}\overline{\texttt{C f}}\texttt{) \{ super(}\overline{\texttt{f'}}\texttt{); } \overline{\texttt{this.f=f;}}\texttt{ \}}\right|\!\right) = \overline{(\!|\texttt{C}|\!) \to} \, (\!|\texttt{C}|\!)$$

$$\boxed{(\!|\texttt{M}|\!)_{\texttt{C}} = x : \tau} \qquad \text{(method typing)}$$

$$(\!|\texttt{C' m(}\overline{\texttt{C x}}\texttt{)\{ ... \}}|\!) = \texttt{m} : (\!|\texttt{C}|\!) \to \overline{(\!|\texttt{C}|\!) \to} \, (\!|\texttt{C'}|\!)$$

$$\boxed{(\!|\texttt{L}|\!) = x : \tau} \qquad \text{(class definition typing)}$$

$$(\!|\texttt{class C extends C' \{ }\overline{\texttt{C f}}\texttt{; K; }\overline{\texttt{M}}\texttt{; \}}|\!) = \texttt{C} : \left\{\overline{\texttt{f} : (\!|\texttt{C}|\!)}, \overline{(\!|\texttt{M}|\!)}\right\}$$

$$\boxed{(\!|\Gamma_{\texttt{C}}|\!) = \Gamma_{\tau}} \qquad \text{(environment)}$$

$$(\!|\overline{\texttt{x : C}}|\!) = \overline{\texttt{x} : (\!|\texttt{C}|\!)}$$

$$\boxed{[\![\texttt{K}]\!]_{\texttt{C}}^{\texttt{C}'} = e} \qquad \text{(constructor)}$$

$$\left[\!\left[\texttt{C(}\overline{\texttt{C f}}\texttt{) \{ super(}\overline{\texttt{f'}}\texttt{); } \overline{\texttt{this.f=f;}}\texttt{ \}}\right]\!\right]_{\texttt{C}}^{\texttt{C}'} = \text{let}\, \texttt{C} = \overline{\lambda \texttt{f} : (\!|\texttt{C}|\!).} \left\{\overline{\texttt{f} = \texttt{f}}, \texttt{C}_{\text{proto}} \;\middle|\; \texttt{C}' \, \overline{\texttt{f}'}\right\}$$

$$\boxed{[\![\texttt{M}]\!]_{\texttt{C}} = e} \qquad \text{(method)}$$

$$[\![\texttt{C' m(}\overline{\texttt{C x}}\texttt{)\{ return e; \}}]\!]_{\texttt{C}} = \text{let}\, \texttt{m} = \lambda \texttt{this} : (\!|\texttt{C}|\!).\, \overline{\lambda \texttt{x} : (\!|\texttt{C}|\!).}\, [\![\texttt{e}]\!]$$

$$\boxed{[\![\texttt{L}]\!] = e} \qquad \text{(class definition)}$$

$$\begin{aligned}[\![\texttt{class C extends C' \{ }\overline{\texttt{C f}}\texttt{; K; }\overline{\texttt{M}}\texttt{; \}}]\!] &= \text{let}\, \texttt{C} = [\![\texttt{K}]\!]_{\texttt{C}}^{\texttt{C}'} \\ &\quad\; \text{let}\, \texttt{C}_{\text{proto}} = \left\{\overline{[\![\texttt{M}]\!]_{\texttt{C}}} \;\middle|\; \texttt{C}'_{\text{proto}}\right\} \\ \texttt{Object}_{\text{proto}}\, r &= r\end{aligned}$$

$$\boxed{[\![\texttt{e}]\!] = e} \qquad \text{(expression)}$$

$$\begin{aligned}[\![\texttt{e.f}]\!] &= [\![\texttt{e}]\!].\texttt{f} \\ [\![\texttt{e.m(}\overline{\texttt{e}}\texttt{)}]\!] &= \text{let}\, x = [\![\texttt{e}]\!] \,\text{in}\, x.\texttt{m}\, \overline{x\, [\![\texttt{e}]\!]} \\ [\![\texttt{new C(}\overline{\texttt{e}}\texttt{)}]\!] &= \texttt{C}\, \overline{[\![\texttt{e}]\!]} \\ [\![\texttt{(C)e}]\!] &= [\![\texttt{e}]\!] \rhd (\!|\texttt{C}|\!)\end{aligned}$$

Figure 2.8: Translation from Featherweight Java into the statically typed Featherweight Lua, for expressions $[\![-]\!] = e$ and types $(\!|-|\!) = \tau$. Attribute `f`/method `m` names are preserved for use in record fields ℓ. We abuse record extension notation with $\texttt{C}_{\text{proto}}$, as we know its fields. For brevity, we use let $x = e$ and $x : \tau$ for generated fields in record expressions/types.

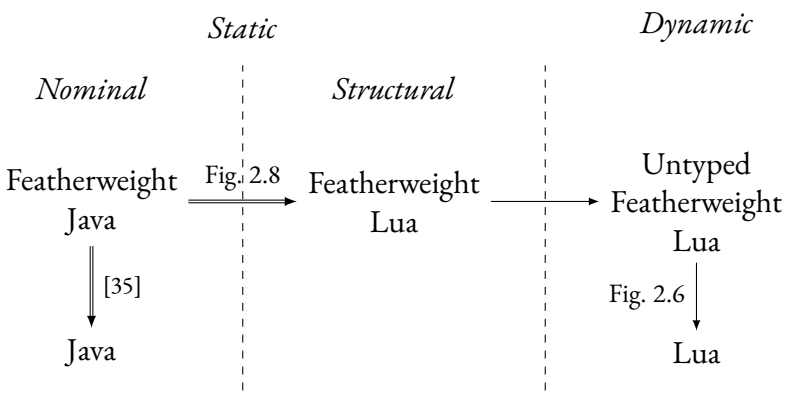


Figure 2.9: Spectrum of nominal, structural, and dynamic typing, spanned by languages connected by translations between them. Its core is the translation of Featherweight Java (nominal) into Featherweight Lua (structural) from Section 2.3. Double arrows ⇒ are translations that preserve both semantics and (non-trivial) types, while single arrows → only preserve semantics.

2.3.1 Construction

The translation is described formally in Figure 2.8, with some notational simplifications for brevity. The FJ notation is the same as in the original paper [35] – its Java-like syntactic constructs should be familiar.

The translation follows directly by considering a **prototype-based** object-oriented programming style: we translate each class into a constructor `C` and the *prototype* $\mathtt{C}_{\text{proto}}$, holding all of the class's methods. Constructing a new class first calls the subclass constructor, and then extends the object with methods from the prototype and the class's fields.

A tricky aspect is that each method must take the object itself – under the name `this` – as an argument. Thankfully, method applications `e.m(`$\overline{\mathtt{e}}$`)` in FJ are syntactically separate, so we can pass the caller object as an argument to its translated method. We note that for this to be well-typed in FL, we require equirecursive types, so that we can express method types (as they both contain and are contained by the class type $(\!|\mathtt{C}|\!)$).

Lastly, note that while FL's type system only allows absent record fields to be extended, this condition is always satisfied – FJ does not allow field shadowing nor method overriding.

2.3.2 Correctness

We now state correctness of the translation.

In these statements, we fix a well-typed class table `CT` (as defined by Igarashi, Pierce and Wadler [35]) – which includes class definitions `L` – necessary both for typing and execution. For brevity, we shall presume that translations $[\![\mathtt{e}]\!]$ implicitly inline definitions from translating `CT` (explicitly written $[\![\mathtt{e}]\!]_{\mathtt{CT}}$).

Theorem 2.1 (Completeness)**.** *If* $[\![\mathtt{e}']\!] = [\![\mathtt{e}'']\!]$*, then* $\mathtt{e}' = \mathtt{e}''$.

Theorem 2.2 (Soundness: Preservation of typing)**.** *If* $\Gamma \vdash \mathtt{e} : \mathtt{C}$*, then* $(\!|\Gamma|\!) \vdash [\![\mathtt{e}]\!] : (\!|\mathtt{C}|\!)$.

Theorem 2.3 (Soundness: Preservation of semantics)**.** *If* $\cdot \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{e} \rightsquigarrow^* \mathtt{v}$*, then* $[\![\mathtt{e}]\!] \rightsquigarrow^* [\![\mathtt{v}]\!]$.

Proof sketches are given in Appendix B.

Note that, as pointed out by Cook, Hill and Canning [19], inheritance is *not* subtyping – it does not follow that $(\!|\mathtt{C}|\!) \leqslant (\!|\mathtt{C}'|\!)$ for any class `C` that `extends C'`.

2.3.3 Consequences

Existence of straightforward FJ-FL translations shows that even very simple structural subtyping is sufficient to capture the essence of classic object-oriented programming, as in (Featherweight) Java – and its nominal approach to static typing. I conclude that structural types are a more flexible typing discipline, as nominal types can be straightforwardly erased from simply-typed programs.

I contextualise this with the fact the structurally typed FL remains useful when we consider its untyped variety (as, in contrast to FJ, it is extrinsically typed [60]). Its statically typed fragment captures some – but not all – programs that do not go wrong.

We conclude that **structural subtyping brings us closer to dynamic languages**. In conjunction, we have placed nominal, structural, and dynamic typing on a spectrum – depicted in Figure 2.9.

It would be interesting to generalise the translation to Generic FJ (FJ with Java-style generics) – a calculus introduced by Igarashi, Pierce and Wadler [35] following Bracha, Odersky, Stoutamire and Wadler [8]. While generic classes seem straightforward, generic methods lead to types with higher-order polymorphism, which would hamper type inference. Indeed, even type checking Java generics is undecidable [28]. This displays the limitation of *anonymising types*: nominal typing makes certain type system features easier than in the structurally typed case.

2.4 Summary

I have identified **duck typing** as a well-behaved pattern in dynamically typed programs, and **structural subtyping** as its appropriate model suitable for constructing expressive static type systems. I motivate this choice by presenting translations and embeddings for a pair of calculi: the nominally typed Featherweight Java, and the structurally typed Featherweight Lua. These formally reinforce intuitive beliefs about the static-dynamic and nominal-structural axes of type systems, and serve as groundwork for the thesis.

Though this dissertation does not explore the application of structural subtyping to statically typing specific dynamic languages (e.g. as part of a gradual type system [66]), it makes contributions towards it, and motivates the application of structural subtyping for this purpose. In the next chapter, I develop a type inference framework for languages featuring structural subtyping – including Featherweight Lua.

3 Constraint-Based Algebraic Subtyping

Convenience of programming with dynamic typing is partially thanks to the absence of type annotations. This comfort comes at a price – we cannot ensure safety guarantees at compile-time. The aim of this thesis is to statically type languages with similar flexibility in mind. As such, we need to recover static types in the absence of annotations – to have *implicit typing* [56]. This is the mission statement of **type inference** [50].

In this chapter, I explore the problem of type inference for languages with structural subtyping. In doing so, I follow Dolan's seminal thesis on **algebraic subtyping** – a type inference technique in the presence of subtyping and (bounded) parametric polymorphism. Specifically, I contribute the following:

- A **constraint-based type inference framework** – Warp – based on the current state-of-the-art in algebraic subtyping. My framework does not fix specific type or expression languages – it sets out some requirements for them (the *signature*) and operates on an intermediate *constraint language*. The constraint-based description is simple and direct, though formal and close to the implementation. The framework is a step towards understanding when we can apply algebraic subtyping in practice.
- An extension of algebraic subtyping that supplements the type language with **type algebra homomorphisms**, described for Warp. Using this extension, I give a method to statically type extensible records using algebraic subtyping – a novel alternative to row polymorphism. Thus, I can infer types for Chapter 2's Featherweight Lua using Warp.

We begin with background on type inference, including our constraint-based setting and a review of the developments in algebraic subtyping (Section 3.1). The rest of the chapter's contents are the description of the framework. Firstly, in Section 3.2 I give a description of Warp's **signature** – the requirements on the source type language. Afterwards, in Section 3.3 I explain **constraint solving** in Warp, and show the homomorphism extension in Section 3.4. Finally, I state and conjecture the correctness theorems of the framework in Section 3.5.

Example

To ease understanding, the technical text of this chapter will be interleaved with boxes like this one, containing examples using Featherweight Lua (with some described extensions).

3.1 Background

Since this chapter concerns describing algebraic subtyping in the framework of constraint-based type inference, I explain these two concepts. We also set up the setting of the type inference problem we consider.

3.1.1 Type inference

Type inference (also called *type reconstruction*), at its core, concerns determining the type τ of a given expression e under an environment Γ – as given by the *typing judgement* $\Gamma \vdash e : \tau$.

Polymorphism & type schemes Type inference is straightforward in a *simply-typed* setting. This is often unsatisfactory – for instance, $\mathrm{id} = \lambda x.\, x$ has type $\tau \to \tau$ for any type τ, but this is inexpressible at the object-level. This leads us to **parametric polymorphism**, where types contain type variables (denoted $\alpha, \beta, \gamma, \ldots$), which can stand for any type. In this setting, expressions are not only given a type τ, but also a **type scheme** σ which can be **instantiated** to a type τ (written $\sigma \models \tau$). We define a judgement $\Gamma \vdash e : \sigma$ for ascribing a type scheme (e.g. $\cdot \vdash \mathrm{id} : \forall \alpha.\, \alpha \to \alpha$), checking the expression is typed at any instantiation:

$$\frac{\forall \tau.\ (\sigma \models \tau \implies \Gamma \vdash e : \tau)}{\Gamma \vdash e : \sigma}$$

The classical solution to type inference under parametric polymorphism is through the Hindley-Milner (HM) type system, which relies on unifications to compute most-general type substitutions [50, 54]. It underlies type systems of languages in the ML family.

In this chapter, we will consider type inference in the presence of $F_{\leqslant}$-style **bounded** parametric polymorphism (in essence, *bounded quantification* of Cardelli and Wegner [13]). Our type schemes σ have a form reminiscent of Java-style generics [8, 44]:[1]

$$\sigma ::= \forall \overline{\tau^+ \leqslant \alpha \leqslant \tau^-}.\, \tau$$

where the body τ of σ must not contain unquantified type variables, and omitted lower/upper bounds are presumed $\bot/\top$. Instantiation $\models$ derives from a type assignment $\psi ::= \cdot \mid \psi, \tau/\alpha$, which satisfies the bounds for each free type variable α. Writing $[\psi]\tau$ for a substitution in τ under ψ, we define:

$$\frac{\overline{[\psi]\tau_\alpha^+ \leqslant [\psi]\alpha \leqslant [\psi]\tau_\alpha^-}}{\left(\forall \overline{\tau_\alpha^+ \leqslant \alpha \leqslant \tau_\alpha^-}.\, \tau\right) \models [\psi]\tau}$$

Example

We extend FL with bounded parametric polymorphism from here onwards, so $\tau ::= \cdots \mid \alpha$. Consider $e = \lambda x.\ x.\mathrm{foo}$. Then we have both

$$\cdot \vdash e : \forall \alpha, (\beta \leqslant \{\mathrm{foo} : \alpha\}).\, \beta \to \alpha \quad \text{and} \quad \cdot \vdash e : \underbrace{\forall \alpha.\, \{\mathrm{foo} : \alpha\} \to \alpha}_{\sigma}$$

and that $\sigma \models \{\mathrm{foo} : \top\} \to \top$ at $\top/\alpha$. Possibility of multiple type schemes points us to *type scheme simplification* – here, we obtain the second type scheme by *inlining* the bound on β in the first. We discuss this important topic in Section 3.3.3.

We allow bounds $\tau_\alpha^+/\tau_\alpha^-$ in type schemes to refer to other type variables, naturally leading us to *recursive types*. Specifically, we will consider type systems with **equirecursive types** – infinite terms in the type language.[2] We will write $\mu\, \alpha.\, \tau$ for a type in which α is equal to the entire type, i.e. $\mu\, \alpha.\, \tau = \mu\, \alpha.\, [\mu\, \alpha.\, \tau/\alpha]\tau$.

Example

We further extend FL with equirecursive μ types. The type $\tau = \mu\, \alpha.\, \top \to \alpha$ corresponds to the infinite type $\top \to (\top \to (\top \to \cdots)))$. Given $\cdot \vdash e : \tau$ and $\cdot \vdash e' : \top$ we have $\cdot \vdash e\, e' : \tau$. Hence, τ is the type of a function that takes an unbounded number of arguments.

[1] Writing τ^+ and τ^- for lower/upper bounds matches the *polarity restriction* (Section 3.1.2). In Warp, τ^+ and τ^- stand for τ.
[2] As opposed to isorecursive types, which are finite and (un)folded explicitly [50].

We define two more type scheme-related concepts: *subsumption* and *principality* (minimality). We define that σ' **subsumes** σ, written $\sigma \leqslant^\forall \sigma'$:

$$\sigma \leqslant^\forall \sigma' \iff \forall \tau.\, (\sigma \models \tau \impliedby \sigma' \models \tau)$$

meaning that σ admits all the types that σ' does. Based on subsumption, we define the **principal** type scheme σ for an expression e as one that subsumes all its other type schemes σ' (it is minimal[3]), i.e.:

$$\sigma \text{ principal} \iff \forall \sigma'.\, \left(\Gamma \vdash e : \sigma' \implies \sigma \leqslant^\forall \sigma'\right)$$

Subsumption can be seen as a generalisation of subtyping to type schemes.

Example

Consider the following type schemes:

$$\sigma^+ = \forall \alpha, \beta.\, \alpha \to \beta \quad \sigma' = \forall \beta.\, \{\} \to \beta \quad \sigma'' = \forall \alpha.\, \alpha \to \{\} \quad \sigma^- = \{\} \to \{\}$$

Then $\sigma^+ \leqslant^\forall \sigma' \leqslant^\forall \sigma^-$ and $\sigma^+ \leqslant^\forall \sigma'' \leqslant^\forall \sigma^-$, but neither $\sigma' \leqslant^\forall \sigma''$ nor $\sigma'' \leqslant^\forall \sigma'$.
Among the four, σ^+ is minimal with respect to subsumption.

Constraint-based approach While type schemes let us describe the *result* of type inference, we can use **constraints** to describe type inference *problems*. To this end, we follow the approach outlined in 'The Essence of ML Type Inference' by Pottier and Rémy [54] (Pierce [49, Chapter 10]).

This chapter will focus on constraint solving in the presence of subtyping, so we give an adequately simple constraint language in Figure 3.1 with only one predicate – subtyping $\tau \leqslant \tau$, where types τ may contain type variables. We also feature constraint conjunction $c \;\&\; c$ and allow introducing existential variables $\exists \alpha.\, c$. The constraint satisfaction judgement $\psi \vdash c$ defined[4] in Figure 3.2 gives a semantics to this syntax, specifying what variable assignments ψ satisfy a given constraint c.

Example

Take $c = \mathbf{T} \;\&\; (\exists \beta.\, \alpha \leqslant (\beta \to \{\}))$. Then:

$$(\{\} \to \{\})/\alpha \vdash c \quad \text{and} \quad \bot/\alpha \vdash c$$

To construct a type inference problem as a constraint c from an expression e in the source language and its expected type τ, we use **constraint generation** $\langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle$. Crucially, it must agree with typing:[5]

$$\psi \vdash \langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle \iff [\psi]\Gamma \vdash [\psi]e : [\psi]\tau$$

Inferring types, we do not know the specific type τ – but we can introduce as a free variable α and take $\langle\!\langle \Gamma \vdash e : \alpha \rangle\!\rangle$. Analogously, if an expression contains 'type holes' (e.g. unannotated function parameters) these can be filled with type variables and constrained appropriately [50].

[3] Taking type schemes σ for which $\Gamma \vdash e : \sigma$, the type scheme minimal under $\leqslant^\forall$ subsumes them all, admitting the most types.
[4] ... in the style of Emrich, Stolarek, Cheney and Lindley [24], with *delayed* substitution: notice $\psi \vdash c \iff \cdot \vdash [\psi]c$.
[5] Some presentations modify the type-scheme judgement to involve constraints (like $c; \Gamma \vdash e : \sigma$) [54] or focus on constraint entailment instead of satisfaction [43]. For simplicity, we work with instantiations ψ, closer to e.g. [24, Section 3.4].

$$
\begin{array}{rll}
c ::= & \mathbf{T} & \text{(always-true)} \\
\mid & \mathbf{F} & \text{(always-false)} \\
\mid & \tau \leqslant \tau & \text{(subtyping)} \\
\mid & c \,\&\, c & \text{(conjunction)} \\
\mid & \exists \alpha.\, c & \text{(existential)}
\end{array}
$$

Figure 3.1: Syntax of constraints c used in this chapter.

$$
\textsc{CTrue}\ \frac{}{\psi \vdash \mathbf{T}} \qquad
\textsc{CSub}\ \frac{[\psi]\tau \leqslant [\psi]\tau'}{\psi \vdash \tau \leqslant \tau'} \qquad
\textsc{CAnd}\ \frac{\psi \vdash c \quad \psi \vdash c'}{\psi \vdash c \,\&\, c'} \qquad
\textsc{CExist}\ \frac{\psi, \tau/\alpha \vdash c}{\psi \vdash \exists \alpha.\, c}
$$

Figure 3.2: Constraint satisfaction judgement $\psi \vdash c$, defined for a type variable assignment ψ and constraint c. Note that $\psi \vdash \mathbf{F}$ is false for any ψ (**F** signals failure – type errors), and $\psi \vdash \mathbf{T}$ is true for any ψ. We use the Barendregt convention to avoid accidental capture in CExist, and abbreviate $\overline{\exists \alpha}$ to $\exists \overline{\alpha}$.

Example

Consider $e = \lambda x : \alpha.\, x.\text{quack}\,\{\}$. We might generate the constraint c for e:

$$c = \langle\!\langle \cdot \vdash e : \tau \rangle\!\rangle = \exists \beta.\, (\alpha \leqslant \{\text{quack} : \beta\}) \,\&\, (\exists \gamma.\, (\beta \leqslant \{\} \to \gamma) \,\&\, (\gamma \leqslant \tau))$$

FL constraint generation is defined in Appendix C. Notice subtyping constraints follow dataflow [21]. Later, we show c can be *rewritten* to $\exists \beta.\, (\{\text{quack} : \{\} \to \beta\} \to \beta) \leqslant \tau$, and hence $\cdot \vdash e : \forall \beta.\, \{\text{quack} : \{\} \to \beta\} \to \beta$.

Lastly, we define *constraint equivalence* $\cong$:

$$c \cong c' \iff \forall \psi.\ (\psi \vdash c \iff \psi \vdash c')$$

The description of Warp can be used to instantiate an existing constraint-based framework like HM(X) of Odersky, Sulzmann and Wehr [43]. Hence, we do not consider let-polymorphism (like Dolan [20] and Parreaux [44] do for algebraic subtyping), which HM(X) could yield 'for free'. We focus on **resolution of subtyping constraints**, and not more advanced constraints (like local type assumptions [71]).

3.1.2 Algebraic subtyping

Combining bounded parametric polymorphism with both principal type inference and decidability of type scheme subsumption historically proved to be a difficult problem, leading to distrust in implicit subtyping as part of language design [45] – so much so that research would avoid subtyping due to its problematic interaction with type inference (see e.g. [6, Section 3.5]). Seminal work in the area is by Pottier [53], who set out a framework for type inference under subtyping, but did not reach a satisfactory solution. The problem was ultimately resolved by Dolan [20] in his thesis.

Original work (Dolan)

There are two (closely linked) core principles guiding Dolan's approach [20, Section 1.3]:

Extensibility Dolan identified that a core problem in previous solutions was closed-world reasoning on the language of types [20, Section 1.3.1]. Thus, he requires *extensibility*: that for considered type systems, extending their type language preserves typing of programs.

Algebra before syntax Dolan found that the common syntactic approach to defining the type language has neglected ensuring subtyping is well-behaved [20, Section 1.3.2]. Thus, he argued for an algebraic construction of the type language. Extensibility motivated many algebraic properties [20, Section 2.1.5].

Using these principles and a careful formal treatment relying on abstract algebra, Dolan invented MLsub – a statically typed language with support for structural subtyping, building on core ML, and boasting ML-style type inference with bounded parametric polymorphism and decidable type scheme subsumption.

Two assumptions underlie MLsub which enable its properties: subtyping forming a **distributive lattice**, and the **polarity restriction** of the type language. While Dolan uses an HM-like presentation (with *bi*unification and *bi*substitution), I illustrate these assumptions in a constraint-based setting.

Distributive lattice While constructing a lattice of types for conveniently well-behaved subtyping relations was standard, Dolan found it crucial for the lattice to be *distributive* [20, Section 3.2].[6] While his main motivation is to ensure subsumption is decidable, this is a useful assumption in general.

Polarity restriction Dolan [20, Section 5.1] uses a *polar types* construction due to Pottier [53], where the type language is split into positive τ^+ and negative τ^- types – corresponding to *outputs* (which we lower-bound) and *inputs* (dually: upper-bound). We restrict joins and meets so that $\tau^+ ::= \cdots \mid \tau^+ \vee \tau^+$ and $\tau^- ::= \cdots \mid \tau^- \wedge \tau^-$, and ensure polarities in type constructors agree with their variance.[7]

The polarity restriction ensures subtyping constraints have form $\tau^+ \leqslant \tau^-$ (cf. data flows from outputs to inputs [20, Section 1.1]). Hence, we can *split* meets and joins via lattice laws [44]:

$$\tau' \vee \tau'' \leqslant \tau \iff \tau' \leqslant \tau \text{ and } \tau'' \leqslant \tau \qquad \tau \leqslant \tau' \wedge \tau'' \iff \tau \leqslant \tau' \text{ and } \tau \leqslant \tau''$$

I decided to investigate follow-up work that loosens this restriction (e.g. allowing functions of type $\alpha \to \alpha \wedge \{\ell : \beta\}$), and Warp does not apply the polarity restriction. Initially, I motivated this choice by the intuition it would be necessary for typing extensible records. However, my technique is ultimately compatible with the polarity restriction (see end of Section 3.4.3).

