First-Class Subtypes

Jeremy Yallop
University of Cambridge
jeremy.yallop@cl.cam.ac.uk

Stephen Dolan
University of Cambridge
stephen.dolan@cl.cam.ac.uk

First class type equalities, in the form of generalized algebraic data types (GADTs), are commonly found in functional programs. However, first-class representations of other relations between types, such as subtyping, are not yet directly supported in most functional programming languages.

We present several encodings of first-class subtypes using existing features of the OCaml language (made more convenient by the proposed modular implicits extension), show that any such encodings are interconvertible, and illustrate the utility of the encodings with several examples.

1 Introduction

One appealing feature of ML-family languages is the ability to define fundamental data structures — pairs, lists, streams, and so on — in user code. For example, although *lazy* computations are supported as a built-in construct in OCaml, it is also possible to implement them as a library.

Laziness in variants The following data type can serve as a basis for lazy computations[1]:

```
type 'a lazy_cell = | Thunk of (unit → 'a)
                    | Value of 'a
                    | Exn of exn
```

The constructors of a `lazy_cell` value represent the three possible states of a lazy computation: it may be an unevaluated thunk, a fully-evaluated value, or a computation whose evaluation terminated with an exception. Since the state of a lazy computation may change over time, lazy values are represented as mutable references that hold `lazy_cell` values:

```
type 'a lzy = 'a lazy_cell ref
```

Finally, there are two operations: `delay` creates a thunk from a function, while `force` either forces a thunk and caches the result, or returns the value or exception cached by a previous call.

```
let delay f = ref (Thunk f)
```

```
let force r = match !r with
  | Thunk f →
     (match f () with
      | v → r := Value v; v
      | exception e → r := Exn e; raise e)
  | Value v → v
  | Exn e → raise e
```

[1] The code in this paper uses the proposed *modular implicits* extension to OCaml [20]. The modular implicits compiler can be installed as the OPAM switch `4.02.0+modular-implicits`.

Submitted to:
ML 2017

Laziness invariants The characterising feature of lazy computations is that each computation is run only once, although the result may be read many times. The `delay` and `force` functions enforce this property. For simplicity, we assume that lazy computations do not invoke themselves recursively. (More sophisticated implementations of laziness generally include a fourth state `In_progress` to guard against this occurrence).

Concealing the representation type of `lzy` behind a module interface ensures that other parts of the program cannot violate this invariant:

```
module Lzy :
sig
  type 'a t
  val delay : (unit → 'a) → 'a t
  val force : 'a t → 'a
end
```

```
module Lzy =
struct
  type 'a t = 'a lzy
  let delay = ... (* as above *)
  let force = ... (* as above *)
end
```

Laziness invariance This simple implementation has the same behaviour as the built-in `lazy` type. However, there is one notable difference: unlike the built-in type, our `Lzy.t` is not *covariant*.

Covariant types are parameterised types that preserve the subtyping relationships between parameters. For example, if `u` is a subtype of `v` then, because `option` is covariant, `u option` is a subtype of `v option`. In OCaml, types may be marked as covariant by adding a `+` before a type parameter. For example, here is a definition of the covariant type `option`:

```
type +'a option = None | Some of 'a
```

Not every type can be marked as covariant in this way. For example, it would not be safe to allow `ref`, the type of mutable cells, to behave covariantly. If `ref` were covariant then a program could coerce a value of type `u ref` to `v ref`, and then store a value of another subtype of `v`, unrelated to `u`, in the cell. OCaml therefore prohibits covariance for type parameters that appear under `ref`, as in the definition of `lzy`.

The lack of covariance in our `Lzy.t` has two significant consequences for programmers. First, computations constructed using the built-in type can be coerced, while computations constructed using our `Lzy.t` cannot. Here is a coercion that eliminates a method `m` from the type of a built-in lazy computation:

```
# let o = object method m = () end;;
val o : < m : unit > = <obj>
# (lazy o :> < > Lazy.t);;
- : <  > Lazy.t = <lazy>
```

An attempt to similarly coerce our `Lzy.t` fails:

```
# (Lzy.delay (fun () → o) :> < > Lzy.t);;
Characters 0-40:
  (Lzy.delay (fun () → o) :> < > Lzy.t);;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Type < m : unit > Lzy.t is not a subtype of <  > Lzy.t
       The second object type has no method m
```

