Breaking records

Structural subtyping as a language design principle

Jakub Bachurski (University of Cambridge)
Supervisors: Dominic Orchard and Alan Mycroft

15 April 2025

Introduction

Motivation

- People use dynamically typed languages {Python, Lua, ...}.

Motivation

- People use dynamically typed languages {Python, Lua, ...}.
- Static typing is a great idea!

Motivation

- People use dynamically typed languages {Python, Lua, ...}.
- Static typing is a great idea!
- $\implies$ We should improve {optional} static type systems for those languages.

Quack

Duck typing is a common idiom in dynamic languages.

We should aim to check it statically.

```python
def quack(x):
    return x.quack()
```

Clearly, `x` needs to be an object with a `quack` method. Notionally:

$$\mathrm{x} \leqslant \{\texttt{quack} : () \to \alpha\}$$

Quack!

We need this to work in more complex cases, e.g. involving higher-order functions:

```
def quack(x, is_duck):
    if random.randint(0, 1):
        f = lambda y: y.quack()
    else:
        f = lambda y: y.honk()
    return f(x)
```

`x` needs to have both a `quack` and a `honk`.

$$\mathsf{x} \leqslant \{\texttt{quack} : () \to \alpha, \texttt{honk} : () \to \alpha\}$$

Question. What typing discipline do we follow?

Nominal, structural, dynamic

Nominal

In nominal type systems, we compare types by **name**.

```
type 'a option = None | Some of 'a  (* this type, specifically! *)
let map (f : 'a -> 'b) (x : 'a option) =
  match x with
  |  None   ->  None
  |  Some x ->  Some (f x)
```

Structural

In structural type systems, we compare types by defined **structure**.

```
let map (f : 'a -> 'b) (x : [< `None | `Some of 'a ]) =
  match x with
  | `None   -> `None
  | `Some x -> `Some (f x)
```

Dynamic

With dynamic typing, we only check types at runtime.

```
let map f x =
  match x with
  | `None   -> `None
  | `Some x -> `Some (f x)
```

A gradient of type systems

Claim. **Structural** type systems have a special role in statically typing existing dynamic languages.

- *Generally*, we can **translate** a nominally typed language to a structurally typed one by **erasing names**.
- Afterwards, **erase types** entirely to move to a dynamically typed language. For *well-behaved* programs, **type inference** can recover types.
- Moving along this gradient we make the type system less restrictive – more programs type check.

$$\text{nominal} \xrightarrow{\text{erase names}} \text{structural} \xrightarrow{\text{erase types}} \text{dynamic}$$

Questions. Can these translations be formalised? Do they have an interesting form?

Ideas come to life

Contribution

Translation from **Featherweight Java** {Igarashi et al.} into a structurally typed **record calculus** {Cardelli and Mitchell}.

The source and target languages of this translation serve as **prototypical languages**:

- **FJ** {a core calculus of Java} – nominally typed OOP language.
- Record calculus – language with structurally typed records.
- Untyped record calculus – dynamic OOP lang. {a **core** of Python, Lua, JavaScript, ...}.

All of these rely on a notion of subtyping ($A \leqslant B$ – an A can be used in place of a B).

Lost in translation

Here is a simple example program in Featherweight Java:

```
class Bird extends Object {
    String name;
    Bird(String name) { super(); this.name = name; }
    String name() { return this.name; }
}
class Duck extends Bird {
    Duck(String name) { super(name); }
    String quack() { return "quack"; }
}

(new Duck("mallard")).quack()
```

Break on through

Translation preserving well-typedness into record calculus – λ calculus with extensible records $\{\{\ell = e \mid r\}$ updates r with $\ell = e\}$. The translation uses a prototype-based style.