Example

In the distributive lattice of FL types (see Appendix C), we compute:

$$(\top \to \bot) \wedge (\alpha \to \beta) = \top \to \bot$$
$$\{\text{quack} : \top, \text{walk} : \{\} \mid \square\} \wedge \{\text{quack} : \top \to \top \mid \mathbb{T}\} = \{\text{quack} : \top \to \top, \text{walk} : \{\} \mid \square\}$$
$$\{\text{foo} : \{\}\} \vee (\top \to \top) = \top$$

[6] Looking ahead, Figure 3.4 lists the laws of a distributive lattice (labels L, B).

[7] Covariance preserves polarity, but contravariance flips it: for functions, $\tau^+ ::= \cdots \mid \tau^- \to \tau^+$ and $\tau^- ::= \cdots \mid \tau^+ \to \tau^-$

Later work (Parreaux, Chau)

While Dolan's work is foundational, Warp is closer descended from follow-up work: Simple-sub of Parreaux [44], and MLstruct of Parreaux and Chau [45]. I highlight the following developments:

Constraint graphs Parreaux [44] found that Dolan's *biunification* approach can be difficult to implement and extend. He proposed an alternative approach in Simple-sub that relies on explicitly introducing subtyping constraints and imperatively updating a *constraint graph*. Likewise, I do not use Dolan's biunification (due to the next point), but my method does not rely on mutation.[8]

Boolean algebra In Parreaux's constraint graph setting, Parreaux and Chau [45] propose MLstruct, featuring *complement (negation) types* and removal of the polarity restriction. They use Simple-sub-like constraint solving by requiring complements (with meets & joins) to form a *Boolean algebra*[9]. My approach to constraint solving directly inherits from theirs. However, I attempt to clarify that negations are added to the type language as a *free extension* of the distributive lattice of type constructors, explaining why they are a safe addition (Section 3.2.3).

Non-extensibility For Dolan, extensibility also means that 'useless' types like $(\alpha \to \beta) \vee \{\ell : \gamma\}$ should not be *equal* to $\top$ [20, Section 1.4.1] – even though they can only be eliminated as such. Parreaux and Chau [45] do not follow this consequence of extensibility (but not others, e.g. distributivity), arguing that this provides better user experience and eases constraint solving. I also do so, but do not share their scepticism of extensibility [44], agreeing with Dolan that it is an important principle.

3.2 Signature

We consider the external side of the Warp framework – the *signature*, i.e. the requirements it entails on the source language. All constructs described here are available as **data** upholding certain **laws**, key for constraint solving (Section 3.3). I endeavour to generalise the properties necessary for MLstruct's solver.

3.2.1 Type constructors

Firstly, following standard practice [24, 54], we will abstract away *type constructors* in the type language. We denote them $K[\overline{\tau}]$, where $\overline{\tau}$ stands for the list of types that occur within.

Example

In FL we have the function and record type constructors. Writing '$\cdot$' for *type holes* in constructors:

$$K ::= \top \mid \bot \mid \cdot \to \cdot \mid \left\{\ell : \dot{\phi} \,\middle|\, \dot{\phi}\right\} \qquad \dot{\phi} ::= \mathbb{T} \mid \mathbb{\perp} \mid \cdot \mid \square$$

Writing $K[\overline{\tau}]$, we plug in $\overline{\tau}$ into each '$\cdot$' in K in order. This establishes that type constructors K non-opaquely contain subterms τ. Here are example types $K[\overline{\tau}]$:

$$(\cdot \to \cdot)[\top, \bot] = \top \to \bot \qquad \{\text{foo} : \cdot, \text{bar} : \square \mid \mathbb{T}\}[\top \to \top] = \{\text{foo} : \top \to \top, \text{bar} : \square \mid \mathbb{T}\}$$

[8] While this is arguably a stylistic choice, purity seems to make the implementation easier to reason about.
[9] A bounded, distributive and complemented lattice – see Figure 3.4 as reference.

$$
\begin{array}{rl}
\tau ::= \alpha & \text{(variable)} \\
\mid K[\overline{\tau}] & \text{(constructor)} \\
\mid \tau \vee \tau & \text{(join)} \\
\mid \tau \wedge \tau & \text{(meet)} \\
\mid \neg\tau & \text{(complement)}
\end{array}
$$

Figure 3.3: Syntax of types τ in Warp.

As expected, we require that type constructors $K[\overline{\tau}]$ form a distributive lattice $(\top, \bot, \sqcup, \sqcap)$, where $\top$ and $\bot$ are nullary type constructors and $\sqcup$ and $\sqcap$ are closed binary operators on type constructors.

When solving constraints, we wish to break down constraints into bounds on type variables. Hence, we have to decompose 'compound' subtyping constraints – those on type constructors. We thus require a **decomposition** operator $K[\overline{\tau}] \lhd K[\overline{\tau}] = c$, specific to a given type language. It decomposes a *subtyping constraint* between two type constructors into an equivalent (general) *constraint*:

$$(K'[\overline{\tau'}] \leqslant K''[\overline{\tau''}]) \cong (K[\overline{\tau'}] \lhd K'[\overline{\tau''}])$$

where the resulting constraints contain structurally smaller type constructors.[10] Decomposition is where type errors arise in the system (e.g. $\top \lhd \bot = \mathbf{F}$, as $\top \leqslant \bot$ is always false).

Example

Constraints on function type constructors decompose as such:

$$(\tau \to \pi) \lhd (\tau' \to \pi') = \tau' \leqslant \tau \;\&\; \pi \leqslant \pi'$$

The type constructor lattice given by $\sqcap$ and $\sqcup$ always agrees with $\wedge$ and $\vee$ on types (Fig. 3.4), so:

$$
\begin{aligned}
(\top \to \top) \wedge (\bot \to \bot) &\equiv (\cdot \to \cdot)[\top, \top] \sqcap (\cdot \to \cdot)[\bot, \bot] = (\cdot \to \cdot)[\top \sqcup \bot, \top \sqcap \bot] \\
&= (\cdot \to \cdot)[\top, \bot] \equiv \top \to \bot
\end{aligned}
$$

3.2.2 Type language

We now describe the syntax of types (Figure 3.3). Loosely following Parreaux and Chau [45], we define it so that types form a free Boolean algebra over type constructors and variables[11]. This algebra satisfies Boolean algebra laws and the laws of the type constructor lattice (Figure 3.4) under type equivalence $\tau \equiv \tau$.[12] Subtyping $\leqslant$ must agree with the type algebra and the typing judgement:

$$\tau \leqslant \pi \iff \tau \equiv \tau \wedge \pi \iff \pi \equiv \tau \vee \pi$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \leqslant \tau'}{\Gamma \vdash e : \tau'}$$

[10] We do not formalise this property, but it would be necessary to do so to prove the constraint solving process terminates.
[11] Like Dolan [20], our type variables are *opaque* in the lattice – avoiding a closed-world assumption about their possible values.
[12] Following this, the algebra of types is free extension of the algebra of type constructors with variables and complements.

$$\boxed{\tau \equiv \tau}$$

$$\begin{array}{rr}
\tau \vee (\tau' \vee \tau'') \equiv (\tau \vee \tau') \vee \tau'' & \text{(L: associativity } \vee\text{)} \\
\tau \wedge (\tau' \wedge \tau'') \equiv (\tau \wedge \tau') \wedge \tau'' & \text{(L: associativity } \wedge\text{)} \\
\tau \wedge \tau' \equiv \tau' \wedge \tau & \text{(L: commutativity)} \\
\tau \vee (\tau \wedge \tau') = \tau \quad \tau \wedge (\tau \vee \tau') = \tau & \text{(L: absorption)} \\
\tau \vee \bot \equiv \tau \quad \tau \wedge \top \equiv \tau & \text{(B: bounds)} \\
\tau \wedge (\tau' \vee \tau'') \equiv (\tau \wedge \tau') \vee (\tau \wedge \tau'') & \text{(D: distributivity)} \\
\tau \vee \neg\tau = \top \quad \tau \wedge \neg\tau = \bot & \text{(C complements)} \\
K_1\left[\overline{\tau'}\right] \vee K_2\left[\overline{\tau''}\right] \equiv K_1\left[\overline{\tau'}\right] \sqcup K_2\left[\overline{\tau''}\right] & \text{(type constructor } \sqcup/\vee\text{)} \\
K_1\left[\overline{\tau'}\right] \wedge K_2\left[\overline{\tau''}\right] \equiv K_1\left[\overline{\tau'}\right] \sqcap K_2\left[\overline{\tau''}\right] & \text{(type constructor } \sqcap/\wedge\text{)}
\end{array}$$

$$\frac{\tau \equiv \tau'}{E[\tau] \equiv E[\tau']} \quad \text{(congruence)}$$

$$E\langle\diamond\rangle ::= \diamond \mid K[\overline{\tau}, \diamond, \overline{\tau}] \mid E\langle\diamond\rangle \vee \tau \mid \tau \vee E\langle\diamond\rangle \mid E\langle\diamond\rangle \wedge \tau \mid \tau \wedge E\langle\diamond\rangle \mid \neg E\langle\diamond\rangle$$

Figure 3.4: Laws of the Boolean algebra of types and the lattice of type constructors, forming the equivalence $\tau \equiv \tau$. Equivalence contexts $E\langle\diamond\rangle$ are used to specify the congruence rule. We have laws of a lattice (labelled L) that is bounded (B), distributive (D), and complemented (C) – altogether, a Boolean algebra.

3.2.3 Complement types

Following Parreaux and Chau [45], our complement types have algebraic foundation, as opposed to a set-theoretic one. The two have different interpretations of subtyping, denoted $\neg$ and $\dot{\neg}$:[13]

$$\begin{array}{rcll}
\pi \leqslant \dot{\neg}\tau & \iff & \pi \not\leqslant \tau & \text{(set-theoretic)} \\
\pi \leqslant \neg\tau & \iff & \pi \wedge \tau \leqslant \bot & \text{(algebraic)}
\end{array}$$

Set-theoretic-like complements notoriously make constraint solving harder, smuggling negations into the underlying logic. On the other hand, our algebraic complement types provide a weaker condition, but one of great help to constraint solving – as we explore in Section 3.3.

Example

Take $\pi = \{\text{foo} : \square \mid \mathbb{T}\}$ and $\tau = \{\text{foo} : \top \mid \mathbb{T}\}$. The two interpretations of $\pi \leqslant \neg\tau$ disagree:

Set-theoretic It is not the case that $\pi \leqslant \tau$, thus $\pi \leqslant \dot{\neg}\tau$. ✓

Algebraic Based on Figure 2.4, we have $\pi \wedge \tau = \{\text{foo} : \perp\!\!\!\perp \mid \mathbb{T}\} \not\leqslant \bot$, so $\pi \not\leqslant \neg\tau$. ↯

Note that $\neg$ admits fewer subtypes than $\dot{\neg}$, since for any sets A and B:

$$A \cap B = \varnothing \implies \neg(A \subseteq B) \qquad \text{thus} \qquad \tau \leqslant \neg\pi \implies \tau \leqslant \dot{\neg}\pi$$

[13] The algebraic interpretation follows by: $\pi \leqslant \neg\tau \implies \pi \wedge \tau \leqslant \tau \wedge \neg\tau \implies \pi \wedge \tau \leqslant \bot$, and $\pi \wedge \tau \leqslant \bot \implies (\pi \wedge \tau) \vee \neg\tau \leqslant \neg\tau \implies (\pi \vee \neg\tau) \wedge (\pi \vee \neg\tau) \leqslant \neg\tau \implies (\pi \vee \neg\tau) \wedge \top \leqslant \neg\tau \implies \pi \vee \neg\tau \leqslant \neg\tau \implies \pi \leqslant \neg\tau$.

3.2.4 Summary

Summarising the requirements of Warp's signature:

- Type constructors form a distributive lattice, and subtyping constraints on them can be decomposed.
- The type language forms a Boolean algebra.
- The type system admits an implicit subtyping rule.

To determine these, I was inspired by the requirements of techniques of Dolan and Mycroft [21] and Parreaux and Chau [45]. Later, we see that Fabric (Chapter 4) and Star (Chapter 5) both successfully implement the signature of Warp, showing it is reasonable in practice.

3.3 Constraint solving

We now describe the MLstruct-inspired constraint solving process used in Warp using the already described setting of the constraint language and signature.

I largely follow the tradition of the framework by Pottier [52] and the Boolean algebraic techniques of Parreaux and Chau [45] in MLstruct. The key difference to MLstruct is a simpler and more extensible presentation, using term rewriting of constraints. Furthermore, in Section 3.4 I give the main contribution of the framework: extension of the type language and constraint solving with support for *homomorphisms*.

This section is structured as follows:

3.3.1 Massaging We first show that all subtyping constraints $\leqslant$ can be reduced to a conjunction of *variable-bound* constraints (of syntax $\tau \leqslant \alpha \mid \alpha \leqslant \tau$, mirroring bounded parametric type schemes).

3.3.2 Plumbing Then, we show how we manipulate and combine these variable-bound constraints into a list-of-bounds normal form. We then perform the *closure* computation.

3.3.3 Solutions Lastly, we briefly consider how we can extract a type scheme from a normalised generated constraint, and how these can be simplified.

Altogether, I aim to reproduce[14] the constraint solving of Parreaux and Chau [45] in MLstruct in a constraint-based style that is simpler and easier to extend.

Constraint solving is given in terms of a small-step term rewriting relation $\rightsquigarrow$ (Figure 3.5). The solving process is given by its reflexive-transitive closure, $\rightsquigarrow^*$. To show constraint solving always yields correct results, we rely on a theorem that $\rightsquigarrow^*$ preserves constraint satisfaction:

Theorem 3.1 (Semantic preservation)**.** *If* $c \rightsquigarrow^* c'$*, then* $c \cong c'$*.*

This theorem follows straightforwardly by proving individual cases of $\rightsquigarrow$, relying on known Boolean algebra properties [45].

We now consider various properties of the constraint-solving steps, giving constructive proofs – these directly lead to a practical implementation.

[14] Due to the complexity of their description, a proof of equivalence seems difficult. However, the core ideas remain the same.

$$\boxed{c \rightsquigarrow c}$$

$$\frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \leqslant \tau_2 \rightsquigarrow \tau_1' \leqslant \tau_2'} \quad \text{(equivalence)}$$

$$\mathbf{T} \,\&\, c \rightsquigarrow c \quad \text{(identity } \mathbf{T} \text{ of } \&)$$

$$\mathbf{F} \,\&\, c \rightsquigarrow \mathbf{F} \quad \text{(annihilating } \mathbf{F} \text{ of } \&)$$

$$c \,\&\, (c' \,\&\, c'') \rightsquigarrow (c \,\&\, c') \,\&\, c'' \quad \text{(associative } \&)$$

$$c \,\&\, c' \rightsquigarrow c' \,\&\, c \quad \text{(commutative } \&)$$

$$c \,\&\, c \rightsquigarrow c \quad \text{(idempotent } \&)$$

$$\exists \alpha.\, \mathbf{T} \rightsquigarrow \mathbf{T} \quad \exists \alpha.\, \mathbf{F} \rightsquigarrow \mathbf{F} \quad \text{(always-exists)}$$

$$(\exists \alpha.\, c) \,\&\, c' \rightsquigarrow \exists \alpha.\, (c \,\&\, c') \quad \text{(factor-out)}$$

$$\tau \leqslant \tau' \,\&\, \tau' \leqslant \tau'' \rightsquigarrow (\tau \leqslant \tau' \,\&\, \tau' \leqslant \tau'') \,\&\, \tau \leqslant \tau'' \quad \text{(transitivity)}$$

$$T'\left[\overline{\tau'}\right] \leqslant T''\left[\overline{\tau''}\right] \rightsquigarrow T'\left[\overline{\tau'}\right] \triangleleft T''\left[\overline{\tau''}\right] \quad \text{(decomposition)}$$

$$\neg\tau \leqslant \tau' \rightsquigarrow \neg\tau' \leqslant \tau \quad \neg\tau' \leqslant \tau \rightsquigarrow \neg\tau \leqslant \tau' \quad \text{(negation)}$$

$$\tau \wedge \tau' \leqslant \tau'' \rightsquigarrow \tau \leqslant \neg\tau' \vee \tau'' \qquad \tau \leqslant \tau' \vee \tau'' \rightsquigarrow \tau \wedge \neg\tau' \leqslant \tau'' \quad \text{(swapping)}$$

$$\tau' \vee \tau'' \leqslant \tau \rightsquigarrow \tau' \leqslant \tau \,\&\, \tau'' \leqslant \tau \qquad \tau \leqslant \tau' \wedge \tau'' \rightsquigarrow \tau \leqslant \tau' \,\&\, \tau \leqslant \tau'' \quad \text{(splitting)}$$

$$\tau' \leqslant \tau \,\&\, \tau'' \leqslant \tau \rightsquigarrow \tau' \vee \tau'' \leqslant \tau \qquad \tau \leqslant \tau' \,\&\, \tau \leqslant \tau'' \rightsquigarrow \tau \leqslant \tau' \wedge \tau'' \quad \text{(combining)}$$

$$\frac{c \rightsquigarrow c'}{C[c] \rightsquigarrow C[c']} \quad \text{(congruence)}$$

$$C\langle\diamond\rangle ::= \diamond \mid C\langle\diamond\rangle \,\&\, c \mid c \,\&\, C\langle\diamond\rangle \mid \exists \alpha.\, C\langle\diamond\rangle \quad \text{(solving contexts)}$$

Figure 3.5: Definition of constraint solving steps $\rightsquigarrow$. Congruence for $\rightsquigarrow$ is given by constraint solving contexts $C\langle\diamond\rangle$.

$$\begin{aligned}
\tau_{\text{cnf}} &::= \top \mid \tau_{\text{cnf-clause}} \wedge \tau_{\text{cnf}} \\
\tau_{\text{cnf-clause}} &::= K\left[\overline{\tau_{\text{cnf}}}\right] \vee \neg K\left[\overline{\tau_{\text{cnf}}}\right] \vee \tau_{\text{cnf-clause-lits}} \\
\tau_{\text{cnf-clause-lits}} &::= \bot \mid \tau_{\text{lit}} \vee \tau_{\text{cnf-clause-lits}} \\
\tau_{\text{lit}} &::= \alpha \mid \neg\alpha
\end{aligned}$$

Figure 3.6: Grammar of the conjunctive normal forms (CNF) for types τ – a *meet-of-clauses* τ_{cnf}, where clauses $\tau_{\text{cnf-clause}}$ are *joins-of-atoms*. Clauses have only one type constructor K at each polarity, but many variables. We additionally require that no τ_{lit} occurs twice in $\tau_{\text{cnf-clause-lits}}$.

3.3.1 Massaging

We define the syntax of a *variable bound* V and *variable bounds* $V_\&$ constraints as:

$$V ::= \tau \leqslant \alpha \mid \alpha \leqslant \tau \qquad V_\& ::= \mathbf{T} \mid V \mid V_\& \,\&\, V_\&$$

We shall show by construction that:

Theorem 3.2 (Subtyping constraints yield variable bounds)**.** *Given* $c = (\tau \leqslant \pi)$*, we have that:*

$$c \rightsquigarrow^* \mathbf{F} \;\; \textit{or} \;\; \exists c' = V_\&.\, c \rightsquigarrow^* c'$$

Let us call type constructors and type variables *atoms* (literals). The proof sketch is as follows:

1. Show all subtyping constraints are equivalent to $\top \leqslant \tau_{\text{cnf}}$,[15] for a type τ_{cnf} in *conjunctive normal form* (or a *meet-of-joins-of-atoms*) – as defined in Figure 3.6.
2. Split the constraint into a conjunction of constraints $\top \leqslant \tau_{\text{cnf-clause}}$ (CNF clause; *join-of-atoms*).
3. Use *swapping* to give variable bounds for each occurring variable, yielding $V_\&$.

The technique is essentially the same as Parreaux and Chau [45], but my presentation is generalised (up to the signature) and more direct (constraint-based). Like them, I call this process *constraint massaging*.

Proof. We write $\mathrm{cnf}(\tau)$ for the τ_{cnf} such that $\tau \equiv \tau_{\text{cnf}}$. CNF can be built compositionally, analogously to usual Boolean algebra techniques.[16] To transform any $c = \tau \leqslant \pi$ into $\top \leqslant \tau_{\text{cnf}}$ we step like so:

$$\tau \leqslant \pi \overset{\text{swap}}{\rightsquigarrow} \top \leqslant \neg\tau \wedge \pi \overset{\text{equiv.}}{\rightsquigarrow} \top \leqslant \mathrm{cnf}(\neg\tau \wedge \pi)$$

at which point we reach a $c' = \top \leqslant \tau_{\text{cnf}}$ equivalent to c. By simple induction we can see that we can split this into a conjunction of constraints $\top \leqslant \tau_{\text{cnf-clause}}$, since:

$$\top \leqslant \tau_{\text{cnf-clause}} \wedge \tau_{\text{cnf}} \overset{\text{split}}{\rightsquigarrow} \top \leqslant \tau_{\text{cnf-clause}} \,\&\, \top \leqslant \tau_{\text{cnf}}$$

It remains to show $\top \leqslant \tau_{\text{cnf-clause}} \rightsquigarrow^* V_\&$. Let $\tau_{\text{cnf}} = K'\left[\overline{\tau'_{\text{cnf}}}\right] \vee \neg K''\left[\overline{\tau''_{\text{cnf}}}\right] \vee \tau_{\text{cnf-clause-lits}}$. By cases:

- If $\tau_{\text{cnf-clause-lits}} = \bot$, then:

  $$\top \leqslant K'\left[\overline{\tau'_{\text{cnf}}}\right] \vee \neg K''\left[\overline{\tau''_{\text{cnf}}}\right] \vee \bot \overset{\text{equiv.}}{\rightsquigarrow} \cdots \overset{\text{swap}}{\rightsquigarrow} \cdots \overset{\text{decomp.}}{\rightsquigarrow} K''\left[\overline{\tau''_{\text{cnf}}}\right] \lhd K'\left[\overline{\tau'_{\text{cnf}}}\right]$$

  and we proceed recursively at the result of $\lhd$, assuming it is well-behaved so we eventually terminate. This is the case where we might reach a 'type-error' constraint $\mathbf{F}$ (when $\lhd$ yields $\mathbf{F}$).

- If $\tau_{\text{cnf-clause-lits}} = \tau_{\text{lit}} \vee \tau'_{\text{cnf-clause-lits}}$, the constraint is satisfiable ($\tau_{\text{lit}} \mapsto \top$). We return a conjunction of **separate**[17] constraints for each τ_{lit}. There are two cases:
  — If $\tau_{\text{lit}} = \alpha$, then $\top \leqslant \alpha \vee \tau \overset{\text{swap}}{\rightsquigarrow} \neg\tau \leqslant \alpha$. — If $\tau_{\text{lit}} = \neg\alpha$, then $\top \leqslant \neg\alpha \vee \tau' \overset{\text{swap}}{\rightsquigarrow} \alpha \leqslant \tau'$.

□

[15] Notice that $\top \leqslant \tau \iff \top = \tau$.
[16] There is an analogous *disjunctive normal form* (DNF) in my implementation – while not necessary, it can be more size-efficient.
[17] We can *sufficiently* take one τ_{lit} – but no choice is *natural*, so for a principal type scheme we include all of them.

Example

Let us massage $c = (\alpha \wedge \{\} \to \beta) \leqslant (\{\text{foo} : \gamma\} \to \top)$. We first decompose:

$$c \rightsquigarrow (\alpha \wedge \{\} \to \beta) \lhd (\{\text{foo} : \gamma\} \to \top) = \{\text{foo} : \gamma \leqslant \alpha \wedge \{\}\}_{c'} \,\&\, \beta \leqslant \top$$

We consider the first conjunct, c' – the latter is normalised – and transform into $\top \leqslant \tau_{\text{cnf}}$:

$$c' \rightsquigarrow^* c'_{\text{cnf}} = \top \leqslant \underbrace{(\alpha \vee \neg\{\text{foo} : \gamma\})}_{(1)} \wedge \underbrace{(\{\} \vee \neg\{\text{foo} : \gamma\})}_{(2)}$$

Splitting into clauses (1) and (2), we obtain:

$$\begin{cases} \top \leqslant \alpha \vee \neg\{\text{foo} : \gamma\} \rightsquigarrow \{\text{foo} : \gamma\} \leqslant \alpha \\ \top \leqslant \{\} \vee \neg\{\text{foo} : \gamma\} \rightsquigarrow \{\text{foo} : \gamma\} \leqslant \{\} \rightsquigarrow \{\text{foo} : \gamma\} \lhd \{\} = \mathbf{T} \end{cases}$$

$$\implies c \rightsquigarrow^* (\{\text{foo} : \gamma\} \leqslant \alpha \,\&\, \mathbf{T}) \,\&\, \beta \leqslant \top$$

Note that negations appear in intermediate constraints, but not in the output.

3.3.2 Plumbing

We now consider the normalisation of variable bounds and the transitive closure computation in the style of Pottier [52]. These techniques were first described for algebraic subtyping by Parreaux [44] for Simple-sub (and thus inherited by MLstruct) – I adapt them to the constraint-based presentation.