Second, `let`-bound computations constructed using the built-in `lazy` receive polymorphic types, following the *relaxed value restriction*, which generalizes type variables that appear only in covariant positions [12]:

```
# let f = Lazy.from_fun (fun () → []);;
val f : 'a list Lazy.t = <lazy>
```

In contrast, types built using our `Lzy.t` are ungeneralized, as the leading underscore on the type variable `'_a` indicates:[2]

```
# Lzy.delay (fun () → []);;
- : '_a list Lzy.t = <abstr>
```

The interface to `Lzy.t` only exposes read operations, and so it would be safe for the type to be treated as covariant in its parameter. However, the assignment of variance considers only the use of the parameter in the definition of `Lzy.t`, not the broader module interface. Since the type parameter is passed to the the invariant type `ref` of mutable cells, the type `Lzy.t` is also considered invariant.

These shortcomings in the `Lzy` interface can be overcome with more flexible treatment of subtyping and variance[3]. In particular, making subtypes *first-class* makes it possible to tie together the type definition and the functions exposed in the interface in the consideration of variance assignment, and so make `Lzy.t` covariant.

Variants of first-class subtypes may be found in advanced type systems in the research literature, such as Cretin and Rémy's F_ι [9]. Our contribution here is to show that first-class subtypes can be *encoded* using the features of an existing real-world functional programming language.

First-class subtypes can be defined using a binary type constructor:

```
type (-'a, +'b) sub
```

that has a single constructor:

```
val refl : ('a, 'a) sub
```

and an operation that turns a `sub` value into a function:

```
val (>:) : 'a → ('a, 'b) sub → 'b
```

These three elements, considered in more detail in Section 2 are sufficient to define a covariant variant of `Lzy`. Section 2 adds an additional constuctor, `lift` that, together with the three elements above, suffices as a basis to define a range of useful subtyping operations.

Here is a second, covariant interface to lazy computations:

[2]See `https://caml.inria.fr/pub/docs/manual-ocaml/polymorphism.html`

[3]An alternative solution is to switch to a higher-order representation of lazy computations:

```
type 'a t = unit → 'a
let delay f =
  let r = ref (Thunk f) in
  fun () → match !r with
    | Thunk f →
       (match f () with
        | v → r := Value v; v
        | exception e → r := Exn e; raise e)
    | Value v → v
    | Exn e → raise e
let force f = f ()
```

With this representation the type `t` is covariant in its parameter, which occurs only in a positive position. However, the higher-order representation makes lazy values more difficult to inspect.

```
module CovLzy :
sig
  type +'a t
  val delay : (unit → 'a) → 'a t
  val force : 'a t → 'a
end
```

The implementation of CovLzy can be constructed from the combination of Lzy and first-class subtypes. Here is the definition of CovLzy.t

```
type +'b t = L : 'a Lzy.t * ('a, 'b) sub → 'b t
```

That is, a value of type `'a CovLzy.t` is a pair of a lazy computation of type `'a Lzy.t` and a value of type `('a, 'b) sub` that supports coercions from `'a` to `'b`.

Now `CovLzy.delay` builds on `Lzy.delay`, pairing a lazy computation with a `sub` value[4]:

```
let delay f = L (Lzy.delay f, refl)
```

Finally, the definition of `CovLzy.force` calls `Lzy.force` and applies a coercion to the value returned:

```
let force (L (sub, l)) =
  match Lzy.force l with
  | v → (v >: sub)
  | exception e → raise e
```

These additions to make `CovLzy` covariant bring its behaviour closer to the behaviour of the built-in `lazy`. For example, `let`-bound values built by `CovLzy.delay` receive polymorphic types:

```
# CovLzy.delay (fun () → []);;
- : 'a list CovLzy.t = <abstr>
```

2 First-class subtypes defined

2.1 Subtypes *à la* Liskov & Wing

The first ingredient in a representation of subtyping proofs is a definition of subtyping. Here is Liskov and Wing's characterisation [15]:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

For instance, properties of a record type r should also hold for a widening of r, since the extra fields can be ignored. And dually, properties of a variant type v should also hold for a narrowing of v: any property that holds for all constructors also holds for a subset of constructors.

[4] A reviewer observes that the permission-oriented language Mezzo [4] uses a related approach. In Mezzo, a witness that `s` is a subtype of `t` can be encoded as a permission of the form `id @ a → b`, which can be read as "the identity function has type `a → b`", and these witnesses are also used in Mezzo to make the `lazy` type covariant: `http://protz.github.io/mezzo/code_samples/lazy.mz.html`.