$$\mathrm{Object}_{\mathrm{proto}} = \{\}$$
$$\mathrm{Object} = \lambda o.\, \lambda().\, o$$

$$\mathrm{Bird}_{\mathrm{proto}} = \{\mathrm{name} = \lambda\mathsf{this}.\, \lambda().\, \mathsf{this}.\mathrm{name}\}$$
$$\mathrm{Bird} = \lambda o.\, \lambda\mathrm{name}.\, \{\mathrm{name} = \mathrm{name} \mid \mathrm{Object}\, o\, ()\}$$

$$\mathrm{Duck}_{\mathrm{proto}} = \{\mathrm{quack} = \lambda\mathsf{this}.\, \lambda().\, \texttt{"quack"} \mid \mathrm{Bird}_{\mathrm{proto}}\}$$
$$\mathrm{Duck} = \lambda o.\, \lambda\mathrm{name}.\, \mathrm{Bird}\, o\, \mathrm{name}$$
$$\mathrm{Duck}_{\mathrm{quack}} = \lambda o.\, o.\mathrm{quack}\, o\, ()$$

$$\mathrm{Duck}_{\mathrm{quack}}\ (\mathrm{Duck}\, \mathrm{Duck}_{\mathrm{proto}}\, \texttt{"mallard"})$$

Are we structurally typing yet?

We have seen we can usefully translate from nominal to structural typing.

Ponder. Why are types in static languages mostly nominal?

Nominal types are a convenient assumption.

- It is natural to give names to things.
- *Can be* what you want {e.g. primitives} – want nominal **and** structural {Binder et al.}.
- Easier type inference – unification, Hindley-Milner.

Structural typing, especially in the presence of **subtyping**, is tricky to include in a language – it introduces **complexity**.

Question. What abstraction helps us manage this complexity?

Algebraic subtyping

Algebraic viewpoint

Subtyping $\leqslant$ is a partial order on types τ.

We usually consider type systems with an **implicit** coercion rule:

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

In **algebraic subtyping**, we consider $\leqslant$ which form a *distributive lattice* algebra – we have a meet $\wedge$ (least upper bound) and join $\vee$ (greatest lower bound) with axioms.

$$\tau \leqslant \tau' \iff \tau = \tau \wedge \tau' \iff \tau' = \tau \vee \tau'$$

Nice algebraic properties lead us to nice properties of the type system: like ML-style principal type inference {orig. due to Dolan in **MLsub**; Parreaux: simple(r) constraint solving}.

Example type lattice

$$\begin{aligned}
\tau ::= {} & \top \mid \bot && \text{(top, bottom)} \\
& \mid \text{int} \mid \text{float} && \text{(primitives)} \\
& \mid \tau \to \tau && \text{(functions)} \\
& \mid \{\ell : \tau, \dots\} && \text{(records)}
\end{aligned}$$

τ	π	$\tau \wedge \pi$	$\tau \vee \pi$
int	int	int	int
int	float	$\bot$	$\top$
$\text{int} \to \text{int}$	$\bot \to \top$	$\text{int} \to \text{int}$	$\bot \to \top$
$\{\ell : \text{int}\}$	$\{\ell' : \text{float}\}$	$\{\ell : \text{int}, \ell' : \text{float}\}$	$\{\}$
$\{\ell : \text{int}\} \to \text{int}$	$\{\ell' : \text{float}\} \to \top$	$\{\} \to \text{int}$	$\{\ell : \text{int}, \ell' : \text{float}\} \to \top$

Asking the right question

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

Type inference from a Curry-Howard perspective: *what is the statement τ proven by e?*

Determine constraint system for expression e, then find general solution for its type τ.

Sketch of constraint solver {Pottier; Parreaux}:

- Intro type variables $\alpha, \beta, \ldots$ and track bounds $\alpha_{\mathrm{lo}} \leqslant \alpha \leqslant \alpha_{\mathrm{hi}}$ (**constraint graph**).
- Factor: $a \vee b \leqslant c \iff (a \leqslant c) \,\&\, (b \leqslant c)$ and $a \leqslant b \wedge c \iff (a \leqslant b) \,\&\, (a \leqslant c)$.
- Take transitive closure: $(a \leqslant b) \,\&\, (b \leqslant c) \implies a \leqslant c$. Check $\alpha_{\mathrm{lo}} \leqslant \alpha_{\mathrm{hi}}$.