Normalised bounds W have syntax:

$$W ::= \mathbf{T} \mid (\tau \leqslant \alpha \,\&\, \alpha \leqslant \tau) \,\&\, W$$

constrained such that no α occurs twice, and αs are sorted under some fixed total ordering. We have only shown that subtyping constraints reduce to (multiset-like) variable bounds V^*, but we can strengthen this:

Lemma 3.3 (Normalisation of bounds)**.** *For any $c = V_{\&}$, there exists $c' = W$ such that $c \rightsquigarrow^* c'$.*

Proof. Straightforward by $\&$ forming a commutative monoid and by the *combining* step. □

To move towards general constraints, we also have a lemma that existentials can be lifted to the top-level:

Lemma 3.4 (Top-level existentials)**.** *For any c, there exists c' such that c' has no existentials and $c \rightsquigarrow^* \exists\overline{\alpha}.\, c'$.*

Proof. (This is simplified by the constraint language only featuring conjunctions and existentials besides the subtyping predicate). Straightforward using the fact $\exists$ can always be *factored out* from conjunctions. □

Finally, I give the normal form of constraints $\hat{c}$ – normalised bounds with top-level existentials:

$$\hat{c} ::= \mathbf{F} \mid \exists\overline{\alpha}.\, W$$

We can now state that **constraints normalise**:

Theorem 3.5 (Normalisation of constraints)**.** *For any c, there exists $\hat{c}$ such that $c \rightsquigarrow^* \hat{c}$.*

Proof. By factoring out existentials to the top-level, and normalising the variable bounds. □

Lastly, whenever we are in a list-of-bounds form, we can invoke transitivity. This yields an additional constraint $\tau_\alpha^+ \leqslant \tau_\alpha^-$ for each $\tau_\alpha^+ \leqslant \alpha \,\&\, \alpha \leqslant \tau_\alpha^-$ in the list. Invoking transitivity and normalising can only yield stronger bounds, i.e. the process is monotone under the order $\preceq$ defined as:[18]

$$\hat{c}' \preceq \hat{c} \iff \hat{c}' = \mathbf{F} \text{ or } (\forall \alpha.\, \pi_\alpha^+ \leqslant \tau_\alpha^+ \text{ and } \tau_\alpha^- \leqslant \pi_\alpha^-)$$

where τ_α and π_α are the bounds on a variable α in $\hat{c}'$ and $\hat{c}$, respectively. While it is expected this process terminates [44, 45, 52] – determining satisfiability – I only conjecture it and give empirical evidence of termination:

Conjecture 3.6. *For any normalised constraint $\hat{c}$, applying transitivity at each α and leads to some $\hat{c}'$ such that $\hat{c}' \preceq \hat{c}$. This process eventually reaches a fixed-point – the* ***solved constraint*** $\mathcal{S}(c)$.

Example

Denoting applications of transitivity by $\Rightarrow$, first consider:

$$\begin{aligned} &\{\text{quack} : \beta \,|\, \square\} \leqslant \alpha \,\&\, \alpha \leqslant \{\text{quack} : \top \to \top \,|\, \mathbb{T}\} \\ \Rightarrow\; &\{\text{quack} : \beta \,|\, \square\} \lhd \{\text{quack} : \top \to \top \,|\, \mathbb{T}\} = \gamma \leqslant \top \to \top \end{aligned}$$

where we obtained a bound on the type β of the field quack. Now consider:

$$(\top \to \top \leqslant \gamma) \,\&\, (\gamma \leqslant \{\}) \implies ((\top \to \top) \lhd \{\}) = \mathbf{F}$$

where we reached a contradiction on satisfying bounds on γ.

3.3.3 Solutions

We now briefly consider the question of extracting the type scheme from a solved constraint.

Given a generated constraint $\langle\!\langle \cdot \vdash e : \tau \rangle\!\rangle$, if it is solved successfully we have some $\mathcal{S}(\langle\!\langle \cdot \vdash e : \tau \rangle\!\rangle) = \exists \overline{\alpha}.\, W$, and return the type scheme $\sigma = \forall W.\, \tau$.

It is important to simplify type schemes – however, I do not propose novel ways to do this, and entirely refer to Parreaux [44] and Parreaux and Chau [45]. In practice, I have found that to ensure efficient termination I had to simplify the CNF forms with standard techniques. Inlining bounds (like Dolan [20]) and removing redundant type variables mainly serves to improve readability.

Example

Given $e = \lambda x.\, \lambda y.\, (x\,\{\text{quack} = y\}).\text{sound}$, my implementation produces $\cdot \vdash e : \sigma$ and a simplified $\sigma' \equiv^\forall \sigma$ (i.e. $\sigma' \leqslant^\forall \sigma \leqslant^\forall \sigma'$) as such:

$$\begin{aligned} \mathcal{S}(\langle\!\langle \cdot \vdash e : \alpha \rangle\!\rangle) \cong \exists \beta, \gamma, \delta, \varepsilon.\, \alpha \leqslant \beta \to \gamma \to \delta \,\&\, \beta \leqslant \{\text{quack} : \gamma \,|\, \square\} \to \delta \\ \&\, \varepsilon \leqslant \{\text{sound} : \delta \,|\, \mathbb{T}\} \end{aligned}$$

$$\sigma = \forall \alpha \leqslant \beta \to \gamma \to \delta,\; \beta \leqslant \{\text{quack} : \gamma \,|\, \square\} \to \delta, \varepsilon \leqslant \{\text{sound} : \varepsilon \,|\, \mathbb{T}\}.\, \alpha$$

$$\sigma' = \forall \gamma, \delta.\, (\{\text{quack} : \gamma \,|\, \square\} \to \{\text{sound} : \delta \,|\, \mathbb{T}\}) \to \gamma \to \delta$$

[18] Introducing another subsumption-like relation might be surprising, but $\preceq$ is more useful here as it is syntactic (checks bounds) rather than semantic (as $\leqslant^\forall$, which checks instantiations). It is also a stronger condition: $\preceq$ implies $\leqslant^\forall$.

3.4 Breaking records: Homomorphism extension

Early on, Cardelli and Mitchell [12] identified a crucial issue with typing extensible records using subtyping. We exemplify it on an FL program using record extension:

$$e = \lambda x.\, \lambda y.\, \{\ell = y \mid x\}$$

which would naively be given the (non-principal) type scheme

$$\cdot \vdash e : \forall \alpha.\, \{\ell : \square \mid \mathbb{T}\} \to \alpha \to \{\ell : \alpha \mid \mathbb{T}\}$$

causing *loss of information* about the non-ℓ fields of x. We cannot obviously address this with just subtyping and parametric polymorphism [10], and the standard solution is extending the system with row polymorphism [42, 57]. Such an extension of algebraic subtyping was sketched by Marques et al. [40].

We might prefer to avoid adding rows to the system – avoiding the inherent complexity – and to stick to just bounded parametric polymorphism. One approach to do so is updating a record type underlying a type variable via a 'type-function' $\mathrm{update}_\ell(\tau, \phi) = \tau$, like:

$$\cdot \vdash e : \forall \alpha, \rho \leqslant \{\ell : \square \mid \mathbb{T}\}.\, \rho \to \alpha \to \mathrm{update}_\ell(\rho, \alpha) \qquad \text{(only a sketch!)}$$

It would be convenient to use *metafunctions on types* to specify type schemes.

I propose to follow the algebraic approach and exploit homomorphisms in the type algebra – well-behaved (meta)functions – for this purpose. Thanks to homomorphism laws (and some extra structure), we are able to successfully solve constraints containing applications of homomorphisms to types.

Example

For typing extensible records, we use a homomorphism forget_ℓ that sets the type of a field ℓ to $\mathbb{T}$. Intersecting its result with a singleton $\{\ell : \phi\}$ lets us set any field type (as $\forall \phi.\, \mathbb{T} \wedge \phi = \phi$), e.g.:

$$\begin{aligned} \tau &= \{\mathrm{foo} : \square \mid \square\} \\ \mathrm{forget}_{\mathrm{foo}}(\tau) &= \{\mathrm{foo} : \mathbb{T} \mid \square\} \\ \mathrm{forget}_{\mathrm{foo}}(\tau) \wedge \{\mathrm{foo} : \alpha \mid \mathbb{T}\} &= \{\mathrm{foo} : \alpha \mid \square\} \end{aligned}$$

replacing an absent field in τ with a present α. Note $\mathrm{update}_\ell(\tau, \phi) = \mathrm{forget}_\ell(\tau) \wedge \{\ell : \phi \mid \mathbb{T}\}$. forget_ℓ is essentially the retraction operator on record types of Cardelli and Mitchell [12].

Lorenzen, White, Dolan, Eisenberg and Lindley [38] use a similar approach, introducing a *dagger* function † in a simple polar *mode language*. In contrast, my approach works with variables and without polarities.

3.4.1 Signature

Firstly, we add applications of **homomorphisms** θ – specific to a given language – to the syntax of types:

$$\tau ::= \cdots \mid \theta(\tau)$$

We extend type equivalence to respect the homomorphism laws (Figure 3.7). We also must have a homomorphism id such that $\mathrm{id}(\tau) \equiv \tau$ for all τ, and require homomorphisms *compose* with a $\circ$ such that:

$$(\theta' \circ \theta'')(\tau) \equiv \theta'(\theta''(\tau))$$

$$\boxed{\tau \equiv \tau}$$

$$\theta(\tau \wedge \pi) \equiv \theta(\tau) \wedge \theta(\pi)$$
$$\theta(\tau \vee \pi) \equiv \theta(\tau) \vee \theta(\pi)$$
$$\theta(\top) \equiv \top$$
$$\theta(\bot) \equiv \bot$$

$$E\langle\diamond\rangle ::= \cdots \mid \theta(E\langle\diamond\rangle)$$

Figure 3.7: Homomorphism laws in the Boolean algebra of types, which extend the definition of equivalence $\tau \equiv \tau$. Note that $\theta(\neg\tau) \equiv \neg\theta(\tau)$ follows from these laws (by a routine check of complement axioms).

Adjoints

Motivation While most of the constraint solving process can be easily adapted to homomorphisms, it is problematic to transform constraints $\theta(\alpha) \leqslant \tau$ (or $\tau \leqslant \theta(\alpha)$) into bounds on α. To solve this, I was inspired by a folklore constraint solving trick: for any $f : X \to Y$ we find a $g : Y \to X$ such that $f(x) \leqslant y \iff x \leqslant g(y)$ – a **Galois connection**, i.e. an adjoint pair $f \dashv g$ in the preorder $\leqslant$. Indeed, at $\theta \mapsto f, x \mapsto \alpha, y \mapsto \tau, g \mapsto \overrightarrow{\theta}$ we get a variable bound for α via $\theta(\alpha) \leqslant \tau \iff \alpha \leqslant \overrightarrow{\theta}(\tau)$.

Some useful morphisms (like forget and free) do not fit in this Galois connection framework. I thus extended it with *remainder* constraints.

Development To facilitate solving of general constraints,[19] we require homomorphisms θ have an *adjoint-like* structure given by *left-* and *right-adjoint* homomorphisms $\overleftarrow{\theta}$ and $\overrightarrow{\theta}$ for all θ. We also allow *remainder* constraint functions $\underline{\overleftarrow{\theta}}(\tau) = c$ and $\underline{\overrightarrow{\theta}}(\tau) = c$, so that we have equivalences:

$$(\theta(\tau) \leqslant \pi) \cong \left(\tau \leqslant \overrightarrow{\theta}(\pi) \;\&\; \underline{\overrightarrow{\theta}}(\pi)\right)$$
$$(\tau \leqslant \theta(\pi)) \cong \left(\overleftarrow{\theta}(\tau) \leqslant \pi \;\&\; \underline{\overleftarrow{\theta}}(\tau)\right)$$

The use of $\underline{\overleftarrow{\theta}}$ and $\underline{\overrightarrow{\theta}}$ makes it clear why we call these remainders: when they introduce no further constraint ($\underline{\overleftarrow{\theta}} = \underline{\overrightarrow{\theta}} = \mathbf{T}$), $\overleftarrow{\theta}$ and $\overrightarrow{\theta}$ are precisely left and right Galois connections to θ.

Example

In order to give forget_ℓ adjoint-like structure, it needs a 'dual'. We call this dual free_ℓ, and define it to set field ℓ to $\perp\!\!\!\perp$. We have the following adjoints and remainders (per Appendix C):

θ	$\overleftarrow{\theta}$	$\underline{\overleftarrow{\theta}}(\tau)$	$\overrightarrow{\theta}$	$\underline{\overrightarrow{\theta}}(\tau)$
forget_ℓ	free_ℓ	$\mathbf{T}$	id	$\{\ell : \top \mid \perp\!\!\!\perp\} \leqslant \tau$
free_ℓ	id	$\tau \leqslant \{\ell : \perp\!\!\!\perp \mid \top\}$	forget_ℓ	$\mathbf{T}$
id	id	$\mathbf{T}$	id	$\mathbf{T}$

free_ℓ and forget_ℓ form an 'adjoint pair' (inspiring the naming, cf. free-forgetful adjunction).

[19] In special cases (e.g. under a polarity restriction of homomorphisms), we might only need a left- or right- adjoint.

3.4.2 Constraint solving

We now extend the constraint solving process of Section 3.3 to support homomorphisms. This is relatively straightforward: we only need to amend the construction of variable bounds from subtyping constraints. Thus, we firstly amend the syntax of literals τ_{lit} (in τ_{cnf}):

$$\tau_{\text{lit}} ::= \theta(\alpha) \mid \neg\theta(\alpha)$$

which generalises τ_{lit} (at $\theta = \text{id}$). Crucially, we can still construct the CNF thanks to the fact that θ are homomorphisms: we just *push down* all applications (i.e. $\theta(\tau_{\text{cnf}})$ applies θ to all τ_{lit} within τ_{cnf}).

For constructing variable bounds, only the final cases are affected – we need to show $\top \leqslant \tau_{\text{lit}} \vee \tau$ gives a variable bound. We sketch how to do this by exploiting the adjoint-like structure (via appropriate $\rightsquigarrow$ rules):

- If $\tau_{\text{lit}} = \theta(\alpha)$, then $\top \leqslant \theta(\alpha) \vee \tau \overset{\text{swap}}{\rightsquigarrow} \neg\tau \leqslant \theta(\alpha) \overset{\text{left-adj}}{\rightsquigarrow} \overleftarrow{\theta}(\tau) \leqslant \alpha \;\&\; \underline{\overleftarrow{\theta}}(\tau)$.
- If $\tau_{\text{lit}} = \theta(\neg\alpha)$, then $\top \leqslant \neg\theta(\alpha) \vee \tau \overset{\text{swap}}{\rightsquigarrow} \theta(\alpha) \leqslant \tau \overset{\text{right-adj}}{\rightsquigarrow} \alpha \leqslant \overrightarrow{\theta}(\tau) \;\&\; \underline{\overrightarrow{\theta}}(\tau)$.

where we proceed recursively on any remainder constraints. Note that constraints $\top \leqslant \tau_{\text{lit}} \vee \tau$ remain always satisfiable, since we can always set τ_{lit} to $\top$ (since $\theta(\bot) \equiv \bot, \theta(\top) \equiv \top$).

3.4.3 Typing extensible records

Having described the use of type homomorphisms in constraint solving, we present the promised concrete application – typing extensible records, presented for Featherweight Lua. To this end, we define the homomorphisms forget_ℓ and free_ℓ (introduced in examples):

$$\frac{\tau = \left\{\ell' : \phi_{\ell'}, \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}}{\text{forget}_\ell(\tau) = \left\{\ell' : \mathbb{T}, \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}} \qquad \frac{\tau = \left\{\ell' : \phi_{\ell'}, \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}}{\text{free}_\ell(\tau) = \left\{\ell' : \mathbb{\perp}, \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}}$$

We can now give a rule HExt, equivalent to the *rule scheme* Ext – which we cannot generate constraints for, while avoiding the update problem [12]. The two are given below:

Ext
$$\frac{\Gamma \vdash e : \left\{\ell' : \square, \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\} \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \{\ell' = e' \mid e\} : \left\{\ell' : \tau', \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}}$$

HExt
$$\frac{\Gamma \vdash e : \tau \wedge \{\ell' : \square \mid \mathbb{T}\} \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \{\ell' = e' \mid e\} : \text{forget}_{\ell'}(\tau) \wedge \{\ell' : \tau' \mid \mathbb{T}\}}$$

We can straightforwardly generate constraints for HExt. Thus, the development solves the record update problem under algebraic subtyping without row polymorphism. For details, see Appendix C.

Polarity restriction Lastly, I note that:

$$\text{update}_\ell(\tau, \phi) \equiv \text{free}_\ell(\tau) \vee \{\ell : \phi \mid \mathbb{\perp}\} \equiv \text{forget}_\ell(\tau) \wedge \{\ell : \phi \mid \mathbb{T}\}$$

Hence, the approach is compatible with the polarity restriction: we can use the meet or join version of update depending on the target polarity. Likewise, adjoints agree with polarities.

In fact, Warp has a polarity-restricted subset, in which all constraints are of form $\tau^+ \leqslant \tau^-$, and thus intermediate τ_{cnf} have single-literal clauses. There, we require homomorphisms preserve polarity.

Example

Adapting the update problem example from Cardelli and Mitchell [12] to $e = \lambda r.\ \{p = r.b.\text{not} \mid r\}$,[a] Warp infers $\cdot \vdash e : \sigma$:

$$\sigma \ = \ \alpha \wedge \{b : \{\text{not} : \beta \mid \mathbb{T}\}, p : \square \mid \mathbb{T}\} \ \rightarrow \ \text{free}_p(\alpha) \vee \{p : \beta \mid \mathbb{\perp}\}$$

[a] Note they use a record *update* operator $\{e' \leftarrow \ell = e\}$. It is expressible in Fabric as $\{\ell = e \mid e' \setminus \ell\}$, but not in FL. We replicate the example by updating b in r instead of extending with p, removing $p : \square$ and replacing p by b in σ.

3.5 Correctness

I state correctness theorems for Warp, particularly for *solved constraints* $\mathcal{S}(c)$. We begin with soundness:

Theorem 3.7 (Soundness)**.** *For all ψ and c, $\psi \vdash \mathcal{S}(c)$ if and only if $\psi \vdash c$.*

Proof. Follows directly by semantic preservation of c, since $c \rightsquigarrow^* \mathcal{S}(c)$ by construction. □

However, I only conjecture the following theorems – and only provide evidence based on my implementation (Chapter 4). I note that Warp generalises prior work where these properties hold [44, 45].

Conjecture 3.8 (Termination)**.** *Constraint solving $\mathcal{S}(c)$ described in Section 3.3 always terminates.*

Conjecture 3.9 (Completeness)**.** *For all c, $\mathcal{S}(c) \neq \mathbf{F}$ if and only if $\exists\psi.\ \psi \vdash c$.*

Conjecture 3.10 (Principality)**.** *Returned type schemes are minimal under both $\leqslant^\forall$ and $\preceq$.*

Note that:

- Termination requires appropriate choices of $\lhd$, $\underline{\overleftarrow{\theta}}/\underline{\overrightarrow{\theta}}$, and proving the closure fixed-point is found.
- Completeness relies on the transitive closure finding all contradictions – which is a standard result in constraint solving, particularly Pottier [52] (who attributes *closure* to Eifrig, Smith and Trifonov [22]).
- Principality for $\preceq$ seems straightforward by construction, leading to $\leqslant^\forall$, as $\preceq$ is a stronger condition.

We do not consider decidability of subsumption of type schemes (to the well-founded dismay of Dolan [20]). I instead rely on the claim of Parreaux and Chau [45] that it is resolvable by solving an appropriate constraint, and thus subsumption is decidable given the constraint solving process terminates.

3.6 Conclusions

I have described Warp – a language-agnostic, constraint-based type inference framework based on algebraic subtyping, as introduced by Dolan and Mycroft [21]. It soundly infers bounded parametric type schemes in the presence of structural subtyping.

I use elegant Boolean algebraic techniques, as set out by Parreaux and Chau [45]. I extend their work with *type homomorphisms*, uncovering a new direction in the design of type systems using algebraic subtyping. They are used to infer types for not only FL (Chapter 2; thus Fabric in Chapter 4), but later also Star (Chapter 5; with a *ι shape-index isomorphism*).

Thanks to its general specification in terms of a necessary signature of the source language, Warp improves the understanding of what languages can have their types inferred using algebraic subtyping.

My description lends itself directly to an implementation, which is part of my deliverable – the Weaver compiler for the Fabric language – and described in the following Chapter 4.

4 Design and Implementation of Fabric

In this short chapter, I present **Fabric** – a functional programming language with structural subtyping. Fabric's role in my thesis is similar to MLsub [20] in its goal: it demonstrates type inference. My design (Section 4.1) is driven by:

- Applying Warp (Chapter 3) – making Fabric an example of its expressive power.
- Desire to statically type features present in dynamically languages (Python, Lua).

Furthermore, in Section 4.2 I explore the implementation of my Fabric compiler – Weaver – targeting WebAssembly, which includes a complete implementation of Warp. I thus report my practical experience working with WebAssembly and Warp.

4.1 Design

The basis of Fabric is Chapter 2's Featherweight Lua, extended with more record operations, variants, tuples, and *nominal type abbreviations*. Fabric is essentially a superset of MLsub [21] – I consider Dolan's proposed extensions [20, Chapter 9] – and is influenced by MLstruct [45]. A formal development for Fabric is given in Appendix D.

Selected Fabric programs written in code accepted by Weaver are given in Figures 4.1 and 4.2.

Data types

I focus on describing records and variants, though Fabric also has tuples and integers.

Records Beyond FL, I add field restriction $e \setminus \ell$ [12] and checked-projection $e_?\ell$ (accompanied by an 'optional' field type, $?\boxed{\tau}$), inspired by operations for objects in dynamic languages.

In contrast to Parreaux and Chau [45], I do not use Forsythe-style singleton record types [59], as they lead to a confusing type lattice.[1] Instead, I rely on the same *partial-function* strategy for representing record (and variant) types as in FL (Section 2.2.1).

Variants Following suggestions of Dolan [20, Section 9.2], I add variants $T\ e$ to Fabric (and a match), though not in the form of his *tagged records*. To address his motivating example, I instead propose *untagging* – pattern matching on *any* tag – easily type checked with my syntax of variant types.

Patterns

Structural typing invites a rich pattern-matching facility [17]. I provide a sketch of a design for Fabric in Appendix D, using conservative typing rules to ensure exhaustive matches.

[1] e.g.: in MLstruct $\{x : \alpha\} \vee \{y : \beta\} = \top \implies \neg\{x : \alpha\} \leqslant \{y : \beta\}$, as *there is no empty record type* [45, Section 4.4.5].

```
(* strict Y fixed-point combinator *)
let fix = f => let z = x => f (v => x x v) in z z in
let eval = fix (eval => e =>
  match e with
  | Add r => (eval r.fst) + (eval r.snd)
  | Mul r => (eval r.fst) * (eval r.snd)
  | _ r => r.default (* match on any tag *)
) in eval (
  Add {
    fst: Mul {
      fst: Lit { default: 2 },
      snd: Lit { default: 3 }
    },
    (* different tag and redundant field *)
    snd: Var { default: 1, foo: {} }
  }
)
```

Figure 4.1: Fabric program implementing a recursive function for evaluating arithmetic expressions, where some nodes return a 'default' value. Code generated by Weaver correctly computes 7. Warp infers a polymorphic type scheme for `eval`, e.g. rejecting `eval (Lit {})`. Note $\lambda x.\, e$ is written `x => e`.

```
(* infers that the result is of even length *)
let stutter = xs =>
  match xs with
  | Nil _ => Nil ()
  | Cons c => Cons {
      head: c.head,
      tail: Cons {
        head: c.head, tail: stutter c.tail
      }
    }
(* infers that xs must be of even length *)
in let pairwise = xs =>
  match xs with
  | Nil _ => Nil ()
  | Cons a => (
      match a.tail with
      | Cons b => Cons {
          head: (a.head, b.head), tail: pairwise b.tail
        })
in (xs => pairwise (stutter xs), (* fine! *)
    xs => pairwise (Cons { head: 0, tail: stutter xs })) (* type error! *)
```

Figure 4.2: Fabric program implementing functions `stutter` (duplicating elements in a list, doubling its length) and `pairwise` (pairing adjacent elements of an **even-length** list). Warp detects an error: `pairwise` is passed an odd-length list. Inferring such properties with algebraic subtyping is explored by Binder et al. [7].

Nominal types

Combining both nominal and structural types is desirable [39, 45]. In Fabric, I propose a method inspired by OCaml's private type abbreviations. *Nominal abbreviations* t in Fabric are declared via $\text{type}\, t = \tau \,\text{in}\, e$. They are introduced and eliminated via **abbreviation-casts** $e : \tau \blacktriangleright \tau'$, well-typed exactly when τ is equal to τ' *up to abbreviations*, and e is of type τ. The intermediate type τ is key for type checking.

The same problem is addressed by Parreaux and Chau [45]. Their approach – *nominal tags* (reminiscent of Dolan's *tagged records*) – *refines* existing structural types, and requires dedicated runtime support.

This strategy of adding nominal types to the system could lead to integrating algebraic subtyping with higher-kinded types and higher-rank polymorphism (as set out by Dolan [20, Section 11.1]).

4.2 Implementation

I used OCaml for implementing Weaver. My main dependencies are: Core, `ppx_jane` (pre-processor macros, S-expressions), `alcotest` (test framework), and `angstrom` (parser combinators).

The implementation is complete with minor omissions.[2] For brevity, I describe interesting aspects of type inference (Section 4.2.1) and code generation (Section 4.2.2).

4.2.1 Type inference

I devised an implementation of Warp usable with any type system implementing its signature – not just for Fabric, but for any language for which we can generate constraints.

Exploiting the module language

The framework was particularly satisfying to implement using OCaml's module language. Firstly, the framework's signature is given by an module *signature* (given in Figure 4.3). Type systems are specified by an *implementation* module satisfying the signature, from which a *functor* derives the constraint solver.

Type language representations

Warp's OCaml signature is polymorphic with respect to different type languages. Usually, we use the full type language (following Warp's τ), represented using the OCaml type `Alg.t`. On the other hand, in constraint solving we rely on the use of normal forms – the clause-based CNF/DNF (`CNF.t`/`DNF.t`). As CNF–DNF conversion can blow up the number of clauses, we manipulate constraints as $\tau_{\text{dnf}} \leqslant \tau_{\text{cnf}}$. To avoid code duplication, I implement DNF in terms of CNF, exploiting the duality between them.