Example constraint-based type inference

expression e	$\rightsquigarrow$	**constraints** c	$\rightsquigarrow$	**type** τ
$\lambda x.\, \text{if}\, x.\text{flag}$ $\text{then}\, x.\text{foo}$ $\text{else}\, x.\text{bar} + x.\text{baz}$	$\rightsquigarrow$	$\begin{cases} \mathsf{x} \leqslant \{\text{flag} : \beta\} \\ \beta \leqslant \text{bool} \\ \mathsf{x} \leqslant \{\text{foo} : \iota\} \\ \mathsf{x} \leqslant \{\text{bar} : \iota_1\} \\ \mathsf{x} \leqslant \{\text{baz} : \iota_2\} \\ \iota_1, \iota_2 \leqslant \text{int} \end{cases}$	$\rightsquigarrow$	$\{\text{flag} : \text{bool}, \text{foo} : \iota, \text{bar} : \text{int}, \text{baz} : \text{int}\} \to \iota \vee \text{int}$

Limitations and extensions

There are limitations {Dolan's MLsub}: the **polarity restriction**.

Recent work {Parreaux's MLstruct} improves on this by considering a **Boolean lattice**.

However, we still cannot type common record operations (in FJ translation!) without **row polymorphism**, as done for systems without subtyping {preliminarily: Marques et al.}.

Type inference with lattice homomorphisms

Contribution

(Meta)functions on types – **type lattice homomorphisms** with "adjoints" – work in type inference $\implies$ can infer types for extensible records via drop_ℓ homomorphisms:

$$\mathrm{drop}_{\mathrm{foo}}(\{\mathrm{foo} : \mathrm{int}, \mathrm{bar} : \mathrm{float}\}) = \{\mathrm{bar} : \mathrm{float}\}$$

Example. Take the flag field of a record a and update it to the result of not:

$$\diamond \vdash \lambda a. \left\{\mathrm{flag} = a.\mathrm{flag}.\mathrm{not}\,() \;\middle|\; a \setminus \mathrm{flag}\right\}$$
$$: \alpha \wedge \{\mathrm{flag} : \{\mathrm{not} : () \to \nu\}\} \to \mathrm{drop}_{\mathrm{flag}}(\alpha) \wedge \{\mathrm{flag} : \nu\}$$

$\{\{\ell = e \mid r\}$ extends r with $\ell = e$; $r \setminus \ell$ removes ℓ from r.$\}$

Future work. General, constraint-based formulation of type inference for algebraic subtyping with this extension – prior art focuses on specific cases.

Language design – Fabric

Fabric's design

Fabric

- purely functional – imperative mutability is hard as always!
- statically typed with a focus on **structural types** and **subtyping**
- features usual **extensible** records and variants, and richer arrays.
- admits ML-style type inference thanks to algebraic subtyping.
- compiles to WebAssembly.

Principle. Replicate flexibility of dynamic languages, but safely (structural subtyping).

Goal. Extend existing languages – or their tooling – with Fabric's features.

Fabric's implementation

Prototype compiler in OCaml, making ideas come to life.

- I implemented Parreaux's Simple-sub/MLstruct-style algebraic subtyping type inference. Difficulty: type simplification.
- WebAssembly code generation using the Binaryen toolchain (now with GC!). Can optimize and lower further to native code. Difficulty: unstable toolchain.

WebAssembly is a modern portable target with interesting verification/design work.

Future work. Investigate performance implications of different runtime representations for record and variant types, which admit structural subtyping.

Structuring arrays

Why arrays?

We look at **array programming** as a practical reason for revisiting structural typing. Array programs might look like this (matrix multiplication):

$$C = \Phi\, i.\, \Phi j.\, \Sigma\, k.\, A[i,k] \cdot B[k,j] \qquad \textbf{(pointful style)}$$

$$C = \mathrm{sum}^{(1)}\left(\mathrm{expand}^{(2)}(A) \cdot \mathrm{expand}^{(0)}(B)\right) \qquad \text{(point-free style)}$$

$$C = \mathrm{map}\, A\, (\lambda a.\, \mathrm{map}\, B^T\, (\lambda b.\, \mathrm{sum}\, (\mathrm{map}_2\, (\times)\, a\, b))) \qquad \text{(combinator-based)}$$

Type checking array programs is tricky. There are array type systems, but most resort to dependent types {type-level arithmetic}. Practitioners stick to blissful untypedness.

Question. Can structural typing help construct a type system for array programming?

Seeing stars

Existing systems rely on *n*-dimensional arrays (tensors) – a *shape* is a tuple of integers.

$$\text{shape}\begin{bmatrix}(0,0) & (0,1) & (0,2)\\ (1,0) & (1,1) & (1,2)\end{bmatrix} = (2,3)$$

Contribution

Novel array calculus: **Star**. Features algebraic, structurally typed array shapes, admitting ML-style type inference via algebraic subtyping.

Key idea. **product** $\{\!|\ell : s, \ell' : s', \ldots|\!\}$ and **concatenation** $[\![T : s, T' : s', \ldots]\!]$ shapes – closely tied to structural record and variant types, which are used for indexing.

Under review for ARRAY 2025.

Example of structural shapes

An array of shape of type:

$$\{\!|\mathrm{row} : [\![\mathrm{Top} : \#, \mathrm{Mid} : \#, \mathrm{Bot} : \#]\!], \mathrm{col} : [\![\mathrm{Left} : \#, \mathrm{Mid} : \#, \mathrm{Right} : \#]\!]|\!\}$$

is indexed by values of the type of records-of-variants (e.g. $\{\mathrm{row} : \mathrm{Top}\, 5, \mathrm{col} : \mathrm{Mid}\, 4\}$):

$$\{\mathrm{row} : [\mathrm{Top} : \mathrm{int}, \mathrm{Mid} : \mathrm{int}, \mathrm{Bot} : \mathrm{int}], \mathrm{col} : [\mathrm{Left} : \mathrm{int}, \mathrm{Mid} : \mathrm{int}, \mathrm{Right} : \mathrm{int}]\}$$

We can visualise this shape as a *padded matrix*, composed of 9 regions:

$$\begin{array}{cc} & \text{col} \\ \text{row} & \left[\begin{array}{c|c|c} \text{Top, Left} & \text{Top, Mid} & \text{Top, Right} \\ \hline \text{Mid, Left} & \text{Mid, Mid} & \text{Mid, Right} \\ \hline \text{Bot, Left} & \text{Bot, Mid} & \text{Bot, Right} \end{array}\right] \end{array}$$

This is much more familiar than tracking a shape $(t + n + b)(l + m + r)$ (polynomial!).

Conclusions

Summary

- **Structural subtyping** is a promising direction to *revisit* for designing optional static type systems for dynamic languages – formalise why via translations.
- **Algebraic subtyping** would enable type inference for such type systems. We can find novel extensions, and e.g. type records with type lattice homomorphisms.
- **Designing languages** like Fabric is a useful way of playing around with extensions for existing systems, even if the language is not meant for practical use.
- **Revisiting old problems** and rephrasing them can be good – there is an interesting structural type system for array programs, without dependent types.

Summary

- **Structural subtyping** is a promising direction to *revisit* for designing optional static type systems for dynamic languages – formalise why via translations.
- **Algebraic subtyping** would enable type inference for such type systems. We can find novel extensions, and e.g. type records with type lattice homomorphisms.
- **Designing languages** like Fabric is a useful way of playing around with extensions for existing systems, even if the language is not meant for practical use.
- **Revisiting old problems** and rephrasing them can be good – there is an interesting structural type system for array programs, without dependent types.

Thank you!