Simplification of type schemes

In Section 3.3.3 I hinted at possible approaches to simplifying type schemes. I implemented most heuristics proposed by Parreaux [44] and Parreaux and Chau [45].[3] However, this subset has proven unsatisfactory, often returning redundant type variables that could be simplified by-hand. This limitation is mainly due to the time-consuming implementation. Besides Parreaux's other heuristics, it would be interesting to try automata-simplification of Dolan [20, Chapter 7].

[2] Mainly: compound pattern matching, nominal types, and code generation for recursive let-bindings (but not the Y combinator) and record extension/restriction (except type inference).

[3] Notably, I omitted *unification of indistinguishable variables* and *hash-consing*.

```
module type TypeSystem = sig
  type 'a typ  (* type constructors with holes 'a *)
  type arrow   (* homomorphisms *)

  (* mapping over type constructor subterms *)
  val map : f:('a -> 'b) -> 'a typ -> 'b typ
  (* type constructor lattice *)
  val top : 'a typ
  val bot : 'a typ
  val join : 'a lattice -> 'a typ -> 'a typ -> 'a typ
  val meet : 'a lattice -> 'a typ -> 'a typ -> 'a typ
  (* type constructor decomposition *)
  val decompose : ('a typ * 'a typ) -> ('a * 'a) list Or_error.t

  module Arrow : sig
    type t = arrow

    (* identity homomorphism *)
    val id : t
    val is_id : t -> bool
    (* composition *)
    val compose : t -> t -> t
    (* application to type constructors *)
    val apply : (t -> 'a -> 'a) -> t -> 'a typ -> 'a typ

    (* adjoint structure *)
    val swap_left : ('a typ -> 'a) -> t -> t * 'a
    val swap_right : ('a typ -> 'a) -> t -> t * 'a
  end
end
```

Figure 4.3: The key parts of the OCaml module signature of a type system suitable for type inference.

```
Let (p, e, e') ->
  let* xs, t = pat p in
  let* e = go env e in
  let* () = e <: t in
  go (push xs env) e'
```

Figure 4.4: Example of constraint generation for Fabric let-bindings (let $p = e$ in e'). Here, `go env e` stands for a type τ such that `env` $\vdash$ `e` : τ (under some carried constraint) – a slight deviation from Warp's specification.

Constraint generation

I devised a simple DSL for constraint generation (example in Figure 4.4), using OCaml's let-binding operators for composing constraints and introducing fresh type variables, giving the code a declarative feel.

4.2.2 Code generation

In order to execute Fabric code, I sought to generate code that could be lowered directly to machine code. I chose to go with WebAssembly [29] and to use the Binaryen [74] toolchain, because it is modern, stable, relatively high-level, and well-documented.[4] Furthermore, I wanted to explore its new extension with garbage collection [75], enabling automatic memory management for Fabric.

Binaryen is only accessible from OCaml using the C API.[5] I used the brilliant work of Yallop, Sheets and Madhavapeddy [78] on `ocaml-ctypes` to produce my own, type-safe bindings.

Invoking bindings directly led to clumsy, imperative code. Inspired by Kiselyov [37], I created a DSL on top of Binaryen dubbed Binaryer. Its most prevalent abstraction is `Cell` (Figure 4.5), which encapsulates the different types of storage available in WebAssembly, making it easier to write generic helpers for code generation. I give a basic example of using Binaryer in Figure 4.6.

I used Binaryer to implement code generation for almost all of Fabric. I included a suite of unit tests using a basic printing function, `%print_i32`, provided by Binaryen.

Runtime representation under subtyping

It is well-known that structural typing makes the problem of efficiently representing values at runtime more difficult, as a value might be used as any supertype [50].

This problem also arises for Fabric's record and variant types:

Variants I implemented the same approach as Garrigue [25] – variant values are represented as a pair of a 32-bit hash of its tag and a reference to its payload.

Records Due to time constraints, I did not experiment with record representations, and stuck to a naïve one (with linear-time projection). One interesting direction would be the work of Rémy [55], where a record is a hash table with a pre-computed hashing function (so projection is constant-time).

[4] I also considered other targets – C, Lua, .NET or JVM bytecode, or LLVM. Ultimately, I thought using WebAssembly would be easiest and most interesting.

[5] There are ready solutions – like `binaryen.ml` – but outdated (e.g. no WasmGC) or broken with newer Binaryen versions.

```
type loc = Cell0.loc =
| Local of { idx : T.Index.t }
| Global of { name : string; mut : bool; handle : T.Global.t }
| Table of { name : string; idx : T.Expression.t }
| Address of {
    addr : T.Expression.t;
    size : uint32;
    offset : uint32;
    align : uint32;
    mem : string;
  }
| Struct of {
    target : T.Expression.t;
    struct_type : T.HeapType.t;
    field_idx : T.Index.t;
  }
| Array of {
    target : T.Expression.t;
    array_type : T.HeapType.t;
    idx : T.Expression.t;
  }

type t = { typ : T.Type.t; loc : Cell0.loc }


(* val ( ! ) : t -> T.Expression.t *)
(* val ( := ) : t -> T.Expression.t -> T.Expression.t *)
```

Figure 4.5: Definition of `Cell.t` – abstracting over locals, globals, tables, memory addresses, structure fields, and array elements. I also give the signature of its two primitives: read (`!`) and write (`:=`).

```
(* Set up Binaryen context *)
let (module Ctx) = context () in
let open Ctx in
feature C.Features.reference_types;
feature C.Features.gc;
Memory.set ~initial:10 ~maximum:10 ();
(* Declare this function may print *)
Function.import "print_i32" "spectest" "print_i32" Type.int32 Type.none;
(* Define the main function *)
let main =
  Function.make ~params:Type.none ~result:Type.none (fun _ ->
    let open Cell in
    (* Type foobar_t of a structure with fields foo and bar *)
    let foobar_t =
      Struct.t Type.[ ("foo", field ~mut:true int32); ("bar", field int32) ]
    in
    (* Define a local reference and access it as a foobar_t *)
    let q = local Type.anyref in
    let q_foo = Struct.cell foobar_t !q "foo" in
    let q_bar = Struct.cell foobar_t !q "bar" in
    Control.block
      [
        q :=
          Struct.make foobar_t
            [ ("foo", Const.i32' 42); ("bar", Const.i32' (1337 - 42)) ];
        q_foo := Operator.I32.(!q_foo + !q_bar);
        Function.call "print_i32" [ !q_foo ] Type.none;
      ])
in
(* Mark the main function *)
Function.export "main" main;
Function.start main;
assert (validate ());
interpret ();
```

Figure 4.6: Basic program defined using Binaryer. Under the hood, it calls Binaryen's API, producing WebAssembly code (given in Appendix E) which prints 1337 at runtime.

4.3 Conclusions

I summarise my experience with the three main areas covered in this chapter: the use and implementation of Warp, targeting WebAssembly, and the design of languages for algebraic subtyping.

Experience with Warp By separating concerns of the type and constraint languages in my implementation of Weaver, I was able to experiment with type inference for not only Fabric, but also Star (Chapter 5) – confirming the benefits outlined by Pottier [52]. I have also seen the importance of complete type scheme simplification.

Experience with WebAssembly While the WebAssembly tooling for high-level languages is limited, good bespoke solutions are possible. However, Binaryen had poor error diagnostics for invalid programs, making debugging difficult. WebAssembly's formal specification and stability are great boons towards its practical use. I observed that the current design of automatically managed reference types is limiting in terms of achievable performance, e.g. making some efficient memory representations impossible [23].

Language design with algebraic subtyping Constructing a lattice of types was helpful in preventing misbehaved type system features, and did not prove to be a constraint. Rapidly experimenting with new features using Warp was liberating during design.

5 Structuring Arrays with Algebraic Shapes

Array programming is a programming paradigm focusing on arrays as a fundamental data structure, stemming from Iverson's *array programming model* [36]. It plays a key role in now widespread domains like machine learning and data science. In this paradigm, we operate on often multi-dimensional arrays of data, taking particular advantage of data parallelism hiding within many data processing tasks.

Array programming languages comes in multiple styles: point-free (e.g. MatLab, NumPy [30]), pointful (Dex [47], Ein [3]), combinator-based (Futhark [32]). Despite this variety, there is no satisfactory typing discipline for array programs that balances safety and usability. Array type systems come in two extremes:

Nearly-untyped Often, arrays feature little to no types, which might only describe the number of dimensions or element type. Such types are only sometimes checked statically. This is the case for common array languages (like MatLab, NumPy [30] or PyTorch [46]), but also for arrays in general-purpose languages (like C and Fortran).

Dependently typed Research into array programming languages has produced many approaches reaching for dependent types – where much of the power is used to statically model type-level arithmetic on sizes of array dimensions. Examples representing the paradigms above are: Remora [67], Dex [47], and the recent extensions to Futhark [5, 31].

The array programming community is at an impasse: array types are either too simple, or we have to reach for dependent types.[1] My main contribution towards a solution is the design of a novel calculus, **Star**. Its type system provides useful and expressive types, while admitting ML-style type inference. The key idea of Star follows my thesis: to design its array indices and shapes with **structural subtyping** in mind.

In this chapter (based on a paper of the same name [4]), I describe the following contributions:

- A novel design for an array calculus – Star – which features subtyping and structural types for array shapes. I motivate it in Section 5.1 and elaborate in Section 5.2.
- A formalisation of with an operational semantics (Section 5.2.5), type system (Section 5.3; *without* parametric polymorphism), and proven type safety (Section 5.3.1).
- Finally, in Section 5.3.2 I bestow Star (*extended* with parametric polymorphism) with ML-style type inference using Warp (Chapter 3), testing it with Weaver's implementation (Chapter 4).

[1] This is reminiscent of the static-dynamic split from Chapter 2 – structural types also serve as a middle-ground, like in Star.

5.1 Design

We start by motivating the design of Star, showing how common array patterns arise when we use algebraic data types – records and variants – for array indices.

Standard terminology Arrays have a *rank* – the number of dimensions. The *shape* of an array defines the extent (*size*) of each dimension. Dimensions are sometimes referred to as *axes*. A *batch* refers to a sequence of something, and we speak of a batch dimension (batch axis).

5.1.1 Record indices label axes

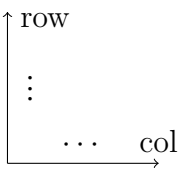
Consider an array representing a batch of images. We would model this as a 4-dimensional array, with a batch dimension, two coordinate dimensions, and a channel dimension. For instance, a size-100 batch of 32×32 RGB images might have shape $(100, 32, 32, 3)$.

We could instead consider a different interpretation for this array. Notice that a value of the record type:

$$\{\text{batch} : \text{int}, \text{row} : \text{int}, \text{col} : \text{int}, \text{channel} : \text{int}\}$$

could clearly be used as an index into our batch of images. In fact, this perspective has its benefits. For example, a simple 4-dimensional structure does not differentiate dimensions beyond their position in the sequence – the programmer would usually need to keep track of the order of dimensions with e.g. unenforceable annotations in code comments.

I propose that for any array shape types $\sigma, \ldots$ and axis labels $\ell, \ldots$ we should have a *product shape* $\{\!|\ell : \sigma, \ldots|\!\}$ indexed by records of type $\{\ell : \sigma, \ldots\}$. Records provide a **product** on shapes. An example illustration of $\{\!|\text{row} : \text{int}, \text{col} : \text{int}|\!\}$ is given on the right.



This idea is reminiscent of named tensors [18] or `xarray` [33] (among others), and this similarity is not accidental – their usefulness motivated generalisation to product shapes.

5.1.2 Variant indices concatenate

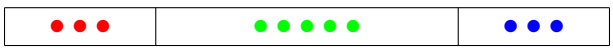
Having seen that indexing arrays with records models (labelled) multidimensional arrays, we turn our attention to the dual of records – variants. Consider an array indexed by values of the variant type

$$[\text{L} : \text{int}, \text{R} : \text{int}]$$

With variants, we compose a sequence of shapes, accessing any *one* of the components – like a **concatenation**.

Why is this perspective useful? Normally, manipulating concatenated arrays requires the programmer to perform arithmetic on index ranges. For instance, padding a vector of size n with a elements at the front and b at the back creates a vector composed of three ranges: front $[0, a)$, centre $[a, n + a)$, and back $[n + a, n + a + b)$. Normally, composing and decomposing a concatenated vector – e.g. setting and getting its components – requires reconstructing these ranges. Instead, let us use an index of variant type:

$$[\text{Front} : \text{int}, \text{Centre} : \text{int}, \text{Back} : \text{int}]$$



which allows us to rely on tagging and pattern matching for composing and decomposing concatenations. Thus, for shapes $\sigma, \ldots$ and tags $T, \ldots$ we should have a shape $[\![T : \sigma, \ldots]\!]$ indexed by variants $[T : \sigma, \ldots]$.

$$
\begin{array}{rll}
e & ::= n \mid f & \text{(integer and float literal)} \\
& \mid \pi(\overline{e}) & \text{(scalar operations)} \\
& \mid x \mid \text{let } x = e \text{ in } e & \text{(variable, let-binding)} \\
& \mid \lambda\, x.\, e \mid e\, e & \text{(function, application)} \\
& \mid \{\overline{\ell = e}\} & \text{(record construction)} \\
& \mid e.\ell & \text{(record projection)} \\
& \mid T\, e & \text{(variant construction)} \\
& \mid \text{match } e \text{ with } \overline{\mid T\, x \Rightarrow e} & \text{(pattern matching)} \\
& \mid \Phi\, x[e].\, e & \text{(array comprehension)} \\
& \mid \mathsf{Arr}(s, J) & \text{(unevaluated array)} \\
& \mid e[e] & \text{(array indexing)} \\
& \mid |e| & \text{(array shape)} \\
& \mid e^{\mathrm{S}} & \text{(shape expression)} \\
\\
e^{\mathrm{S}} & ::= \#e & \text{(sized shape)} \\
& \mid \{\!|\overline{\ell = e}|\!\} & \text{(product shape)} \\
& \mid e_{\circ}\ell & \text{(product dimension projection)} \\
& \mid [\![\overline{T = e}]\!] & \text{(concatenation shape)} \\
& \mid e_{\diamond}T & \text{(concatenation component projection)} \\
& \mid e \sqcap e & \text{(shape broadcasting)}
\end{array}
$$

Figure 5.1: Expressions e in Star, including shape expressions e^{S}.

5.2 Calculus

I now introduce the design of **Star** (**st**ructured **ar**rays), a functional, pointful array programming calculus with a novel type system. We begin by introducing expressions (Section 5.2.1) and our novel structured array shapes (Sections 5.2.2 and 5.2.3) in Star. Lastly, we give an operational semantics (Section 5.2.5).

5.2.1 Introduction

The basis of Star is a λ-calculus with structurally typed records and variants. We define the grammars of expressions e and values v in Star in Figures 5.1 and 5.2. We start with some informal explanations:

- We write π for scalar operations, which consume and produce integers and floats. Examples include $+ : \text{int} \times \text{int} \to \text{int}$, $\sin : \text{float} \to \text{float}$, or $\lfloor \cdot \rfloor : \text{float} \to \text{int}$.
- We include structural records and variants, respectively introduced by construction $\{\overline{\ell = e}\}$ and tagging $T\, e$, and eliminated with with projection $e.\ell$ and pattern matching.[2]

[2] Reflecting the paper [4], Star uses standard typing for records & variants (like Pierce [50]) – unlike FL and Fabric.

$$
\begin{array}{lr}
v ::= n \mid f & \text{(integer, float)} \\
\quad \mid \lambda\, x.\, e & \text{(function)} \\
\quad \mid \{\overline{\ell = v}\} & \text{(record)} \\
\quad \mid T\, v & \text{(variant)} \\
\quad \mid \mathsf{Arr}(s, I) & \text{(array)} \\
\quad \mid s & \text{(shape value)} \\
\\
s ::= \#n & \text{(sized shape)} \\
\quad \mid \{\!|\overline{\ell = s}|\!\} & \text{(product shape)} \\
\quad \mid [\![\overline{T = s}]\!] & \text{(concatenation shape)}
\end{array}
$$

Figure 5.2: Values v in Star, including shape values s.

- Star's array operations are similar to the likes of pointful calculi like $\tilde{F}$ [64]. Our primitives are array comprehensions $\Phi\, x[e].\, e'$ (constructing an array of shape e and elements at index x given by e') and indexing $e[e']$ (accessing index e' of array e). The shape of an array e is accessed by $|e|$.

5.2.2 Array shapes

The design of array shapes in Star – both shape values here and shape types in Section 5.3 – is a novel contribution. In Star, array comprehensions require providing a shape. We informally summarise them now and elaborate later.

Firstly, we have three constructors for shape values:

- A sized shape $\#e$ is used for a usual flat array. For example, $\#10$ is the shape of an array indexed by integers in the range $[0, 10)$.

- Product shapes are used for defining arrays with multiple *named* dimensions. For instance:

  $$\{\!|\mathrm{row} = \#5, \mathrm{col} = \#4|\!\}$$

  is the shape of a 5×4 matrix (with dimensions named as rows and columns). To index into an array of a product shape, we provide a record – e.g. $\{\mathrm{row} = 2, \mathrm{col} = 3\}$. Products capture *rectangular* (regular) multidimensional arrays. Note that an array of shape $\{\!|\}\!|$ has a single element.

- Concatenation shapes are given by a sequence of *named* component shapes. For example,

  $$[\![\mathrm{Left} = \{\!|\}\!|, \mathrm{Centre} = \#8, \mathrm{Right} = \{\!|\}\!|]\!]$$

  describes an 8-element array with an extra element at the start and end (i.e. a halo/padding/boundary of size 1). We index into arrays of concatenation shapes with variants, e.g. $\mathrm{Left}\ \{\}$ or $\mathrm{Centre}\ 2$. Note that an array of shape $[\![]\!]$ is empty.

$$\begin{aligned} \#n \bowtie \#m &= \#n && \text{if } n = m \\ \{\!|\overline{\ell' = s'_\ell}|\!\} \bowtie \{\!|\overline{\ell'' = s''_\ell}|\!\} &= \{\!|\overline{\ell = s'_\ell \bowtie s''_\ell}|\!\} && \text{for } \ell = \ell' \cup \ell'' \\ [\![\overline{T = s'_T}]\!] \bowtie [\![\overline{T = s''_T}]\!] &= [\![\overline{T = s'_T \bowtie s''_T}]\!] \end{aligned}$$

Figure 5.3: Semantics $\bowtie$ of shape broadcasting $\sqcap$. We take the union of sets of dimensions ℓ (defaulting $s'_\ell = s''_\ell$ if either is absent), but equate sets of components T. If $s' \bowtie s''$ is undefined, we say s' and s'' are *incompatible*, and $s' \sqcap s''$ is a stuck state in the operational semantics.

Projections on shapes extract dimensions/components of products/concatenations. For example:

$$\{\!|\mathrm{row} = \#5, \mathrm{col} = \#4|\!\}_\circ \mathrm{row} \rightsquigarrow \#5$$
$$[\![\mathrm{Left} = \#1, \mathrm{Right} = \#0]\!]_\diamond \mathrm{Left} \rightsquigarrow \#1$$

Lastly, Star's *shape broadcasting* operation $e \sqcap e'$ (see Figure 5.3) is used to *align* compatible shapes e and e' by finding their common *sub-shape* (cf. broadcasting in NumPy [30]), so that, for example:

$$\{\!|\mathrm{row} : \#5|\!\} \sqcap \{\!|\mathrm{col} : \#4|\!\} \text{ evaluates to } \{\!|\mathrm{row} : \#5, \mathrm{col} : \#4|\!\}$$

The name sub-shape refers to the result having possibly more dimensions, paralleling subtyping.

5.2.3 Shape bounds

We now formalise what indices v can be used to index into a given shape s by using the *in-bounds* relation $v \lhd s$, defined in Figure 5.4.

For integers, in-bounds behaves as expected. However, structural indices have more interesting behaviour thanks to subtyping: for products, an index with more dimensions than the shape can be used for indexing; for concatenations any component works. To manipulate these extra dimensions, we define a *cast* operator $v \oslash s$ in Figure 5.5, which removes extra dimensions from v, coercing an index to fit the structure of a given shape exactly. We then define the *structurally-exact in-bounds* relation $v \trianglelefteq s$ as:

$$v \trianglelefteq s \iff v \lhd s \text{ and } v = v \oslash s$$

Lastly, we introduce a *structurally-in-bounds* relation $v \blacktriangleleft s$, given by the same rules as $\lhd$ but with $n \blacktriangleleft \#m$ holding for **any** n and m (and not just $0 \le n < m$). We use this to state type safety later – we only guarantee successful indexing up to integer indices.

Let us consider some examples. Both $\{x = 3\} \lhd \{\!|x = \#4|\!\}$ and $\{x = 3\} \trianglelefteq \{\!|x = \#4|\!\}$ hold. While we have $\{x = 3\} \lhd \{\!|\,|\!\}$, $\{x = 3\} \trianglelefteq \{\!|\,|\!\}$ is false (since $\{x = 3\} \oslash \{\!|\,|\!\} = \{\}$) and only $\{\} \trianglelefteq \{\!|\,|\!\}$ holds.

In the operational semantics, $\trianglelefteq$ determines the indices v for which we should compute and store the elements when building an array. $v \oslash s$ is used instead of v for indexing, but only when $v \lhd s$.

5.2.4 Array values

In Star, array comprehensions $\Phi\, x[e].\, e'$ eventually evaluate to array values $\mathsf{Arr}(s, I)$. These consist of a shape value s along with a partial function (taken as a set) I from index values to element values. The domain of I consists of exactly the indices v which are within bounds of s: $v \trianglelefteq s$.

$$\frac{}{n \lhd \#m} \quad \text{if } 0 \le n < m$$

$$\frac{\forall \ell'.\, v_{\ell'} \lhd s_{\ell'}}{\{\overline{\ell = v_\ell}\} \lhd \{|\overline{\ell' = s_{\ell'}}|\}} \qquad \frac{v \lhd s_T}{T\, v \lhd [\![\overline{T = s_T}]\!]}$$

Figure 5.4: Definition of the in-bounds relation $v \lhd s$, stating that an index v can be used to index into a shape s.

$$\begin{aligned} n \oslash \#m &= n \\ \{\overline{\ell = v_\ell}, \overline{\ell' = v'_\ell}\} \oslash \{|\overline{\ell = s_\ell}|\} &= \{\overline{\ell = v_\ell \oslash s_\ell}\} \\ T\, v \oslash [\![\overline{T' = v_{T'}}]\!] &= T\,(v \oslash v_T) \end{aligned}$$

Figure 5.5: Definition of the index-shape cast $v \oslash s$, which removes dimensions from v which do not appear in s.

However, evaluating $\Phi\, x[e].\, e'$ generally requires multiple evaluations of e'; our small-step operational semantics achieves this using *unevaluated array expressions* $\mathsf{Arr}(s, J)$. These generalise array values above, allowing the codomain of J to hold expressions instead of values. Operationally, array comprehensions Φ yield array expressions $\mathsf{Arr}(s, J)$, which are evaluated until they become values $\mathsf{Arr}(s, I)$. $\mathsf{Arr}(s, J)$ is not part of the user-facing expression syntax – it exists solely to enable a *small-step* operational semantics, thus simplifying the statement and proofs of type safety.

5.2.5 Semantics

Finally, I present a call-by-value operational semantics for Star in Figure 5.7 via a *steps-to* relation $e \rightsquigarrow e$.

Run-time errors (e.g. out-of-bounds indexing) manifest as *stuck states*, which do not step further. Most stuck states are prevented by Star's type system – for errors which are not, we introduce a *raises-error* relation $e \rightsquigarrow \lightning$ in Figure 5.6 (we use it to state type safety in Section 5.3.1).

Successful executions eventually reduce to a value v (Figure 5.2).

$$\boxed{e \rightsquigarrow \lightning}$$

$$\mathsf{Arr}(s, I)[v] \rightsquigarrow \lightning \quad \text{if } (v \blacktriangleleft s) \wedge \neg(v \lhd s) \qquad \textsc{StepIndexErr}$$

$$s' \sqcap s'' \rightsquigarrow \lightning \quad \text{if } s' \bowtie s'' \text{ is undefined} \qquad \textsc{StepBroadcastErr}$$

$$\textsc{CongErr}\ \frac{e \rightsquigarrow \lightning}{C\langle e \rangle \rightsquigarrow \lightning}$$

Figure 5.6: Raises-error step relation $e \rightsquigarrow \lightning$, used for specifying errors which *can* occur in well-typed Star programs.

$$\boxed{e \rightsquigarrow e}$$

$$
\begin{array}{rll}
(\lambda\, x.\, e)\, v \rightsquigarrow [v/x]e & & \textsc{StepApply} \\
\mathrm{let}\, x = v\, \mathrm{in}\, e \rightsquigarrow [v/x]e & & \textsc{StepLet} \\
\{\overline{\ell = v_\ell}\}.\ell \rightsquigarrow v_\ell & & \textsc{StepRecordProj} \\
\mathrm{match}\, T\, v\, \mathrm{with}\, T\, x \Rightarrow e \mid \cdots \rightsquigarrow [v/x]e & & \textsc{StepMatch} \\
\Phi\, x[s].\, e \rightsquigarrow \mathsf{Arr}(s, \{v \mapsto [v/x]e \mid v \trianglelefteq s\}) & & \textsc{StepArray} \\
\mathsf{Arr}(s, I)[v] \rightsquigarrow I(v \oslash s) \quad (\text{if } v \lhd s) & & \textsc{StepIndex} \\
|\mathsf{Arr}(s, I)| \rightsquigarrow s & & \textsc{StepShape} \\
\{|\overline{\ell = v_\ell}|\}_\diamond \ell \rightsquigarrow v_\ell & & \textsc{StepProductProj} \\
[\![\overline{T = v_T}]\!]_\diamond T \rightsquigarrow v_T & & \textsc{StepConcatProj} \\
s' \sqcap s'' \rightsquigarrow s' \bowtie s'' & & \textsc{StepBroadcast}
\end{array}
$$

$$\textsc{Cong}\quad \frac{e \rightsquigarrow e'}{C\langle e \rangle \rightsquigarrow C\langle e' \rangle}$$

Figure 5.7: Small-step operational semantics for Star. We write $e \rightsquigarrow e'$ when e reduces to e'. Evaluation contexts $C\langle\diamond\rangle$ are defined in Figure 5.8 – we never evaluate under λ or Φ; but we evaluate expressions in $\mathsf{Arr}(s, J)$. Indexing and broadcasting may result in a *stuck state* – but note $v \oslash s$ is well-defined when $v \lhd s$ holds.

$$
\begin{array}{rl}
C\langle\diamond\rangle ::= & \diamond \\
& \mid \pi(\overline{v}, C\langle\diamond\rangle, \overline{e}) \\
& \mid \mathrm{let}\, x = C\langle\diamond\rangle\, \mathrm{in}\, e \\
& \mid C\langle\diamond\rangle\, e \mid v\, C\langle\diamond\rangle \\
& \mid \{\overline{\ell = v}, \ell = C\langle\diamond\rangle, \overline{\ell = e}\} \mid C\langle\diamond\rangle.\ell \\
& \mid T\, C\langle\diamond\rangle \mid \mathrm{match}\, C\langle\diamond\rangle\, \mathrm{with}\, \overline{\mid T\, x\, \Rightarrow e} \\
& \mid \Phi\, x[C\langle\diamond\rangle].\, e \\
& \mid \mathsf{Arr}(s, \{v \mapsto C\langle\diamond\rangle\} \cup J) \\
& \mid C\langle\diamond\rangle[e] \mid v[C\langle\diamond\rangle] \mid |C\langle\diamond\rangle| \\
& \mid \#C\langle\diamond\rangle \\
& \mid \{|\overline{\ell = v}, \ell = C\langle\diamond\rangle, \overline{\ell = e}|\} \mid C\langle\diamond\rangle_\diamond \ell \\
& \mid [\![\overline{\ell = v}, \ell = C\langle\diamond\rangle, \overline{\ell = e}]\!] \mid C\langle\diamond\rangle_\diamond T \\
& \mid C\langle\diamond\rangle \sqcap e \mid v \sqcap C\langle\diamond\rangle
\end{array}
$$

Figure 5.8: Evaluation contexts C for Star, necessary for specifying the congruence rules for $\rightsquigarrow$.

$$
\begin{array}{rll}
\tau & ::= \top \mid \bot & \text{(top, bottom)} \\
& \mid \text{int} \mid \text{float} & \text{(integer, float)} \\
& \mid \tau \to \tau & \text{(function)} \\
& \mid \{\overline{\ell : \tau}\} & \text{(record)} \\
& \mid [\overline{T : \tau}] & \text{(variant)} \\
& \mid [\sigma_1 .. \sigma_2]\tau \quad \text{for } \sigma_1 \leqslant \sigma_2 & \text{(array)} \\
& \mid \sigma & \text{(shape)} \\
\\
\sigma & ::= \# & \text{(sized shape)} \\
& \mid \{\!|\overline{\ell : \sigma}|\!\} & \text{(product shape)} \\
& \mid [\![\overline{T : \sigma}]\!] & \text{(concatenation shape)} \\
\\
\eta & ::= \text{int} \mid \{\overline{\ell : \eta}\} \mid [\overline{T : \eta}] & \text{(indices in } \tau\text{)}
\end{array}
$$

Figure 5.9: Types τ in Star, showing their subtypes: shape types σ and index types η. Array types $[\sigma_1..\sigma_2]\tau$ are defined only when $\sigma_1 \leqslant \sigma_2$ (with $\leqslant$ as in Figure 5.10) – we elaborate in Section 5.3.

Shape		Index	
Type	Value	Type	Value
$\#$	$\#5$	int	4
$\{\!\|r : \#, c : \#\|\!\}$	$\{\!\|r = \#5, c = \#4\|\!\}$	$\{r : \text{int}, c : \text{int}\}$	$\{r = 4, c = 3\}$
$[\![L : \#, R : \#]\!]$	$\{\!\|L = \#10, R = \#20\|\!\}$	$[L : \text{int}, R : \text{int}]$	$R\,19$

Table 5.1: Example shapes with matching indices. For a shape of type σ on the left, an index of type $\iota(\sigma)$ is on the right.

5.3 Typing

We move on to Star's type system, which features novel aspects matching the design of structural shape and index types. I give a complete description of the type language in Figure 5.9, subtyping in Figure 5.10, and the typing judgement in Figure 5.11.

I first focus on the typing of array shapes and indices. Afterwards, I state type safety for Star in Section 5.3.1, and show how to apply the type inference framework from Chapter 3 to Star in Section 5.3.2.

Shapes and indices Recall that in Section 5.2.3 we determined that integers index into sized shapes, records into product shapes, and variants into concatenation shapes. This happens both at the value- and type-level, with examples in Table 5.1.

While in basic array type systems it is sufficient to presume that the type of a shape σ[3] and a matching index η are the same, in Star this is clearly not the case. We thus use a ι metafunction on types (defined in Figure 5.12): given a shape type σ, $\eta = \iota(\sigma)$ is the supertype of all matching index types for σ.

[3]Note that in Chapter 3 we denoted type schemes with σ, as is standard. In this chapter, except for one use in Section 5.3.2, σ is always a shape type – matching the paper.

$$\boxed{\tau \leqslant \tau}$$

$$\textsc{SubRefl}\ \frac{}{\tau \leqslant \tau} \qquad \textsc{SubTrans}\ \frac{\tau \leqslant \tau' \quad \tau' \leqslant \tau''}{\tau \leqslant \tau''} \qquad \textsc{SubTop}\ \frac{}{\tau \leqslant \top} \qquad \textsc{SubBot}\ \frac{}{\bot \leqslant \tau}$$

$$\textsc{SubFun}\ \frac{\tau_1' \leqslant \tau_1 \quad \tau_2 \leqslant \tau_2'}{(\tau_1 \to \tau_2) \leqslant (\tau_1' \to \tau_2')} \qquad \textsc{SubArray}\ \frac{\sigma_1' \leqslant \sigma_1 \quad \sigma_2 \leqslant \sigma_2' \quad \tau \leqslant \tau'}{[\sigma_1..\sigma_2]\tau \leqslant [\sigma_1'..\sigma_2']\tau'}$$

$$\textsc{SubRecWidth}\ \frac{}{\{\ell : \tau, \rho\} \leqslant \{\rho\}} \qquad \textsc{SubRecDepth}\ \frac{\tau \leqslant \tau'}{\{\ell : \tau, \rho\} \leqslant \{\ell : \tau', \rho\}} \qquad \textsc{SubProdWidth}\ \frac{}{\{\!|\ell : \tau, \rho|\!\} \leqslant \{\!|\rho|\!\}} \qquad \textsc{SubProdDepth}\ \frac{\tau \leqslant \tau'}{\{\!|\ell : \tau, \rho|\!\} \leqslant \{\!|\ell : \tau', \rho|\!\}}$$

$$\textsc{SubVarWidth}\ \frac{}{[\rho] \leqslant [T : \tau, \rho]} \qquad \textsc{SubVarDepth}\ \frac{\tau \leqslant \tau'}{[T : \tau, \rho] \leqslant [T : \tau', \rho]} \qquad \textsc{SubConcatWidth}\ \frac{}{[\![T : \tau, \rho]\!] \leqslant [\![\rho]\!]} \qquad \textsc{SubConcatDepth}\ \frac{\tau \leqslant \tau'}{[\![T : \tau, \rho]\!] \leqslant [\![T : \tau', \rho]\!]}$$

Figure 5.10: Subtyping relation $\tau \leqslant \tau$. SubArray is particularly noteworthy, as it mixes contravariance of indices and covariance of shape-access. We write ρ for a list of entries in a type (like fields of a record).

$$\boxed{\Gamma \vdash e : \tau}$$

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

$$\textsc{Sub}\ \frac{\Gamma \vdash e : \tau \quad \tau \leqslant \tau'}{\Gamma \vdash e : \tau'} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash n : \mathrm{int}} \qquad \textsc{Float}\ \frac{}{\Gamma \vdash f : \mathrm{float}}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathrm{let}\, x = e\, \mathrm{in}\, e' : \tau'} \qquad \textsc{Lambda}\ \frac{\Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \lambda\, x.\, e : \tau \to \tau'} \qquad \textsc{Apply}\ \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}$$

$$\textsc{Record}\ \frac{\forall \ell.\, \Gamma \vdash e_\ell : \tau_\ell}{\Gamma \vdash \{\overline{\ell = e_\ell}\} : \{\overline{\ell : \tau_\ell}\}} \qquad \textsc{RecordProj}\ \frac{\Gamma \vdash e : \{\ell : \tau\}}{\Gamma \vdash e.\ell : \tau} \qquad \textsc{Tag}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash T\, e : [T : \tau]} \qquad \textsc{Match}\ \frac{\Gamma \vdash e : \boxed{T : \tau_T}\ \Gamma, x_T{:}\tau_T \vdash e_T : \tau}{\Gamma \vdash \mathrm{match}\, e\, \mathrm{with}\, \overline{T\, x_T \Rightarrow e_T} : \tau}$$

$$\textsc{Array}\ \frac{\Gamma \vdash e : \sigma \quad \Gamma, x : \iota(\sigma) \vdash e' : \tau}{\Gamma \vdash \Phi\, x[e].\, e' : [\sigma]\tau} \qquad \textsc{ArrayLit}\ \frac{\Gamma \vdash s : \sigma \quad \forall (v \mapsto e) \in J.\, \Gamma \vdash v : \iota(\sigma)\ \text{and}\ \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Arr}(s, J) : [\sigma]\tau}$$

$$\textsc{Index}\ \frac{\Gamma \vdash e : [\sigma_1..\sigma_2]\tau \quad \Gamma \vdash e' : \iota(\sigma_1)}{\Gamma \vdash e[e'] : \tau} \qquad \textsc{Shape}\ \frac{\Gamma \vdash e : [\sigma_1..\sigma_2]\tau}{\Gamma \vdash |e| : \sigma_2} \qquad \textsc{Broadcast}\ \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash e' : \sigma'}{\Gamma \vdash e \sqcap e' : \sigma \wedge \sigma'} \qquad \textsc{Sized}\ \frac{\Gamma \vdash e : \mathrm{int}}{\Gamma \vdash \#e : \#}$$

$$\textsc{Product}\ \frac{\forall \ell.\, \Gamma \vdash e_\ell : \sigma_\ell}{\Gamma \vdash \{\!|\overline{\ell = e_\ell}|\!\} : \{\!|\overline{\ell : \sigma_\ell}|\!\}} \qquad \textsc{ProductProj}\ \frac{\Gamma \vdash e : \{\!|\ell : \sigma|\!\}}{\Gamma \vdash e_\circ \ell : \sigma} \qquad \textsc{Concat}\ \frac{\forall T.\, \Gamma \vdash e_T : \sigma_T}{\Gamma \vdash [\![\overline{T = e_T}]\!] : [\![\overline{T : \sigma_T}]\!]} \qquad \textsc{ConcatProj}\ \frac{\Gamma \vdash e : [\![T : \sigma]\!]}{\Gamma \vdash e_\diamond T : \sigma}$$

Figure 5.11: Rules for the typing judgement $\Gamma \vdash e : \tau$ for Star expressions e. We use σ for metavariables standing for shape types. Note ι in Array and Index maps shape types σ into corresponding index types η. Quantifiers $\forall \ell / \forall T$ stand for an expanded list of assumptions instantiated for all appropriate ℓ / T.

$$\begin{aligned}\iota(\#) &= \mathrm{int}\\ \iota\big(\{\!|\overline{\ell : \sigma_\ell}|\!\}\big) &= \left\{\overline{\ell : \iota(\sigma_\ell)}\right\}\\ \iota\big([\![\overline{T : \sigma_T}]\!]\big) &= \left[\overline{T : \iota(\sigma_T)}\right]\end{aligned}$$

Figure 5.12: Definition of the metafunction ι. For a shape type σ, $\eta = \iota(\sigma)$ is the least upper bound on its index type.

Constructing ι is unsurprising – similar concepts often arise when generalising array-like structures, e.g. *containers* [2] or Naperian (representable) functors, which Gibbons [27] has found useful for statically modelling certain patterns of array programming. To the author's knowledge, neither of these had led to the key ideas present in Star.

Splitting σ Consider Star's array types, which have the syntax $[\sigma_1..\sigma_2]\tau$ for $\sigma_1 \leqslant \sigma_2$. They are carefully constructed to provide useful subtyping with a technique used by both Dolan [20] and Pottier [53] (which they use for typing mutable reference types). The crux is to split the shape σ in $[\sigma]\tau$ into a contravariant part σ_1 and covariant part σ_2, so that indexing uses σ_1, and accessing the shape returns a σ_2. The obvious array type, $[\sigma]\tau$ (which we write to abbreviate $[\sigma..\sigma]\tau$), would necessarily be invariant [4, Appendix B]. Note that by requiring $\sigma_1 \leqslant \sigma_2$ there is a σ such that $\sigma_1 \leqslant \sigma \leqslant \sigma_2$: we have an *actual* shape σ into which we index with $\iota(\sigma_1) \leqslant \iota(\sigma)$, and but accessing it we only obtain $\sigma_2 \geqslant \sigma$.

Broadcasting We view broadcasting as a meet in the lattices of both shape bounds and their types. Indeed, $s' \bowtie s''$ (for shape values $s' : \sigma'$, $s'' : \sigma''$ of types σ', σ'') is a partial operator that finds the shape $s : \sigma$ for which $\sigma = \sigma' \wedge \sigma''$ (where $\wedge$ is the meet on types agreeing with $\leqslant$), so that for any index v:

$$v \lhd s \iff v \lhd s' \text{ and } v \lhd s''$$

Hence, $\sqcap$ is the meet in a lattice spanned by the unary predicate $\_ \lhd s$, and its result type takes the meet of shape types: clarifying how broadcasting finds sub-shapes, as defined in Section 5.2.2.

5.3.1 Type safety

To show my design of Star is well-behaved, I state and prove type safety for it in the form of theorems of Preservation and Progress (in the simply-typed case). Stating Progress we write $e \rightsquigarrow \lightning$, which captures errors that can occur for well-typed programs – out-of-bounds *integer* indexing and invalid broadcasting.

Theorem 5.1 (Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : \tau$*.*

Theorem 5.2 (Progress)**.** *If* $\cdot \vdash e : \tau$*, then either* e *is a value,* $e \rightsquigarrow e'$ *for some* e'*, or* $e \rightsquigarrow \lightning$*.*

I prove these theorems in Appendix F, but they follow straightforwardly by structural induction. Crucially, I show agreement between the subtyping relation $\leqslant$ and the structurally-in-bounds relation $\blacktriangleleft$.

5.3.2 Type inference & polymorphism

Star's subtyping forms a distributive lattice,[4] and ι is an isomorphism – we can apply Warp (Chapter 3).

Though I do not give a complete description of a polymorphic Star calculus, I show a sketch of constraint generation for Star in Figure 5.13. To do so, we extend ι to be an involutive isomorphism in the algebra of types ($\iota^{-1} = \iota$, i.e. ι swaps shapes and indices), so that $\overleftarrow{\iota} = \overrightarrow{\iota} = \iota$ and $\underleftarrow{\iota} = \underrightarrow{\iota} = \mathbf{T}$.

Extending Star with polymorphism enables a *shape-polymorphic* programming style – a generalisation of rank polymorphism in existing literature [61, 63]. It is worth noting that shape polymorphism gives us index safety for free akin to Wadler [72] (modulo integer indices).

Example

Given:

$$e = \lambda a.\, \lambda b.\, \Phi\, x[\{\!|a = \#5, b = \#4|\!\}].\, \pi(a[x.a], b[x.b])$$

Then Warp (at $\pi : \text{float} \times \text{float} \to \text{float}$) infers the type scheme:

$$\sigma = [\#..\top]\text{float} \to [\#..\top]\text{float} \to [\{\!|a : \#, b : \#|\!\}]\text{float}$$

Further, given $e' = \Phi\, x[\{\!|a = \#5|\!\}].\, x.b$ a type error is raised from $\iota(\{b : \text{int}\}) \leqslant \{\!|a : \#|\!\}$.
Usability is mainly limited by type scheme simplification – a shortcoming discussed in Section 4.2.1.

5.4 Case study

We consider a case study on *padding arrays* – showing how Star's elegance shines even in common use-cases.

Consider padding a matrix with l left and r right columns, as well as t top and b bottom rows. Padding should have value -1 at top, $+1$ at bottom, and 0 elsewhere. For instance, for $l = t = b = 1$ and $r = 2$:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \mapsto \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 \\ 0 & 1 & 2 & 3 & 0 & 0 \\ 0 & 4 & 5 & 6 & 0 & 0 \\ +1 & +1 & +1 & +1 & +1 & +1 \end{bmatrix}$$

Programming this can be inconvenient with unstructured array types: an array of shape (n, m) becomes one of shape $(n + t + b, m + l + r)$, burdening us with index manipulation. Consider the example of NumPy – to pad a `numpy.ndarray` named `x` one might write:

```
n, m = x.shape
p = numpy.empty((n + t + b, m + l + r))
p[t:n+t, l:m+l] = x[:, :]
p[:t, :] = -1;    p[n+t:, :] = +1
p[t:n+t, :l] = 0; p[t:n+t, n+l:] = 0
```

The programmer needs to do menial mental arithmetic to find (unsatisfyingly *asymmetric*) index ranges.

[4] The trickiest aspect is that joins/meets are closed for arrays. Checking $[\sigma'_1..\sigma'_2]\tau' \vee [\sigma''_1..\sigma''_2]\tau'' = [\sigma'_1 \wedge \sigma''_1..\sigma'_2 \vee \sigma''_2](\tau' \vee \tau'')$, indeed $\sigma'_1 \leqslant \sigma'_2$ and $\sigma''_1 \leqslant \sigma''_2$ implies $\sigma'_1 \wedge \sigma''_1 \leqslant \sigma'_2 \wedge \sigma''_2 \leqslant \sigma'_2 \vee \sigma''_2$.

$$\boxed{\langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle = c}$$

$$\begin{aligned}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle &= \Gamma(x) \leqslant \tau && \textsc{Var} \\
\langle\!\langle \Gamma \vdash n : \tau \rangle\!\rangle &= \mathrm{int} \leqslant \tau && \textsc{Int} \\
\langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle &= \mathrm{float} \leqslant \tau && \textsc{Float} \\
\langle\!\langle \Gamma \vdash \mathrm{let}\, x = e' \,\mathrm{in}\, e : \tau \rangle\!\rangle &= \exists \tau'.\, \langle\!\langle \Gamma \vdash e' : \tau' \rangle\!\rangle \;\&\; \langle\!\langle \Gamma, e' : \tau' \vdash e : \tau \rangle\!\rangle && \textsc{Let} \\
\langle\!\langle \Gamma \vdash \lambda x.\, e : \tau \rangle\!\rangle &= \exists \tau', \tau''.\, \langle\!\langle \Gamma, x : \tau' \vdash e : \tau'' \rangle\!\rangle \;\&\; \tau \leqslant \tau' \to \tau'' && \textsc{Lambda} \\
\langle\!\langle \Gamma \vdash e\, e' : \tau \rangle\!\rangle &= \exists \tau'.\, \langle\!\langle \Gamma \vdash e' : \tau' \rangle\!\rangle \;\&\; \langle\!\langle \Gamma \vdash e : \tau' \to \tau \rangle\!\rangle && \textsc{Apply} \\
\left\langle\!\!\left\langle \Gamma \vdash \left\{\overline{\ell = e_\ell}\right\} : \tau \right\rangle\!\!\right\rangle &= \exists \overline{\tau_\ell}.\, \overline{\langle\!\langle \Gamma \vdash e_\ell : \tau_\ell \rangle\!\rangle} \;\&\; \left\{\overline{\ell : \tau_\ell}\right\} \leqslant \tau && \textsc{Record} \\
\langle\!\langle \Gamma \vdash e.\ell : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e : \{\ell : \tau\} \rangle\!\rangle && \textsc{RecordProj} \\
\langle\!\langle \Gamma \vdash T\, e : \tau \rangle\!\rangle &= \exists \tau_T.\, \langle\!\langle \Gamma \vdash e : \tau_T \rangle\!\rangle \;\&\; [T : \tau_T] \leqslant \tau && \textsc{Tag} \\
\left\langle\!\!\left\langle \Gamma \vdash e\, \overline{T\, x_T \Rightarrow e_T} : \tau \right\rangle\!\!\right\rangle &= \exists \overline{\tau_T}.\, \overline{\langle\!\langle \Gamma, x_T : \tau_T \vdash e_T : \tau \rangle\!\rangle} \;\&\; \left\langle\!\!\left\langle \Gamma \vdash e : \left[\overline{T : \tau_T}\right] \right\rangle\!\!\right\rangle && \textsc{Match} \\
\langle\!\langle \Gamma \vdash \Phi\, x[e'].\, e'' : \tau \rangle\!\rangle &= \exists \tau''.\, \langle\!\langle \Gamma \vdash e' : \tau' \rangle\!\rangle \& \langle\!\langle \Gamma, x{:}\iota(\tau') \vdash e'' : \tau'' \rangle\!\rangle \;\&\; [\tau']\tau'' \leqslant \tau && \textsc{Array} \\
\langle\!\langle \Gamma \vdash e[e'] : \tau \rangle\!\rangle &= \exists \tau_1, \tau_2.\, \langle\!\langle \Gamma \vdash e : [\tau_1..\tau_2]\tau \rangle\!\rangle \;\&\; \left\langle\!\!\left\langle \Gamma \vdash e' : \iota^{-1}(\tau_1) \right\rangle\!\!\right\rangle && \textsc{Index} \\
\langle\!\langle \Gamma \vdash |e| : \tau_2 \rangle\!\rangle &= \exists \tau_1, \tau.\, \langle\!\langle \Gamma \vdash e : [\tau_1..\tau_2]\tau \rangle\!\rangle && \textsc{Shape} \\
\langle\!\langle \Gamma \vdash e' \sqcap e'' : \tau \rangle\!\rangle &= \exists \tau', \tau''.\, \langle\!\langle \Gamma \vdash e' : \tau' \rangle\!\rangle \;\&\; \langle\!\langle \Gamma \vdash e'' : \tau'' \rangle\!\rangle \;\&\; \tau' \wedge \tau'' \leqslant \tau && \textsc{Broadcast} \\
\langle\!\langle \Gamma \vdash \#e : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e : \mathrm{int} \rangle\!\rangle \;\&\; \# \leqslant \tau && \textsc{Sized} \\
\left\langle\!\!\left\langle \Gamma \vdash \left\{\!\!\left|\overline{\ell = e_\ell}\right|\!\!\right\} : \tau \right\rangle\!\!\right\rangle &= \exists \overline{\tau_\ell}.\, \overline{\langle\!\langle \Gamma \vdash e_\ell : \tau_\ell \rangle\!\rangle} \;\&\; \left\{\!\!\left|\overline{\ell : \tau_\ell}\right|\!\!\right\} \leqslant \tau && \textsc{Product} \\
\langle\!\langle \Gamma \vdash e_\circ \ell : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e : \{\!|\ell : \tau|\!\} \rangle\!\rangle && \textsc{ProductProj} \\
\left\langle\!\!\left\langle \Gamma \vdash \left[\!\!\left[\overline{\ell = e_\ell}\right]\!\!\right] : \tau \right\rangle\!\!\right\rangle &= \exists \overline{\tau_\ell}.\, \overline{\langle\!\langle \Gamma \vdash e_\ell : \tau_\ell \rangle\!\rangle} \;\&\; \left[\!\!\left[\overline{\ell : \tau_\ell}\right]\!\!\right] \leqslant \tau && \textsc{Concat} \\
\langle\!\langle \Gamma \vdash e_\diamond \ell : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e : [\![\ell : \tau]\!] \rangle\!\rangle && \textsc{ConcatProj}
\end{aligned}$$

Figure 5.13: Definition of Warp constraint generation for Star, defined inductively on e. Each case here corresponds to a typing rule of the same name. Note that Broadcast breaks the polarity restriction, and that Array and Index use the ι homomorphism.

Finally, we consider using Star. Intuitively, first note the shape of a padded matrix has different *regions*:

$$\begin{array}{c} \\ \left[\begin{array}{ccc} \text{top left} & \text{top centre} & \text{top right} \\ \text{centre left} & \text{centre centre} & \text{centre right} \\ \text{bot left} & \text{bot centre} & \text{bot right} \end{array}\right] \end{array} \begin{array}{c} \text{col} \\ \\ \text{row} \end{array}$$

These correspond to index ranges used prior. For instance, centre right is in range $[t, n{+}t) \times [n{+}l, n{+}l{+}r)$. With this intuition, we find the Star shape type σ – and its value s:

$$\begin{aligned} \sigma = \{\!|\, &\text{row} : [\![\text{Top} : \#, \text{Centre} : \#, \text{Bottom} : \#]\!], \\ &\text{col} : [\![\text{Left} : \#, \text{Centre} : \#, \text{Right} : \#]\!] \,|\!\} \end{aligned}$$

$$\begin{aligned} s = \{\!|\, &\text{row} = [\![\text{Top} = \#t, \text{Centre} = |a|_{\circ}\text{row}, \text{Bottom} = \#b]\!], \\ &\text{col} = [\![\text{Left} = \#l, \text{Centre} = |a|_{\circ}\text{col}, \text{Right} = \#r]\!] \,|\!\} \end{aligned}$$

and easily construct the padding of an array $a : [\{|\text{row} : \#, \text{col} : \#|\}]\text{int}$:

$$\begin{aligned} &\Phi\, x[s].\ \text{match}\, x.\text{row}, x.\text{col}\ \text{with} \\ &\quad \mid \text{Centre}\, i, \text{Centre}\, j \Rightarrow a[\{\text{row} = i, \text{col} = j\}] \\ &\quad \mid \text{Top}\ \_, \_ \Rightarrow -1 \\ &\quad \mid \text{Bot}\ \_, \_ \Rightarrow +1 \\ &\quad \mid \_, \_ \Rightarrow 0 \end{aligned}$$

Satisfyingly, index arithmetic (which most existing array languages would have us rely on) has become pattern matching – and the program's safety is guaranteed.

5.5 Conclusions

I have presented a novel design for an array programming calculus, emphasising the use of structural types for shapes and indices. Though I do not statically ensure bounds checking for *integer* indices – like dependent types – I circumvent the problem with *structured* array shapes and indices. This makes it much easier to provide useful static types to common array programming patterns – bringing down conventions for writing nearly-untyped array code.

In addition to contributions presented in this chapter, in the paper [4] we give further examples and avenues for further work.

My novel type system for Star fills the gap between dynamically and dependently typed array programming. Thus, I reveal a new area in the design space of array programming languages – a middle-ground between basic and complicated type systems.

6 Conclusions

I set out to show that structural subtyping is an interesting and rich approach to designing flexible type systems. I addressed this with techniques from both theory of programming languages and practical design.

From the theoretical side, I demonstrated that structural subtyping takes us closer towards the flexibility of dynamic languages by giving the FJ-FL translation (Chapter 2). I built a constraint-based type inference framework for languages with structural subtyping, **Warp**, deriving from *algebraic subtyping* (Chapter 3). It generalises prior work and has a simpler presentation, giving clear requirements for the type system. By extending the type language of existing systems in a novel way (with *homomorphism applications*), I unlocked a new solution to the record update problem. Thus, I widened the range of language features that can be conveniently type-checked without annotations.

From the practical side, I designed a functional language with structural subtyping – **Fabric** (Chapter 4). I implemented a compiler targeting WebAssembly for it – **Weaver** – including my type inference framework (Warp). Lastly, I gave a novel array calculus design – **Star** (Chapter 5). It uniquely features array shapes and indices that exploit structural subtyping, providing safety while admitting type inference with Warp. It thus serves as a new middle-ground between the conventional untyped discipline (flexible – but unsafe), and complicated (but safe) dependently-typed systems.

Publications I gave a talk about my thesis at BCTCS 2025. A research paper on Star co-authored with my supervisors was accepted for publication in the ARRAY Workshop Proceedings.

Further work I identify the following directions for further work as most interesting:

- Fully proving the correctness of constraint-based algebraic subtyping presentation – particularly in presence of homomorphism applications.
- Determining the scope of type system features expressible using homomorphisms, besides extensible records (FL) and type isomorphisms (Star).
- With Warp developed, it would be exciting to use it to for typing real-world dynamic languages, like Python or Lua (as done with set-theoretic types [14, 15, 62]).
- Implementing Star in practice (e.g. extending Futhark [32], embedding in Python [3]), which leads to interesting problems: performant code generation, shape inference, and recursive indices.

Despite the age of this natural idea, I have shown structural subtyping should be revisited as an effective way for designing practical type systems. Likewise, it still leads to many interesting theoretical questions.

Bibliography

1. M. Abadi and L. Cardelli. 'An Imperative Object Calculus'. In: *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*. Ed. by P. D. Mosses, M. Nielsen, and M. I. Schwartzbach. Vol. 915. Lecture Notes in Computer Science. Springer, 1995, pp. 471–485. DOI: 10.1007/3-540-59293-8_214. URL: https://doi.org/10.1007/3-540-59293-8%5C_214.
2. M. G. Abbott, T. Altenkirch, N. Ghani, and C. McBride. 'Derivatives of Containers'. In: *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings*. Ed. by M. Hofmann. Vol. 2701. Lecture Notes in Computer Science. Springer, 2003, pp. 16–30. DOI: 10.1007/3-540-44904-3_2. URL: https://doi.org/10.1007/3-540-44904-3%5C_2.
3. J. Bachurski and A. Mycroft. 'Points for Free: Embedding Pointful Array Programming in Python'. In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY 2024, Copenhagen, Denmark, 25 June 2024*. Ed. by A. Hsu and A. Sinkarovs. ACM, 2024, pp. 1–12. DOI: 10.1145/3652586.3663312. URL: https://doi.org/10.1145/3652586.3663312.
4. J. Bachurski, A. Mycroft, and D. Orchard. 'Structuring Arrays with Algebraic Shapes'. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY 2025, Copenhagen, Denmark, 17 June 2025*. Ed. by M. Elsman and A. Sinkarovs. ACM, 2025, pp. 1–16. DOI: 10.1145/3736112.3736141. URL: https://doi.org/10.1145/3736112.3736141.
5. L. Bailly, T. Henriksen, and M. Elsman. 'Shape-Constrained Array Programming with Size-Dependent Types'. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, FHPNC 2023, Seattle, WA, USA, 4 September 2023*. Ed. by G. Keller and S. Westrick. ACM, 2023, pp. 29–41. DOI: 10.1145/3609024.3609412. URL: https://doi.org/10.1145/3609024.3609412.
6. J. Bernardy, M. Boespflug, R. R. Newton, S. P. Jones, and A. Spiwack. 'Linear Haskell: practical linearity in a higher-order polymorphic language'. *Proc. ACM Program. Lang.* 2:POPL, 2018, 5:1–5:29. DOI: 10.1145/3158093. URL: https://doi.org/10.1145/3158093.
7. D. Binder, I. Skupin, D. Läwen, and K. Ostermann. 'Structural refinement types'. In: *TyDe '22: 7th ACM SIGPLAN International Workshop on Type-Driven Development, Ljubljana, Slovenia, 11 September 2022*. ACM, 2022, pp. 15–27. DOI: 10.1145/3546196.3550163. URL: https://doi.org/10.1145/3546196.3550163.
8. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. 'Making the Future Safe for the Past: Adding Genericity to the Java Programming Language'. In: *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*. Ed. by B. N. Freeman-Benson and C. Chambers. ACM, 1998, pp. 183–200. DOI: 10.1145/286936.286957. URL: https://doi.org/10.1145/286936.286957.

9. L. Cardelli. 'A Semantics of Multiple Inheritance'. In: *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*. Ed. by G. Kahn, D. B. MacQueen, and G. D. Plotkin. Vol. 173. Lecture Notes in Computer Science. Springer, 1984, pp. 51–67. DOI: 10.1007/3-540-13346-1_2. URL: https://doi.org/10.1007/3-540-13346-1%5C_2.
10. L. Cardelli. 'Extensible records in a pure calculus of subtyping'. In: *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, Cambridge, MA, USA, 1994, pp. 373–425. ISBN: 026207155X.
11. L. Cardelli. 'Structural Subtyping and the Notion of Power Type'. In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. Ed. by J. Ferrante and P. Mager. ACM Press, 1988, pp. 70–79. DOI: 10.1145/73560.73566. URL: https://doi.org/10.1145/73560.73566.
12. L. Cardelli and J. C. Mitchell. 'Operations on Records'. *Math. Struct. Comput. Sci.* 1:1, 1991, pp. 3–48. DOI: 10.1017/S0960129500000049. URL: https://doi.org/10.1017/S0960129500000049.
13. L. Cardelli and P. Wegner. 'On Understanding Types, Data Abstraction, and Polymorphism'. *ACM Comput. Surv.* 17:4, 1985, pp. 471–522. DOI: 10.1145/6041.6042. URL: https://doi.org/10.1145/6041.6042.
14. M. Cassola, A. Talagorria, A. Pardo, and M. Viera. 'A gradual type system for Elixir'. *J. Comput. Lang.* 68, 2022, p. 101077. DOI: 10.1016/J.COLA.2021.101077. URL: https://doi.org/10.1016/j.cola.2021.101077.
15. G. Castagna, M. Laurent, and K. Nguyen. 'Polymorphic Type Inference for Dynamic Languages'. *Proc. ACM Program. Lang.* 8:POPL, 2024, pp. 1179–1210. DOI: 10.1145/3632882. URL: https://doi.org/10.1145/3632882.
16. G. Castagna, T. Petrucciani, and K. Nguyen. 'Set-theoretic types for polymorphic variants'. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by J. Garrigue, G. Keller, and E. Sumii. ACM, 2016, pp. 378–391. DOI: 10.1145/2951913.2951928. URL: https://doi.org/10.1145/2951913.2951928.
17. L. Cheng and L. Parreaux. 'The Ultimate Conditional Syntax'. *Proc. ACM Program. Lang.* 8:OOPSLA2, 2024, pp. 988–1017. DOI: 10.1145/3689746. URL: https://doi.org/10.1145/3689746.
18. D. Chiang, A. M. Rush, and B. Barak. 'Named Tensor Notation'. *Trans. Mach. Learn. Res.* 2023, 2023. URL: https://openreview.net/forum?id=hVT7SHlilx.
19. W. R. Cook, W. L. Hill, and P. S. Canning. 'Inheritance Is Not Subtyping'. In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. Ed. by F. E. Allen. ACM Press, 1990, pp. 125–135. DOI: 10.1145/96709.96721. URL: https://doi.org/10.1145/96709.96721.
20. S. Dolan. 'Algebraic subtyping'. PhD thesis. University of Cambridge, 2017.
21. S. Dolan and A. Mycroft. 'Polymorphism, subtyping, and type inference in MLsub'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM, 2017, pp. 60–72. DOI: 10.1145/3009837.3009882. URL: https://doi.org/10.1145/3009837.3009882.

22. J. Eifrig, S. F. Smith, and V. Trifonov. 'Type inference for recursively constrained types and its application to OOP'. In: *Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995, Tulane University, New Orleans, LA, USA, March 29 - April 1, 1995*. Ed. by S. D. Brookes, M. G. Main, A. Melton, and M. W. Mislove. Vol. 1. Electronic Notes in Theoretical Computer Science. Elsevier, 1995, pp. 132–153. DOI: 10.1016/S1571-0661(04)80008-2. URL: https://doi.org/10.1016/S1571-0661(04)80008-2.

23. M. Elsman. 'Double-Ended Bit-Stealing for Algebraic Data Types'. *Proc. ACM Program. Lang.* 8:ICFP, 2024, pp. 88–120. DOI: 10.1145/3674628. URL: https://doi.org/10.1145/3674628.

24. F. Emrich, J. Stolarek, J. Cheney, and S. Lindley. 'Constraint-based type inference for FreezeML'. *Proc. ACM Program. Lang.* 6:ICFP, 2022, pp. 570–595. DOI: 10.1145/3547642. URL: https://doi.org/10.1145/3547642.

25. J. Garrigue. 'Programming with polymorphic variants'. In: *ML workshop*. Vol. 13. 7. Baltimore. 1998.

26. J. 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.

27. J. Gibbons. 'APLicative Programming with Naperian Functors'. In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by H. Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 556–583. DOI: 10.1007/978-3-662-54434-1_21. URL: https://doi.org/10.1007/978-3-662-54434-1%5C_21.

28. R. Grigore. 'Java generics are turing complete'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM, 2017, pp. 73–85. DOI: 10.1145/3009837.3009871. URL: https://doi.org/10.1145/3009837.3009871.

29. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. 'Bringing the web up to speed with WebAssembly'. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by A. Cohen and M. T. Vechev. ACM, 2017, pp. 185–200. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363.

30. C. R. Harris, K. J. Millman, S. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. 'Array programming with NumPy'. *Nat.* 585, 2020, pp. 357–362. DOI: 10.1038/S41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

31. T. Henriksen and M. Elsman. 'Towards size-dependent types for array programming'. In: *ARRAY 2021: Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, Virtual Event, Canada, 21 June, 2021*. Ed. by T. M. Low and J. Gibbons. ACM, 2021, pp. 1–14. DOI: 10.1145/3460944.3464310. URL: https://doi.org/10.1145/3460944.3464310.

32. T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. 'Futhark: purely functional GPU-programming with nested parallelism and in-place array updates'. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by A. Cohen and M. T. Vechev. ACM, 2017, pp. 556–571. DOI: 10.1145/3062341.3062354. URL: https://doi.org/10.1145/3062341.3062354.

33. S. Hoyer and J. Hamman. 'xarray: ND labeled arrays and datasets in Python'. *Journal of Open Research Software* 5:1, 2017, pp. 10–10.

34. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. *Lua 5.4 Reference Manual*. Lua.org, PUC-Rio, https://www.lua.org/manual/5.4/manual.html, 2020.

35. A. Igarashi, B. C. Pierce, and P. Wadler. 'Featherweight Java: a minimal core calculus for Java and GJ'. *ACM Trans. Program. Lang. Syst.* 23:3, 2001, pp. 396–450. DOI: 10.1145/503502.503505. URL: https://doi.org/10.1145/503502.503505.

36. K. E. Iverson. 'A programming language'. In: *Proceedings of the 1962 spring joint computer conference, AFIPS 1962 (Spring), San Francisco, California, USA, May 1-3, 1962*. Ed. by G. A. B. III. ACM, 1962, pp. 345–351. DOI: 10.1145/1460833.1460872. URL: https://doi.org/10.1145/1460833.1460872.

37. O. Kiselyov. 'Generating C: Heterogeneous metaprogramming system description'. *Sci. Comput. Program.* 231, 2024, p. 103015. DOI: 10.1016/J.SCICO.2023.103015. URL: https://doi.org/10.1016/j.scico.2023.103015.

38. A. Lorenzen, L. White, S. Dolan, R. A. Eisenberg, and S. Lindley. 'Oxidizing OCaml with Modal Memory Management'. *Proc. ACM Program. Lang.* 8:ICFP, 2024, pp. 485–514. DOI: 10.1145/3674642. URL: https://doi.org/10.1145/3674642.

39. D. Malayeri and J. Aldrich. 'Integrating Nominal and Structural Subtyping'. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Ed. by J. Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 260–284. DOI: 10.1007/978-3-540-70592-5_12. URL: https://doi.org/10.1007/978-3-540-70592-5%5C_12.

40. R. Marques, M. Florido, and P. B. Vasconcelos. 'Towards Algebraic Subtyping for Extensible Records'. *CoRR* abs/2407.06747, 2024. DOI: 10.48550/ARXIV.2407.06747. arXiv: 2407.06747. URL: https://doi.org/10.48550/arXiv.2407.06747.

41. N. Milojkovic, M. Ghafari, and O. Nierstrasz. 'It's duck (typing) season!' In: *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*. Ed. by G. Scanniello, D. Lo, and A. Serebrenik. IEEE Computer Society, 2017, pp. 312–315. DOI: 10.1109/ICPC.2017.10. URL: https://doi.org/10.1109/ICPC.2017.10.

42. J. G. Morris and J. McKinna. 'Abstracting extensible data types: or, rows by any other name'. *Proc. ACM Program. Lang.* 3:POPL, 2019, 12:1–12:28. DOI: 10.1145/3290325. URL: https://doi.org/10.1145/3290325.

43. M. Odersky, M. Sulzmann, and M. Wehr. 'Type Inference with Constrained Types'. *Theory Pract. Object Syst.* 5:1, 1999, pp. 35–55.

44. L. Parreaux. 'The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)'. *Proc. ACM Program. Lang.* 4:ICFP, 2020, 124:1–124:28. DOI: 10.1145/3409006. URL: https://doi.org/10.1145/3409006.

45. L. Parreaux and C. Y. Chau. 'MLstruct: principal type inference in a Boolean algebra of structural types'. *Proc. ACM Program. Lang.* 6:OOPSLA2, 2022, pp. 449–478. DOI: 10.1145/3563304. URL: https://doi.org/10.1145/3563304.

46. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett. 2019, pp. 8024–8035. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

47. A. Paszke, D. D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. J. Johnson, J. Ragan-Kelley, and D. Maclaurin. 'Getting to the Point: index sets and parallelism-preserving autodiff for pointful array programming'. *Proc. ACM Program. Lang.* 5:ICFP, 2021, pp. 1–29. DOI: 10.1145/3473593. URL: https://doi.org/10.1145/3473593.

48. T. Petrucciani. 'Polymorphic set-theoretic types for functional languages. (Types ensemblistes polymorphes pour les langages fonctionnels)'. PhD thesis. Sorbonne Paris Cité, France, 2019. DOI: 10.15167/PETRUCCIANI-TOMMASO_PHD2019-03-14. URL: https://tel.archives-ouvertes.fr/tel-02119930.

49. B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN: 0262162288.

50. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

51. B. C. Pierce. 'Programming with intersection types and bounded polymorphism'. PhD thesis. Carnegie Mellon University, 1992.

52. F. Pottier. 'A Framework for Type Inference with Subtyping'. In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. Ed. by M. Felleisen, P. Hudak, and C. Queinnec. ACM, 1998, pp. 228–238. DOI: 10.1145/289423.289448. URL: https://doi.org/10.1145/289423.289448.

53. F. Pottier. *Type Inference in the Presence of Subtyping: from Theory to Practice*. Research Report RR-3483. INRIA, 1998. URL: https://inria.hal.science/inria-00073205.

54. F. Pottier and D. Rémy. 'The Essence of ML Type Inference'. In: *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN: 9780262281591. DOI: 10.7551/mitpress/1104.003.0016. eprint: https://direct.mit.edu/book/chapter-pdf/2389924/9780262281591_caj.pdf. URL: https://doi.org/10.7551/mitpress/1104.003.0016.

55. D. Rémy. 'Efficient representation of extensible records'. In: *Proceedings of the ACM SIGPLAN Workshop on ML and its applications*. 1994, pp. 12–16.

56. D. Rémy. 'Type inference for records in natural extension of ML'. In: *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, Cambridge, MA, USA, 1994, pp. 67–95. ISBN: 026207155X.

57. D. 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.

58. D. Rémy and J. Vouillon. 'Objective ML: An Effective Object-Oriented Extension to ML'. *Theory Pract. Object Syst.* 4:1, 1998, pp. 27–50.

59. J. C. Reynolds. 'Design of the programming language Forsythe'. In: *ALGOL-like languages*. Springer, 1997, pp. 173–233.

60. J. C. Reynolds. 'The Meaning of Types From Intrinsic to Extrinsic Semantics'. *BRICS Report Series* 7:32, 2000. DOI: 10.7146/brics.v7i32.20167. URL: https://tidsskrift.dk/brics/article/view/20167.

61. R. Schenck, N. H. Hinnerskov, T. Henriksen, M. Madsen, and M. Elsman. 'AUTOMAP: Inferring Rank-Polymorphic Function Applications with Integer Linear Programming'. *Proc. ACM Program. Lang.* 8:OOPSLA2, 2024, pp. 1787–1813. DOI: 10.1145/3689774. URL: https://doi.org/10.1145/3689774.

62. A. Schimpf, S. Wehr, and A. Bieniusa. 'Set-theoretic Types for Erlang'. In: *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages, IFL 2022, Copenhagen, Denmark, 31 August 2022- 2 September 2022*. ACM, 2022, 4:1–4:14. DOI: 10.1145/3587216.3587220. URL: https://doi.org/10.1145/3587216.3587220.

63. S. Scholz and A. Sinkarovs. 'Tensor comprehensions in SaC'. In: *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*. Ed. by J. Stutterheim and W. Chin. ACM, 2019, 15:1–15:13. DOI: 10.1145/3412932.3412947. URL: https://doi.org/10.1145/3412932.3412947.

64. A. Shaikhha, A. W. Fitzgibbon, D. Vytiniotis, and S. P. Jones. 'Efficient differentiable programming in a functional array-processing language'. *Proc. ACM Program. Lang.* 3:ICFP, 2019, 97:1–97:30. DOI: 10.1145/3341701. URL: https://doi.org/10.1145/3341701.

65. J. Siek and W. Taha. 'Gradual Typing for Functional Languages'. In: *Proceedings of the Scheme and Functional Programming Workshop*. 2006, pp. 81–92.

66. J. G. Siek and W. Taha. 'Gradual Typing for Objects'. In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Ed. by E. Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 2–27. DOI: 10.1007/978-3-540-73589-2_2. URL: https://doi.org/10.1007/978-3-540-73589-2%5C_2.

67. J. Slepak, O. Shivers, and P. Manolios. 'An Array-Oriented Language with Static Rank Polymorphism'. In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Z. Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 27–46. DOI: 10.1007/978-3-642-54833-8_3. URL: https://doi.org/10.1007/978-3-642-54833-8%5C_3.

68. Stack Exchange Inc. *Stack Overflow Developer Survey*. 2024. URL: https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages (visited on 24/05/2025).

69. Z. Su, A. Aiken, J. Niehren, T. Priesnitz, and R. Treinen. 'The first-order theory of subtyping constraints'. In: *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. Ed. by J. Launchbury and J. C. Mitchell. ACM, 2002, pp. 203–216. DOI: 10.1145/503272.503292. URL: https://doi.org/10.1145/503272.503292.

70. W. Tang, D. Hillerström, J. McKinna, M. Steuwer, O. Dardha, R. Fu, and S. Lindley. 'Structural Subtyping as Parametric Polymorphism'. *Proc. ACM Program. Lang.* 7:OOPSLA2, 2023, pp. 1093–1121. DOI: 10.1145/3622836. URL: https://doi.org/10.1145/3622836.

71. D. Vytiniotis, S. L. P. Jones, T. Schrijvers, and M. Sulzmann. 'OutsideIn(X) Modular type inference with local assumptions'. *J. Funct. Program.* 21:4-5, 2011, pp. 333–412. DOI: 10.1017/S0956796811000098. URL: https://doi.org/10.1017/S0956796811000098.

72. P. Wadler. 'Theorems for Free!' In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. Ed. by J. E. Stoy. ACM, 1989, pp. 347–359. DOI: 10.1145/99370.99404. URL: https://doi.org/10.1145/99370.99404.

73. M. Wand. 'Complete Type Inference for Simple Objects'. In: *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 1987, pp. 37–44. URL: http://www.ccs.neu.edu/home/wand/papers/wand-lics-87.pdf.

74. WebAssembly contributors. *Binaryen*. Version 123. URL: https://github.com/WebAssembly/binaryen.

75. WebAssembly contributors. *GC Proposal for WebAssembly*. 2024. URL: https://github.com/WebAssembly/gc.

76. L. White, F. Bour, and J. Yallop. 'Modular implicits'. In: *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*. Ed. by O. Kiselyov and J. Garrigue. Vol. 198. EPTCS. 2014, pp. 22–63. DOI: 10.4204/EPTCS.198.2. URL: https://doi.org/10.4204/EPTCS.198.2.

77. N. Xie, B. C. d. S. Oliveira, X. Bi, and T. Schrijvers. 'Row and Bounded Polymorphism via Disjoint Polymorphism'. In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*. Ed. by R. Hirschfeld and T. Pape. Vol. 166. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 27:1–27:30. DOI: 10.4230/LIPICS.ECOOP.2020.27. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2020.27.

78. J. Yallop, D. Sheets, and A. Madhavapeddy. 'A modular foreign function interface'. *Sci. Comput. Program.* 164, 2018, pp. 82–97. DOI: 10.1016/J.SCICO.2017.04.002. URL: https://doi.org/10.1016/j.scico.2017.04.002.

Notation

Common symbols

x, y, z Variable

α, β, γ Type variable

e Expression

τ, π Type

ℓ Record label

T Variant tag

Γ Typing environment

$[e/x]e$ Substitution

$[\tau/\alpha]\tau$ Type substitution, with $\psi ::= \cdot \mid \psi, \tau/\alpha$

Common syntax

$\triangleq$ Definition

$\overline{t}$ Repetition of t

$\top$ Top type – supertype of all types

$\bot$ Bottom type – subtype of all types

$\{\cdots\}$ Record

$[\cdots]$ Variant

$[\![-]\!]$ Expression translation: semantics- and type-preserving

$⟦-⟧$ Expression translation: semantics-preserving

$(\!|-|\!)$ Type translation

$\Gamma \vdash e : \tau$ Typing judgement

Featherweight Java

`e` Expression *(written in monospace font)*

`C` Class

`m` Method name

`f` Field, variable name

K Class constructor definition

M Method definition

L Class definition

Featherweight Lua

ϕ Field type

$\{\overline{\ell = e} \mid e\}$ Record extension

$\{\overline{\ell : \phi} \mid \phi\}$ Record type

$\mathbb{T}$ Top field type

$\mathbb{\perp}$ Bottom field type

$\boxed{\tau}$ Present field type

$\square$ Absent field type

$e.\ell$ Let-binding

$e \rhd \tau$ Runtime type coercion

`e` Lua expression *(written in sans-serif font)*

Type inference

σ Type scheme

$\sigma \models \tau$ Type scheme instantiation

$\mu\,\alpha.\,\tau$ Recursive type, where variable α refers to entire type when occuring in τ

Subtyping and the type lattice

$\equiv$ Type equivalence under the Boolean algebra

$\leqslant$ Subtype

$\geqslant$ Supertype

$K[\overline{\tau}]$ Type constructor K with type subterms $\overline{\tau}$

$\lhd$ Decomposition of type constructor subtyping constraint

$\sqcap$ Meet in the lattice of type constructors

$\sqcup$ Join in the lattice of types constructors

$\wedge$ Meet in the lattice of types

$\vee$ Join in the lattice of types

$\neg$ Complement in the Boolean algebra of types

$\leqslant^{\forall}$ Subsumption of type schemes

θ Homomorphism in the type algebra

id Identity homomorphism

$\overleftarrow{\theta}$ Left adjoint of a type homomorphism

$\overrightarrow{\theta}$ Right adjoint of a type homomorphism

$\underline{\overleftarrow{\theta}}$ Remainder of a left adjoint of a type homomorphism

$\underline{\overrightarrow{\theta}}$ Remainder of a right adjoint of a type homomorphism

Constraint-based algebraic subtyping

c Constraint

$\langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle$ Constraint generation for $\Gamma \vdash e : \tau$

$E\langle\diamond\rangle$ Type equivalence context

$C\langle\diamond\rangle$ Constraint solving context

$\&$ Conjunction of constraints

$\mathbf{T}$ Always-true constraint

$\mathbf{F}$ Always-false constraint

$\cong$ Constraint equivalence under the Boolean algebra

$\rightsquigarrow$ Solver step

$\mathcal{S}(c)$ Constraint in solved form

Fabric

p Pattern

$e \setminus \ell$ Record restriction – removing a field ℓ from e

$\ell_? e$ Checked projection – a tagged value $\mathrm{Some}\, e.\ell$ if ℓ is in e, $\mathrm{None}\ ()$ otherwise

$\boxed{\tau}?$ Optional field – absent, or present with τ

κ Case type

$T\, e$ Tagging

$!e$ Untagging

t Nominal type

$e : \tau \blacktriangleright \tau$ Abbreviation-aware type cast

Star

e^S Shape expression

s Shape value

σ Shape type

float Floating point type

int Integer type

f Float

n, m Integer

ι Index for shape: upper bound on index type given shape type

$[\sigma_1..\sigma_2]\tau$ Array type

$[\sigma]\tau$ Abbreviated array type ($[\sigma..\sigma]\tau$)

J Unevaluated array elements

I Evaluated array elements

$\mathsf{Arr}(s, I)$ Array literal value

$\mathsf{Arr}(s, J)$ Array literal expression

$\Phi\, x[e].\, e'$ Array comprehension – array of shape e with element e' at index x

$|e|$ Array shape access

$\#e$ Sized shape

$\{\!|\cdots|\!\}$ Product shape

$[\![\cdots]\!]$ Concatenation shape

$e_\circ \ell$ Product dimension projection

$e_\diamond T$ Concatenation component projection

$e \sqcap e$ Broadcasting (expression)

$s \bowtie s$ Broadcasting (semantics)

$v \oslash s$ Index coercion into shape

$v \lhd s$ Index in-bounds of shape

$v \trianglelefteq s$ Index structurally-exactly in-bounds of shape

$v \blacktriangleleft s$ Index structurally-in-bounds of shape

$C\langle\diamond\rangle$ Evaluation context

$e \rightsquigarrow e$ Small-step (operational semantics)

$e \rightsquigarrow \lightning$ Raises-error step (operational semantics)

A Semantics for Featherweight Lua

The following is an operational semantics in the same style as for Star (Section 5.2.5).

$$\boxed{e \rightsquigarrow e}$$

$$\begin{array}{rll}
\text{let } x = v \text{ in } e \rightsquigarrow [v/x]e & & \textsc{Let} \\
(\lambda x.\, e)\, v \rightsquigarrow [v/x]e & & \textsc{App} \\
\left\{\ell = e \,\middle|\, \left\{\overline{\ell = v_\ell}\right\}\right\} \rightsquigarrow \left\{\overline{\ell = v_\ell}, \ell = e\right\} & \text{(if } \ell \text{ not in } \overline{\ell}\text{)} & \textsc{Ext} \\
\left\{\overline{\ell = v_\ell}\right\}.\ell \rightsquigarrow v_\ell & & \textsc{Proj} \\
e \rhd \tau \rightsquigarrow e & & \textsc{Cast}
\end{array}$$

$$\textsc{Cong}\ \frac{e \rightsquigarrow e'}{C\langle e\rangle \rightsquigarrow C\langle e'\rangle}$$

Where we define evaluation contexts $C\langle\diamond\rangle$ for the congruence rule:

$$\begin{aligned}
C\langle\diamond\rangle ::= {} & \diamond \\
& \mid \text{let } x = C\langle\diamond\rangle \text{ in } e \\
& \mid C\langle\diamond\rangle\, e \mid v\, C\langle\diamond\rangle \\
& \mid \left\{\overline{\ell = v}, \ell = C\langle\diamond\rangle, \overline{\ell = e}\right\} \\
& \mid \{\ell = e \mid C\langle\diamond\rangle\} \mid \{\ell = C\langle\diamond\rangle \mid v\} \\
& \mid C\langle\diamond\rangle.\ell
\end{aligned}$$

B Correctness of the FJ-FL Translation

B.1 Correctness theorems

Proving completeness (stated as injectivity) is straightforward, so we focus on soundness – preservation of types and semantics.

B.1.1 Preservation of typing

For brevity (as the proofs are similarly straightforward), we assume *soundness of the class table translation*, stating that definitions introduced by $[\![\mathtt{K}]\!]/[\![\mathtt{M}]\!]/[\![\mathtt{L}]\!]$ have the claimed types $(\!|\mathtt{K}|\!)/(\!|\mathtt{M}|\!)/(\!|\mathtt{L}|\!)$, which is straightforward to see.

Proof. We are required to prove

$$\Gamma \vdash \mathtt{e} : \mathtt{C} \implies (\!|\Gamma|\!) \vdash [\![\mathtt{e}]\!] : (\!|\mathtt{C}|\!)$$

We proceed by structural induction on FJ typing $\Gamma \vdash \mathtt{e} : \mathtt{C}$ and consider possible cases.

Variable `x` Obvious.

Attribute `e.f` By rule inversion, $\Gamma \vdash \mathtt{e} : \mathtt{C}$ for some `C` whose instances have the field `f`. But then since $(\!|\Gamma|\!) \vdash [\![\mathtt{e}]\!] : (\!|\mathtt{C}|\!)$ (inductive hypothesis) and thus $(\!|\Gamma|\!) \vdash [\![\mathtt{e}]\!] : \{f : (\!|\mathtt{C}'|\!)\}$ (subsumption; soundness of translating `L`), and thus $(\!|\Gamma|\!) \vdash [\![\mathtt{e}]\!].\mathtt{f} : (\!|\mathtt{C}'|\!)$.

Method `e.m`$(\overline{\mathtt{e}})$ By rule inversion, we get that $\Gamma \vdash \mathtt{e} : \mathtt{C}$ for some `C` which has the method `m` defined, and the typing of arguments $\overline{\Gamma \vdash \mathtt{e} : \mathtt{C}}$ matches `m`. We then finish immediately, as as above `m` is among the fields of the record $[\![\mathtt{e}]\!]$, and all argument types match in the application $[\![\mathtt{e.m(\overline{e})}]\!] = \mathrm{let}\, x = [\![\mathtt{e}]\!]\, \mathrm{in}\, x.\mathtt{m}\, x\, \overline{[\![\mathtt{e}]\!]}$ (by soundness of translating `M` and inductive hypothesis).

Constructor `new C`$(\overline{\mathtt{e}})$ We see that $\overline{e}$ are well-typed, i.e. $\overline{\Gamma \vdash e : \mathtt{C}}$. By the inductive hypothesis, we then have $\overline{(\!|\Gamma|\!) \vdash [\![e]\!] : (\!|\mathtt{C}|\!)}$. Since $(\!|\Gamma|\!) \vdash \mathtt{C} : \overline{(\!|\mathtt{C}|\!)} \to (\!|\mathtt{C}|\!)$ (where `C` is the translated constructor) by soundness of the `CT` translation at constructors `K`, we immediately see that all argument types match in the repeated application $[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!] = \mathtt{C}\, \overline{[\![\mathtt{e}]\!]}$, finishing the inductive step.

Cast `(C)e` Immediate, as FL does not check casts. Nominal-style runtime type checking would require adding an extra type-tag field to each record for representing objects [50]. Including such tags would extend soundness with respect to semantics, so that if `e` gets stuck then does $[\![\mathtt{e}]\!]$ too.

□

B.1.2 Preservation of semantics

We recall that in FJ, values are expressions in normal form, which are composed of only `new` (with 'leaves' given by nullary constructors). Furthermore, we state a canonical form lemmas for FL and a sketch of the canonical form lemma for translated FJ values.

Lemma B.1 (Canonical forms in FL)**.** *Records and functions have the following canonical forms:*

- *If* $\cdot \vdash v : \tau \to \tau'$*, then* $v = \lambda x.\, e$ *so that* $x : \tau \vdash e : \tau'$*.*
- *If* $\cdot \vdash v : \{\overline{\ell : \tau}\}$*, then* $v = \{\overline{\ell = v}, \ldots\}$*.*

Lemma B.2 (Soundness: canonical form of translated FJ values)**.** *If* $\cdot \vdash \mathtt{v} : \mathtt{C}$*, then* $\cdot \vdash [\![\mathtt{v}]\!]_{CT} : (\!|\mathtt{C}|\!)$*, so that* $[\![\mathtt{v}]\!]$ *is a record containing all attributes* `f` *(defined in* $(\!|\mathtt{C}|\!)$*) and methods* `m` *(which match definitions in* $\mathtt{C}_{\text{proto}}$*).*

We now prove the theorem.

Proof. We prove that

$$\cdot \vdash \mathtt{e} : \mathtt{C} \implies \mathtt{e} \rightsquigarrow^* \mathtt{v} \implies [\![\mathtt{e}]\!] \rightsquigarrow^* [\![\mathtt{v}]\!]$$

and, as before, proceed by structural induction on $\Gamma \vdash \mathtt{e} : \mathtt{C}$.

Variable `x` Vacuously, as encountering a variable is contradictory with $\cdot \vdash \mathtt{e} : \mathtt{C}$.

Attribute `e.f` By inductive hypothesis, $[\![\mathtt{e}]\!] \rightsquigarrow^* [\![\mathtt{v}]\!]$. But then $[\![\mathtt{v}]\!]$ must be a record with a field `f` (canonical form for an object with attribute `f`). Since a record's field is its substructure up to congruence, we immediately get $[\![\mathtt{e.f}]\!] \rightsquigarrow^* [\![\mathtt{v.f}]\!]$, which is sufficient as $\mathtt{e} \rightsquigarrow^* \mathtt{v} \implies \mathtt{e.v} \rightsquigarrow^* \mathtt{v.f}$.

Method `e.m`($\overline{\mathtt{e}}$) Accessing the field `m` proceeds analogically, so we focus on application. By looking at the definition of $[\![\mathtt{M}]\!]$ for the accessed method `m`, it is clear to see that in instantiation `this` and all arguments `x` all substituted into the method's body as in FJ (specifically, we need a class table translation soundness for semantics as we did for typing). Here, it is key that the object itself is passed as `this` and its attributes and methods are accessible therein. With that said, clearly if $\mathtt{e.m}(\overline{\mathtt{e}}) \rightsquigarrow^* \mathtt{e}'$ – where $\mathtt{e}'$ is the body of `m` with arguments substituted – then likewise $[\![\mathtt{e.m}(\overline{\mathtt{e}})]\!] \rightsquigarrow^* [\![\mathtt{e}']\!]$, and we proceed by induction from $\mathtt{e}'$ (which is well-founded, as we only consider terminating programs).

Constructor `new C`($\overline{\mathtt{e}}$) Since constructors `new` with all arguments evaluated serve as canonical forms in FJ, the inductive hypothesis tells us that all these arguments evaluate as required ($\overline{[\![\mathtt{e}]\!] \rightsquigarrow^* [\![\mathtt{v}]\!]}$), and thus we can apply the translated constructor `C` to obtain a record matching the canonical form of the translated value $[\![\mathtt{v}]\!]$ for $\mathtt{new\ C}(\overline{\mathtt{e}}) \rightsquigarrow^* \mathtt{v}$.

Cast `(C)e` Immediate – but see note about casts in the preceding proof.

□

C Type inference for Featherweight Lua

This appendix gives technical details necessary to instantiate Warp to perform type inference for Featherweight Lua programs.

C.1 Lattice of types

For simplicity, I denote instantiated type constructors $K[\overline{\tau}]$ with τ. The lattice of type constructors in FL is defined as such:

$$\begin{aligned}
\tau \sqcup \top &= \top \\
\tau \sqcup \bot &= \tau \\
(\tau_1 \to \tau_1') \sqcup (\tau_2 \to \tau_2') &= (\tau_1 \sqcap \tau_2) \to (\tau_1' \sqcup \tau_2') \\
\left\{\overline{\ell : \phi_\ell} \,\middle|\, \phi\right\} \sqcup \left\{\overline{\ell' : \phi'_\ell} \,\middle|\, \phi'\right\} &= \left\{\overline{\ell : \phi_\ell \sqcup \phi'_\ell} \,\middle|\, \phi \sqcup \phi'\right\}
\end{aligned}$$

with $\wedge$ defined dually (by swapping $\sqcup$ with $\sqcap$, and $\top$ with $\bot$). It can be straightforwardly proven this definition yields a distributive lattice.

C.2 Constraint generation

I present the full constraint generation for FL.

$$\boxed{\langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle = c}$$

$$\begin{aligned}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle &= \Gamma(x) \leqslant \tau && \textsc{Var} \\
\langle\!\langle \Gamma \vdash \text{let } x = e' \text{ in } e : \tau \rangle\!\rangle &= \exists \tau'.\, \langle\!\langle \Gamma \vdash e' : \tau' \rangle\!\rangle \;\&\; \langle\!\langle \Gamma, e' : \tau' \vdash e : \tau \rangle\!\rangle && \textsc{Let} \\
\langle\!\langle \Gamma \vdash \lambda x.\, e : \tau \rangle\!\rangle &= \exists \tau', \tau''.\, \langle\!\langle \Gamma, x : \tau' \vdash e : \tau'' \rangle\!\rangle \;\&\; \tau \leqslant \tau' \to \tau'' && \textsc{Lam} \\
\langle\!\langle \Gamma \vdash e\, e' : \tau \rangle\!\rangle &= \exists \tau'.\, \langle\!\langle \Gamma \vdash e' : \tau' \rangle\!\rangle \;\&\; \langle\!\langle \Gamma \vdash e : \tau' \to \tau \rangle\!\rangle && \textsc{App} \\
\left\langle\!\left\langle \Gamma \vdash \left\{\overline{\ell = e_\ell}\right\} : \tau \right\rangle\!\right\rangle &= \exists \overline{\tau_\ell}.\, \overline{\langle\!\langle \Gamma \vdash e_\ell : \tau_\ell \rangle\!\rangle} \;\&\; \left\{\overline{\ell : \tau_\ell} \,\middle|\, \mathbb{T}\right\} \leqslant \tau && \textsc{Cons} \\
\langle\!\langle \Gamma \vdash e.\ell : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e : \{\ell : \tau\} \rangle\!\rangle && \textsc{Proj} \\
\langle\!\langle \Gamma \vdash \{\ell = e_\ell \,|\, e\} : \tau' \rangle\!\rangle &= \exists \tau, \tau_\ell.\, \langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle \;\&\; \langle\!\langle \Gamma \vdash e_\ell : \tau_\ell \rangle\!\rangle \;\& && \textsc{Ext} \\
&\qquad \& \; \tau \leqslant \{\ell : \square \,|\, \mathbb{T}\} \;\&\; \text{forget}_\ell(\tau) \wedge \{\ell : \tau_\ell \,|\, \mathbb{T}\} \leqslant \tau'
\end{aligned}$$

It also types record extension (Ext) by applying the forget homomorphism (considered throughout examples in Section 3.4) – but in doing so it breaks Dolan's polarity restriction (Section 3.1.2) and involves a homomorphism application (Section 3.4).

C.3 Correctness of homomorphisms

I now make precise the technical development of homomorphisms for typing extensible records, as I presented it in Section 3.4.

C.3.1 Language

We now make precise the language of homomorphisms θ for FL. A key aspect is that not only need morphisms forget_ℓ (denoted here as $[\ell \mapsto \mathbb{T}]$) and free_ℓ ($[\ell \mapsto \perp\!\!\!\perp]$), but also their compositions. We define θ as such:

$$\theta ::= \mathrm{id} \mid \theta \circ [\ell \mapsto \mathbb{T}] \mid \theta \circ [\ell \mapsto \perp\!\!\!\perp]$$

and, as required, define application on type constructors:

$$\begin{aligned}
\theta(\top) &= \top \\
\theta(\bot) &= \bot \\
\theta(\tau \to \tau') &= \tau \to \tau' \\
\mathrm{id}\big(\{\overline{\ell : \phi_\ell} \mid \phi\}\big) &= \{\overline{\ell : \phi_\ell} \mid \phi\} \\
(\theta \circ (\ell \mapsto \phi_\ell))\big(\{\overline{\ell : \phi_\ell} \mid \phi\}\big) &= \theta\big(\{\overline{\ell : \phi_\ell}, \ell : \phi_\ell \mid \phi\}\big)
\end{aligned}$$

There are some difficulties here regarding multiple updates at the same field ℓ. We will establish the convention that if a field ℓ occurs twice in a list of assignments (as it might in θ or in a record type), then we ignore the one that occurs first.

C.3.2 Laws

We now check that our definition of $\theta(\tau)$ indeed satisfies the homomorphism laws (Fig. 3.7).

Proof. Clearly, each component of θ operates *pointwise*, and identically on non-record types. Thus, we only check here that the homomorphism laws apply at singleton record types of form $\{\ell : \phi\}$, carrying the rest of the fields through implicitly.

Laws for top and bottom are given by definition, so we check meets are preserved (joins are symmetric). By induction over the structure of θ:

$$\begin{aligned}
\mathrm{id}(\{\ell : \phi'\}) \wedge \mathrm{id}(\{\ell : \phi''\}) &= \{\ell : \phi' \wedge \phi''\} \\
&= \mathrm{id}(\{\ell : \phi'\} \wedge \{\ell : \phi''\}) \\
(\theta \circ [\ell \mapsto \phi])(\{\ell : \phi'\}) \wedge (\theta \circ [\ell \mapsto \phi])(\{\ell : \phi''\}) &= \theta(\{\ell : \phi\}) \wedge \theta(\{\ell : \phi\}) = \theta(\{\ell : \phi\}) \\
&= (\theta \circ [\ell \mapsto \phi])(\{\ell : \phi' \wedge \phi''\}) \\
&= (\theta \circ [\ell \mapsto \phi])(\{\ell : \phi'\} \wedge \{\ell : \phi''\})
\end{aligned}$$

□

C.3.3 Adjoints

As part of the example given in Section 3.4.1, I gave the table of adjoints for free_ℓ and forget_ℓ. I now show that these indeed fulfil the necessary laws. We focus on forget_ℓ ($[\ell \mapsto \mathbb{T}]$), and analogously to the previous proof we can use pointwise reasoning (with respect to both the fields and the composition of morphisms). Proofs for free_ℓ ($[\ell \mapsto \mathbb{\bot}]$) follow exactly analogously.

The key idea behind the proofs is that an application of free/forget sends the field specifically to top and bottom, leading to either a constraint being always satisfied along the considered field ℓ, or only ever being satisfied when the field is exactly $\mathbb{\bot}/\mathbb{T}$.

Firstly, I prove that left-adjoints work:

$$\tau' \leqslant \underbrace{[\ell \mapsto \mathbb{T}]}_{\theta}(\tau'') \quad \cong \quad \underbrace{[\ell \mapsto \mathbb{\bot}]}_{\overleftarrow{\theta}}(\tau') \leqslant \tau'' \;\&\; \underbrace{\mathbf{T}}_{\underline{\theta}(\tau')}$$

Proof. For brevity, I will only consider $\tau' = \{\ell : \phi'\}$ and $\tau' = \{\ell : \phi''\}$:

$$\underbrace{\{\ell : \phi'\}}_{\tau'} \leqslant [\ell \mapsto \mathbb{T}](\underbrace{\{\ell : \phi''\}}_{\tau''}) = \{\ell : \mathbb{T}\} \quad \cong \mathbf{T} \cong \quad \{\ell : \mathbb{\bot}\} = [\ell \mapsto \mathbb{\bot}](\underbrace{\{\ell : \phi'\}}_{\tau'}) \leqslant \underbrace{\{\ell : \phi''\}}_{\tau''}$$

the general case follows by carrying through the other fields of both records. □

Secondly, I prove right-adjoints also work:

$$\underbrace{[\ell \mapsto \mathbb{T}]}_{\theta}(\tau') \leqslant \tau'' \quad \cong \quad \tau' \leqslant \underbrace{\mathrm{id}}_{\overrightarrow{\theta}}(\tau'') \;\&\; \underbrace{\{\ell : \mathbb{T} \mid \mathbb{\bot}\} \leqslant \tau''}_{\underline{\theta}(\tau'')}$$

Proof. Analogously as before:

$$\{\ell : \mathbb{T}\} = [\ell \mapsto \mathbb{T}](\underbrace{\{\ell : \phi'\}}_{\tau'}) \leqslant \underbrace{\{\ell : \phi''\}}_{\tau''} \quad \cong \quad \mathbb{T} = \phi'' \quad \cong \quad \{\ell : \mathbb{T} \mid \mathbb{\bot}\} \leqslant \{\ell : \phi''\}$$

as this is satisfied only iff $\mathbb{T} = \phi''$, we can add the always-satisfied constraint $\tau' \leqslant \tau''$ ($\phi' \leqslant \phi''$), yielding:

$$\underbrace{\{\ell : \phi'\}}_{\tau'} \leqslant \mathrm{id}(\underbrace{\{\ell : \phi''\}}_{\tau''}) \;\&\; \{\ell : \mathbb{T} \mid \mathbb{\bot}\} \leqslant \underbrace{\{\ell : \phi''\}}_{\tau''}$$

□

This shows that the described homomorphisms are indeed well-behaved in constraint solving.

C.3.4 Record extension

Lastly, I sketch the equivalence of the rules Ext (quantifying over record types) and HExt (applying forget_ℓ). For completeness, they are given below:

Ext

$$\frac{\Gamma \vdash e : \left\{\ell' : \square, \overline{\ell : \phi_\ell} \,\middle|\, \phi\right\} \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \{\ell' = e' \mid e\} : \left\{\ell' : \tau', \overline{\ell : \phi_\ell} \,\middle|\, \phi\right\}}$$

HExt

$$\frac{\Gamma \vdash e : \tau \wedge \{\ell' : \square \mid \mathbb{T}\} \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \{\ell' = e' \mid e\} : \mathrm{forget}_{\ell'}(\tau) \wedge \{\ell' : \tau' \mid \mathbb{T}\}}$$

Proof. Firstly, note that the assumptions of the two rules are equivalent. The only difference is in the constraint on e – to see that the two are equivalent, notice that $\left\{\ell' : \square, \overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}$ are exactly the types of form $\tau \wedge \{\ell : \square \mid \top\}$ for some $\tau = \left\{\overline{\ell : \phi_\ell} \;\middle|\; \phi\right\}$ (unless $\ell : \bot\!\!\!\bot$, in which case the assumption also follows).

Since assumptions of the two rules are equivalent, it remains to show their consequences are as well. However, this also follows straightforwardly via the reasoning presented in the FL example given in Section 3.4 – $\text{forget}_\ell(\tau) \wedge \{\ell : \phi \mid \top\}$ sends the field ℓ in τ to ϕ. □

Is is then straightforward to see that constraint generation (Sec. C.2) in the case Ext is correct, as it follows the form of the corresponding rule.

D Formal development of Fabric

This appendix sketches the formal development of Fabric, as described in Chapter 4.

D.1 Expressions

Syntax is given in terms of variable names x, field labels ℓ, variant tags T, integers n, and arithmetic operations π. I define the grammars of expressions e (Figure D.1), patterns p (Figure D.2), and values v (Figure D.3).

I note that in Fabric, as customary, we use the empty tuple $\langle\rangle$ as a unit value (and later, unit type). Tuples and variants are eliminated using pattern matching.

Operational semantics FL is given a call-by-value, small-step operational semantics. We focus on interesting cases.

Patterns Let us denote substitution of values in patterns by $[v/x]p$. Then the substitution $\overline{v/x}$ for which $\left[\overline{v/x}\right]p = v$ is unique if it exists (taking the wildcard tag '_' as equal to any tag T). If it does, we denote it $\mathrm{unpack}_p(v)$. Then we have the following:

$$\begin{aligned} \mathrm{let}\, p = v \,\mathrm{in}\, e &\rightsquigarrow \left[\mathrm{unpack}_p(v)\right] e \\ (\lambda p.\, e)\, v &\rightsquigarrow \left[\mathrm{unpack}_p(v)\right] e \\ \mathrm{match}\, v \,\mathrm{with}\, \overline{p \Rightarrow e} &\rightsquigarrow \left[\mathrm{unpack}_p(v)\right] e \qquad \text{for first matching } p \text{ among } \overline{p} \end{aligned}$$

Where $\mathrm{unpack}_p(v)$ is undefined, the state on the left-hand side is stuck.

Records Besides FL's usual record operations, Fabric also includes:

- Checked projections, which test whether the record contains a given field and return it if it is present. The operation returns tagged values:

$$\begin{aligned} \left\{\overline{\ell = v}\right\}_?\ell &\rightsquigarrow \mathrm{Some}\, v && \text{if } \ell \text{ occurs in } \overline{\ell} \\ \left\{\overline{\ell = v}\right\}_?\ell &\rightsquigarrow \mathrm{None}\, \langle\rangle && \text{if } \ell \text{ does not occur in } \overline{\ell} \end{aligned}$$

- Record restriction, which removes a present field from a record:

$$\left\{\ell = v, \overline{\ell = v}\right\} \setminus \ell \rightsquigarrow \left\{\overline{\ell = v}\right\}$$

Abbreviation-casts Abbreviation-casts are run-time no-ops, as they are intended to coerce between values of *effectively* the same type – but with some parts of that type made opaque by a nominal type.

$$e : \tau \blacktriangleright \tau' \rightsquigarrow e$$

Because of this rule, Progress is stated only *up to nominal abbreviations*.

$$
\begin{array}{rl}
e \ ::= & x \mid \mathrm{let}\, p = e \,\mathrm{in}\, e \qquad \text{(variable, let-binding)} \\
\mid & \lambda p.\, e \mid e\, e \qquad \text{(function, application)} \\
\mid & \mathrm{match}\, e \,\mathrm{with}\, \overline{p \Rightarrow e} \qquad \text{(pattern matching)} \\
\mid & n \mid \pi(\overline{e}) \qquad \text{(integer literal, integer arithmetic)} \\
\mid & \langle \overline{e} \rangle \qquad \text{(tuple)} \\
\mid & \{\overline{\ell = e}\} \qquad \text{(record construction)} \\
\mid & e.\ell \qquad \text{(record projection)} \\
\mid & e_?\ell \qquad \text{(record checked projection)} \\
\mid & \{\ell = e \mid e\} \qquad \text{(record extension)} \\
\mid & e \setminus \ell \qquad \text{(record restriction)} \\
\mid & T\, e \qquad \text{(variant tagging)} \\
\mid & e : \tau \blacktriangleright \tau \qquad \text{(abbreviation-cast)}
\end{array}
$$

Figure D.1: Expressions e.

$$
\begin{array}{rl}
p \ ::= & x \qquad \text{(atom)} \\
\mid & n \qquad \text{(integer)} \\
\mid & \langle \overline{p} \rangle \qquad \text{(tuple)} \\
\mid & \{\overline{\ell : p}\} \qquad \text{(record)} \\
\mid & T\, p \qquad \text{(variant)} \\
\mid & \_\, v \qquad \text{(variant wildcard)}
\end{array}
$$

Figure D.2: Patterns p, used in expressions with binders.

$$
\begin{array}{rl}
v \ ::= & n \qquad \text{(integer)} \\
\mid & \lambda p.\, e \qquad \text{(function)} \\
\mid & \langle \overline{v} \rangle \qquad \text{(tuple)} \\
\mid & \{\overline{\ell = v}\} \qquad \text{(record)} \\
\mid & T\, v \qquad \text{(variant)}
\end{array}
$$

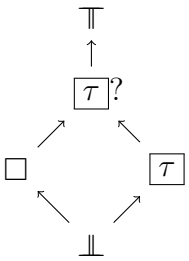
Figure D.3: Values v.

D.2 Types

I define the grammars of types τ (Figure D.4), including record field types ϕ (Figure D.5) and variant case types κ (Figure D.6). For simplicity, I only present the simply-typed version of Fabric, but extension with polymorphism amounts adding type variables α to τ, and annotated atoms $x : \tau$ to patterns p.

The syntax refers to nominal types t. To simplify presentation, I assume the set of nominal types and definitions $\mathbf{type}\, t = \tau$ is fixed, so that τ_t is the definition of a nominal type t.

Subtyping We focus on interesting cases. Firstly, subtyping for fields is given by the this Hasse diagram:



For case types, it is given by $\bot\!\!\!\bot \leqslant \boxed{\tau} \leqslant \top$ (i.e. compatible with the above diagram).

For subtyping of records and variants, we exactly follow the formulation in FL:

$$\frac{\forall \ell.\, \tau_1.\ell \leqslant \tau_2.\ell}{\tau_1 \leqslant \tau_2} \qquad \frac{\forall T.\, \tau_1.T \leqslant \tau_2.T}{\tau_1 \leqslant \tau_2}$$

where we use the type-level field type projection operator, as defined for FL record types. It is worth noting that joins and meets of records and variants are, analogously to subtyping, defined pointwise.

Note that an 'open' record type (with unknown other fields) is given by $\left\{\overline{\ell : \phi} \,\middle|\, \top\right\}$, while a 'closed' one (with all other fields absent) is $\left\{\overline{\ell : \phi} \,\middle|\, \square\right\}$. Likewise, an 'open' variant type (with other cases possible) is given by $\left[\overline{T : \kappa} \mid \top\right]$, while a 'closed' variant type (with other cases impossible) is $\left[\overline{T : \kappa} \mid \bot\!\!\!\bot\right]$.

Typing I give complete typing rules for Fabric in Figure D.7. I note the following:

Abbreviations We use the function $\mathrm{expand}(\tau)$, which produces a type τ' in which we substitute all nominal types for their definitions τ_t. We type abbreviation-casts $e : \tau \blacktriangleright \tau'$ by checking $\mathrm{expand}(\tau) = \mathrm{expand}(\tau')$, ensuring the operation is safe.

Record operations As shown in Section 3.4.3, we use homomorphisms free and forget for typing record operations. For brevity, I will use the abbreviation $\mathrm{update}_\ell(\tau, \phi)$.

Patterns The typing rules for patterns are problematic to describe in a non-ad-hoc manner, which is why I only provide a sketch of the design. They interact with polymorphism in a particular way, as each bound variable might introduce a type variable.

Patterns are typed with an operator $\bigoplus \overline{p}$, returning a type τ such that all values of type τ match at least one of $\overline{p}$. For example, $\bigoplus$ takes the union of possible variant cases and required field tags. We do not allow matching on mixed types, e.g. variants and integers. We also use an operator $@p$, returning an environment Γ with variables bound in p, where types given to variables are compatible with $\bigoplus \overline{p}$.

The trick to typing untagging lies in using a type with no tag-specific cases, but a default expressing all possible cases – specifically, $[\ \cdot \mid \boxed{\tau}\ ]$ where all possible tags contain subtypes of τ – as $\bot\!\!\!\bot \leqslant \boxed{\tau}$.

$$
\begin{array}{rl}
\tau ::= \top & \text{(top)} \\
\mid \mathrm{int} & \text{(integer)} \\
\mid \tau \to \tau & \text{(function)} \\
\mid \langle \overline{\tau} \rangle & \text{(tuple)} \\
\mid \left\{ \overline{\ell : \phi} \,\middle|\, \phi \right\} & \text{(record)} \\
\mid \left[\overline{T : \kappa} \mid \kappa \right] & \text{(variant)} \\
\mid \bot & \text{(bottom)}
\end{array}
$$

Figure D.4: Types τ.

$$
\begin{array}{rl}
\phi ::= \mathbb{T} & \text{(top – unknown)} \\
\mid \boxed{\tau}? & \text{(optional)} \\
\mid \boxed{\tau} & \text{(present)} \\
\mid \square & \text{(absent)} \\
\mid \mathbb{\perp} & \text{(bottom – impossible)}
\end{array}
$$

Figure D.5: Field types ϕ. These are similar to FL, but extended with optional fields $\boxed{\tau}?$.

$$
\begin{array}{rl}
\kappa ::= \mathbb{T} & \text{(top – unknown)} \\
\mid \boxed{\tau} & \text{(possible)} \\
\mid \mathbb{\perp} & \text{(bottom – impossible)}
\end{array}
$$

Figure D.6: Case types κ – somewhat analogical to field types ϕ, but simpler and used for specifying variant types.

$$\textsc{Sub}\ \frac{\Gamma \vdash e : \tau \quad \tau \leqslant \tau'}{\Gamma \vdash e : \tau'} \qquad \textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash e : \tau} \qquad \textsc{Lit}\ \frac{}{\Gamma \vdash n : \mathrm{int}} \qquad \textsc{Arith}\ \frac{\overline{\Gamma \vdash e : \mathrm{int}}}{\Gamma \vdash \pi(\overline{e}) : \mathrm{int}}$$

$$\textsc{Fun}\ \frac{\Gamma, @p \vdash e : \tau}{\Gamma \vdash \lambda p.\, e : \bigoplus p \to \tau} \qquad \textsc{Apply}\ \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e : \bigoplus p \quad \Gamma, @p \vdash e' : \tau'}{\Gamma \vdash \mathrm{let}\, p = e\, \mathrm{in}\, e' : \tau'} \qquad \textsc{Match}\ \frac{\Gamma \vdash e : \bigoplus \overline{p_i} \quad \overline{\Gamma, @(p_i) \vdash e_i : \tau}}{\Gamma \vdash \mathrm{match}\, e\, \mathrm{with}\, \overline{p_i \Rightarrow e_i} : \tau}$$

$$\textsc{Tuple}\ \frac{\overline{\Gamma \vdash e : \tau}}{\Gamma \vdash \langle \overline{e} \rangle : \langle \overline{\tau} \rangle} \qquad \textsc{Cons}\ \frac{\overline{\Gamma \vdash e_\ell : \tau_\ell}}{\Gamma \vdash \{\overline{\ell = e_\ell}\} : \{\overline{\ell = \tau_\ell} \,|\, \square\}} \qquad \textsc{Tag}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash T\, e : [T : \tau \mid \bot]}$$

$$\textsc{Project}\ \frac{\Gamma \vdash e : \{\ell : \tau \,|\, \top\}}{\Gamma \vdash e.\ell : \tau} \qquad \textsc{CheckProject}\ \frac{\Gamma \vdash e : \{\ell : \tau \,|\, \top\}}{\Gamma \vdash e_?\ell : [\mathrm{Some} : \tau, \mathrm{None} : \langle\rangle \mid \bot]}$$

$$\textsc{Extend}\ \frac{\Gamma \vdash e : \tau \quad \tau \leqslant \{\ell : \square \,|\, \top\} \quad \Gamma \vdash e_\ell : \pi}{\Gamma \vdash \{\ell = e_\ell \,|\, e\} : \mathrm{update}_\ell(\tau, \boxed{\pi})} \qquad \textsc{Restrict}\ \frac{\Gamma \vdash e : \tau \quad \tau \leqslant \{\ell : \boxed{\pi} \,|\, \top\}}{\Gamma \vdash e \setminus \ell : \mathrm{update}_\ell(\tau, \square)}$$

$$\textsc{AbbrevCast}\ \frac{\Gamma \vdash e : \tau \quad \mathrm{expand}(\tau) = \mathrm{expand}(\tau')}{\Gamma \vdash (e : \tau \blacktriangleright \tau') : \tau'}$$

Figure D.7: Typing rules for Fabric.

Type safety I state and conjecture type safety – Preservation and Progress theorems – for Fabric. Their most interesting aspect is that Preservation is only *up to nominal types*. This could be fixed by introducing tagging values with nominal types.

Conjecture D.1 (Preservation)**.** *If* $\cdot \vdash e : \tau$ *and* $e \rightsquigarrow e'$*, then* $\cdot \vdash e' : \tau'$ *s.t.* $\mathrm{expand}(\tau) = \mathrm{expand}(\tau')$.

Conjecture D.2 (Progress)**.** *If* $\cdot \vdash e : \tau$*, then* $e = v$ *or* $e \rightsquigarrow e'$ *for some* e'.

E WebAssembly code generation

The OCaml program using Binaryer in Figure 4.6 produces WebAssembly code with the following textual representation:

```
(module
 (type $0 (struct (field (mut i32)) (field i32)))
 (type $1 (func (param i32)))
 (type $2 (func))
 (import "spectest" "print_i32"
   (func $print_i32 (type $1) (param i32)))
 (memory $__memory__ 10 10)
 (export "main" (func $__fun1))
 (start $__fun1)
 (func $__fun1 (type $2)
  (local $0 anyref)
  i32.const 42
  i32.const 1295
  struct.new $0
  local.set $0
  local.get $0
  ref.cast (ref $0)
  local.get $0
  ref.cast (ref $0)
  struct.get $0 0
  local.get $0
  ref.cast (ref $0)
  struct.get $0 1
  i32.add
  struct.set $0 0
  local.get $0
  ref.cast (ref $0)
  struct.get $0 0
  call $print_i32
 )
)
```

Primitives marked in red use the WasmGC extension.

F Type safety of Star

I give the proof of type safety theorems in the simply-typed case without parametric polymorphism. I give some useful lemmas and prove the Progress and Preservation theorems.

These are largely an extension of the proofs given by Pierce [50] for $\lambda_{<:}$ (simply-typed lambda calculus with subtyping and records). We additionally consider variants and Star's arrays.

In the following text, $\Gamma \vdash e :: \tau$ states that the *strongest* (least in the type lattice) type of e is τ, defined:

$$\Gamma \vdash e :: \tau \iff (\forall \tau'.\, \Gamma \vdash e : \tau' \implies \tau \leqslant \tau')$$

For a value, any typing derivation that does not apply Sub yields the strongest type.

Lemmas

We start with a standard substitution lemma. Besides functions, it is also used for array comprehensions.

Lemma F.1 (Substitution)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$*, then* $\Gamma \vdash [e/x]e' : \tau'$*.*

Proof. By straightforward induction on the derivation of $\Gamma, x : \tau \vdash e' : \tau'$. □

We also include lemmas about values – their canonical form given their type and that they do not reduce.

Lemma F.2 (Canonical forms)**.** *We have the following canonical forms for well-typed values* v*:*

- *If* $\Gamma \vdash v : \mathrm{int}$*, then* $v = n$*.*
- *If* $\Gamma \vdash v : \mathrm{float}$*, then* $v = f$*.*
- *If* $\Gamma \vdash v : \tau \to \tau'$*, then* $v = \lambda x.\, e$*.*
- *If* $\Gamma \vdash v : \{\overline{\ell : \tau}\}$*, then* $v = \{\overline{\ell = v_\ell}, \ldots\}$ *(up to ordering).*
- *If* $\Gamma \vdash v : [\overline{T : \tau_T}]$*, then* $v = T\, v_T$*. In particular, we have* $\Gamma \vdash v_T : \tau_T$ *and* $\Gamma \vdash v :: [T : \tau_T]$*.*
- *If* $\Gamma \vdash v : [\sigma_1..\sigma_2]\tau$*, then* $v = \mathsf{Arr}(s, I)$*. In particular, there exist* σ *such that* $\sigma_1 \leqslant \sigma \leqslant \sigma_2$ *and* $\Gamma \vdash v :: [\sigma]\tau$*.*
- *If* $\Gamma \vdash v : \#$*, then* $v = \#n$*.*
- *If* $\Gamma \vdash v : \{|\overline{\ell : \tau}|\}$*, then* $v = \{|\overline{\ell = v_\ell}, \ldots|\}$ *(up to ordering).*
- *If* $\Gamma \vdash v : [\![\overline{\ell : \tau}]\!]$*, then* $v = [\![\overline{\ell = v_\ell}, \ldots]\!]$ *(up to ordering).*

Proof. By inversion on the typing derivation $\Gamma \vdash v : \tau$, eliminating impossible cases which do not match the syntax of values. Where the implicit subtyping rule is applied at the end of a derivation, we proceed from the subtype.

Note that as noted canonical forms might have a stronger type than advertised by the well-typedness assumption – in particular, records (analogously products and concatenations) might contain more fields than in their type (we denote these by . . .). This follows from the subtyping rules. □

Lemma F.3 (Values are normal forms)**.** *For any value v there exist no e such that $v \rightsquigarrow e$, nor $v \rightsquigarrow \lightning$.*

Proof. By routine check of axioms and rules of $\rightsquigarrow$. □

We then assert that our subtyping relation is well-behaved.

Lemma F.4 (Subtyping lattice)**.** *The relation $\leqslant$ together with appropriate meets $\wedge$ and joins $\vee$ forms a distributive lattice $(\leqslant, \top, \bot, \wedge, \vee)$.*

Proof. By routine check of all the axioms of a distributive lattice. We also need to construct meets $\wedge$ and joins $\vee$ and show that they agree with the subtyping order.

We briefly consider just the cases specific to Star for the construction of $\wedge$ and $\vee$. Product and concatenation shapes behave exactly as records under subtyping, so their their meet takes the union of fields and the elementwise meet of field types. For arrays:

$$[\sigma_1'..\sigma_2']\tau' \wedge [\sigma_1''..\sigma_2'']\tau'' \stackrel{\text{def}}{=} [(\sigma_1' \vee \sigma_1'')..(\sigma_2' \wedge \sigma_2'')](\tau' \wedge \tau'')$$

Joins are constructed analogously. □

Due to transitivity of $\leqslant$ (by Trans), we can replace repeated application of Sub with one. Subtyping forming a distributive lattice is also a necessary assumption for algebraic subtyping to apply.

Lastly, we specify crucial lemmas about indexing – particularly the correspondence of the structurally-in-bounds relation $\blacktriangleleft$ with typing.

Lemma F.5 (Typing agrees with indexing)**.** $\cdot \vdash s :: \sigma$ *and* $\cdot \vdash v : \iota(\sigma)$ *if and only if* $v \blacktriangleleft s$.

Proof. Right to left is a straightforward check by structural induction on v and s. The other direction follows by structural induction on the typing derivations of s and v, checking that appropriate cases of $\blacktriangleleft$ follow. Avoiding loss of information about s by considering its strongest type ensures that v contains all the necessary dimensions (and possibly more). □

It is worth noting that given an array of type $[\sigma_1..\sigma_2]\tau$, for its shape's strongest type σ it follows $\sigma_1 \leqslant \sigma \leqslant \sigma_2$. Thus, for any index of type $\eta \leqslant \iota(\sigma_1)$ we also have $\eta \leqslant \iota(\sigma)$ and the lemma holds for v and the array's shape – this is key to our proof of Progress.

Lemma F.6 (Casting is idempotent)**.** *If $v \lhd s$, then*

$$(v \oslash s) \oslash s = v \oslash s$$

Lemma F.7 (In-bounds relations agree with indexing)**.** *If $v \lhd s$, then $(v \oslash s) \lhd s$ – in particular, $(v \oslash s) \trianglelefteq s$.*

As a consequence, if an index v occurs in the bounds of the shape s ($v \lhd s$) of an array $\mathsf{Arr}(s, I)$, then StepIndex is sound (as $v \oslash s$ is indeed in the domain of I).

Lemma F.8 (Covariance of ι)**.** *If $\sigma \leqslant \sigma'$, then $\iota(\sigma) \leqslant \iota(\sigma')$.*

Proof. By structural induction on σ and σ', with the proof following by rules of $\leqslant$ and definition of ι. □

Theorems

Theorem F.9 (Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : \tau$*.*

Proof. We perform induction on $\Gamma \vdash e : \tau$. A common case is where we find the reduction $e \rightsquigarrow e'$ derived from Cong, we can usually finish by inductive hypothesis – which states the subject of Cong preserves the type. The most interesting case is Array, where we have to carry through the type of each array element via the Substitution lemma.

Case Sub We have $\Gamma \vdash e : \tau'$ for some $\tau' \leqslant \tau$. Since $e \rightsquigarrow e'$ we can invoke the inductive hypothesis and obtain $\Gamma \vdash e' : \tau'$ immediately, and finish by Sub.

Case Var Impossible – no $\rightsquigarrow$ rule applies to variables.

Case Int, Float Cannot happen – e is a value.

Case Let By inversion on $\rightsquigarrow$, either StepLet or Cong apply. In the former case, we finish by Substitution.

Case Lambda Cannot happen – e is a value.

Case Apply By inversion on $\rightsquigarrow$, we reduce by StepApply – and can finish by Substitution.

Case Record By inversion on $\rightsquigarrow$, only Cong applies and we finish immediately.

Case RecordProj By inversion on $\rightsquigarrow$, StepRecordProj or Cong apply to our expression $e = e^*.\ell$. In the former, we must have $e^* = v^* = \{\overline{\ell = v_\ell}\}$, so that $\Gamma \vdash e : \{\ell : \tau_\ell\}$, $\tau = \tau_\ell$ and $e' = v_\ell$. Since e is well-typed, we must have $\Gamma \vdash v_\ell : \tau_\ell$ as required.

Case Tag By inversion on $\rightsquigarrow$, only Cong applies.

Case Match Similarly to the dual RecordProj. By inversion, we either have Cong (immediate) or StepMatch, in which case $e = \text{match}\, T\, v \,\text{with}\, \overline{T \Rightarrow e_T}$ for some value v (canonical form $T\, v$ of the match target). Since $e \rightsquigarrow e'$, it must be that $e' = [v/x]e_T$ (StepMatch) and we finish by Substitution at $\Gamma, x : \tau_T \vdash e_T : \tau$ (assumption of Match).

Case Array We have $\tau = [\sigma]\tau^*$. By inversion on $\rightsquigarrow$, either we evaluate the shape via Cong (and finish immediately), or return an array via StepArray. We check the latter – $e = \Phi\, x\, [e^{\mathrm{S}}].\, e^*$, $e' = \mathsf{Arr}(s, J)$, and we have the inductive hypothesis at each element $v \mapsto e_v$ in J. By the inductive case e is typed via Array, so we have a well-typed shape value $\Gamma \vdash s : \sigma$ and know $\Gamma, x : \iota(\sigma) \vdash e^* : \tau^*$. StepArray reduces via indices v such that $v \lhd s$, from which we know it follows $\Gamma \vdash v : \iota(\sigma)$. By the inductive hypothesis and Substitution at $e_v = [v/x]e^*$ it thus follows that $\Gamma \vdash e_v : \tau^*$ at all v. We thus ascribed the right types to both the shape s and all elements $v \mapsto e_v$ of J and we are done via ArrayLit.

Case Index By inversion on $\rightsquigarrow$ we either have Cong or StepIndex, checking the latter. In that case $e = v[e^\circ]$ where $\Gamma \vdash v : [\sigma_1..\sigma_2]\tau$ and $\Gamma \vdash e^\circ : \iota(\sigma_1)$. we know $v = \mathsf{Arr}(s, I)$ (canonical form). Since v must be well-typed (by inversion, with ArrayVal), each value in I – including v' – must have type τ, finishing the case.

Case Shape Similarly to Index – in StepShape we must have the canonical form $e = v = \mathsf{Arr}(s, I)$, with $\Gamma \vdash e : [\sigma_1..\sigma_2]\tau$, $\Gamma \vdash v : [\sigma]\tau$ for some $\sigma_1 \leqslant \sigma \leqslant \sigma_2$, and $e' = s$. But then we immediately have $\Gamma \vdash s : \sigma$ and $\sigma \leqslant \sigma_2$ as required so $\Gamma \vdash s : \sigma_2$ by Sub.

Case Broadcast We check StepBroadcast. By the definition of broadcasting we find that if it does not get stuck, the result must be of type $\sigma \wedge \sigma'$.

Case Sized By inversion on $\rightsquigarrow$, only Cong applies.

Case Product, Concat Analogously to Record.

Case ProductProj, ConcatProj Like RecordProj.

□

Theorem F.10 (Progress)**.** *If $\cdot \vdash e : \tau$, then either e is a value, $e \rightsquigarrow e'$ for some e', or $e \rightsquigarrow \lightning$.*

Proof. As before, we apply induction on $\cdot \vdash e : \tau$. Note that e is necessarily closed, since Γ is empty. Where we say we can proceed by Cong, CongErr might instead apply if an error is raised – this still meets the condition in the statement of the theorem. The most interesting case is **Index**, where we make use of the lemma that typing agrees with the structurally-in-bounds relation $\blacktriangleleft$.

Case Sub Finish by immediate application of the inductive hypothesis at the assumption of Sup, $\cdot \vdash v : \tau' \geqslant \tau$.

Case Var Impossible – no $\rightsquigarrow$ rule applies to variables.

Case Int, Float Immediate, since e must be a value.

Case Let If the let-bound expression is not a value, then we finish by the inductive hypothesis and Cong. Otherwise, we finish by StepLet.

Case Lambda Immediate, since $e = \lambda x.\, e'$ is a value.

Case Apply Let $e = e'\ e''$. If either e' or e'' are not values, then Cong applies. Otherwise $e = v'\ v''$. But $\cdot \vdash v' : \tau'' \to \tau$ (assumption of Apply) and we know v' is in canonical form $v' = \lambda x.\, e^*$, so we can finish by StepApply.

Case Record If not all expressions occurring in fields are values, we can proceed by Cong. Otherwise, e is already a value.

Case RecordProj Let $e = e^*.\ell$. If e^* is not a value, we can apply Cong. Otherwise, $e^* = v^*$ and by assumption of RecordProj, $\cdot \vdash v^* : \{\ell : \tau\}$, so $v^* = \{\ell = v, \ldots\}$. Thus ℓ occurs in v^* and we finish by StepRecordProj.

Case Tag Similarly to Record: if the tagged expression is not a value, we Cong. Otherwise, e is a value.

Case Match Dually to RecordProj: since the expression is well-typed, the target of the match is a tagged value that does occur in the case list, and we can finish by StepMatch.

Case Array Let us assume shape s is already a value (otherwise we have a Cong) in $e = \Phi\, x[s].\, e^*$ with $\tau = [\sigma]\tau$ and $\cdot \vdash s : \sigma$, $\cdot \vdash e : \tau$. In that case, we can immediately finish by StepArray.

Case ArrayLit Either the array literal is already a value – if all the elements in J have been evaluated – or we can evaluate by Cong.

Case INDEX Without loss of generality, let us take both operands to be values – otherwise, CONG applies. Let $e = v[v']$ so that $\cdot \vdash v : [\sigma_1..\sigma_2]\tau$ and $\cdot \vdash v' : \iota(\sigma_1)$ (assumptions of INDEX). Then $v = \mathsf{Arr}(s, I)$ (canonical), so that in particular $\cdot \vdash s :: \sigma$ and $\cdot \vdash v : [\sigma]\tau$ where $\sigma_1 \leqslant \sigma \leqslant \sigma_2$. But since $\sigma_1 \leqslant \sigma$ and thus $\iota(\sigma_1) \leqslant \iota(\sigma)$, by SUB we also have $\cdot \vdash v' : \iota(\sigma)$. Invoking the agreement of typing and in-bounds lemma on v' and s, we immediately get that $v' \blacktriangleleft s$. There remain two cases: either $v' \lhd s$, in which case we finish by STEPINDEX as v' must be in the domain of I (each index v^* in the domain has $v^* \lhd s$ thus $v^* \blacktriangleleft s$). Otherwise, v' is not in-bounds up to integer indices, i.e. $\neg(v' \lhd s) \wedge (v' \blacktriangleleft s)$, and we get STEPINDEXERR.

Case SHAPE Let $e = |e^*|$. If e^* is not a value, CONG. Otherwise, STEPSHAPE.

Case BROADCAST If both operands are not already values, CONG. Otherwise, STEPBROADCAST or STEPBROADCASTERR apply – with the two cases exclusive, depending on whether $\sqcap$ is defined.

Case SIZED If e is not already a value, we apply CONG.

Case PRODUCT, CONCAT Like RECORD.

Case PRODUCTPROJ, CONCATPROJ Like RECORDPROJ.

□