2.2 Subtypes *à la* Curry & Howard

The Curry-Howard correspondence turns Liskov and Wing's characterisation of subtyping into an executable program.

With a propositions-as-types perspective [18], a property provable about objects of type T is represented as a type $\phi(T)$ involving T, and a proof of that property is a term of that type [5]. Liskov and Wing's proposition that S is a subtype of T corresponds to the following (poly)type:

$$\forall \phi . \phi(T) \to \phi(S)$$

Two points deserve note.

First, although Liskov and Wing's characterisation is couched in terms of *objects* x and y, it is really about their *types* T and S. To see this, consider that, in the characterisation, it is sufficient to know y's type to know that $\Phi(y)$ holds. Since the characterisation only involves properties about all the objects of a type, not properties of individual objects, there is no need for dependent types in the type corresponding to the subtyping proposition.

Second, a "property about objects" is a context that *consumes* an object. For example, consider the property "for every object of type T there is an object of type R", which can reasonably be described as a property about objects of type T, but not as a property about objects of type R. In a propositions-as-types setting, a proof of this property is a context that consumes an object of type T and produces an object of type R. Since the properties of interest are consumers of objects, ϕ ranges over *negative* contexts.

2.3 Contexts and variance

```
module type POS = sig type +'a t end
module type NEG = sig type -'a t end

module Id = struct type 'a t = 'a end

module Compose_PN(F:NEG)(G:POS) = struct type 'a t = 'a F.t G.t end
module Compose_PP(F:POS)(G:POS) = struct type 'a t = 'a F.t G.t end
```

Figure 1: Positive and negative contexts

Figure 1 defines OCaml signatures, `POS` and `NEG`, of positive and negative type contexts. The `-` preceding the type parameter `'a` indicates that 'a can only appear in negative (contravariant) positions in instantiations of the signature. The `Id` module and `Compose` functors represent the identity context and the composition of two contexts. Each composition of variance in the argument contexts requires a separate `Compose` (but see §4 for a generalization).

2.4 Encoding subtypes

Figure 2 defines an interface to subtype witnesses. A value of type `(s, t) sub` serves as evidence that `s` is a subtype of `t`. There are two ways to construct such evidence. First, `refl` represents the fact that every type is a subtype of itself. Second, `lift` represents the fact that subtyping lifts through covariant

[5] At least, in total languages. In OCaml the waters are muddied by side-effects and nontermination.

```
type (-'a, +'b) sub
val refl : ('a, 'a) sub
val lift: {P:POS} → ('a,'b) sub → ('a P.t,'b P.t) sub
val (>:) : 'a → ('a, 'b) sub → 'b
```

Figure 2: First-class subtypes: minimal interface

contexts, which are passed as implicit arguments [20]. (Lifts through contravariant contexts are defined below in terms of the minimal interface.) The single destructor, `>:`, which mimics OCaml's built-in coercion operator `:>`, supports converting a value of type `s` to a supertype `t`.

This small interface suffices as a basis for many useful subtyping-related functions. For example, the transitivity of subtyping is represented by a function of the following type:

```
val trans : ('a,'b) sub → ('b,'c) sub → ('a,'c) sub
```

and may be defined as follows:

```
let trans (type a b c) (x : (a,b) sub) (y : (b,c) sub) =
  let module M = struct type +'d t = (a,'d) sub end
  in x >: lift {M} y
```

Here the application `lift M y` builds a value of type `(b M.t, c M.t) sub` — that is to say, a value of type `((a, b) sub, (a, c) sub) sub` — which is used to coerce `x` from type `(a, b) sub` to type `(a, c) sub`.

Similarly, a function that lifts subtyping witnesses through negative contexts

```
val lift₋ : {N:NEG} →('a, 'b) sub → ('b N.t, 'a N.t) sub
```

may also be defined by supplying a suitable implementation of `POS` to `lift`:

```
let lift₋ (type a b) {N:NEG} (x: (a,b) sub) : (b N.t,a N.t) sub =
  let module M = struct type +'b t = ('b N.t, a N.t) sub end in
  refl >: lift {M} x
```

Using the variance of `sub`, `refl` can be used to define a witness for any subtyping fact that holds in the typing environment. For example, in OCaml the object type `< m:int >`, with one method `m`, is a subtype of the type `< >` of objects with no methods. This fact can be turned into a `sub` value by coercing `refl`, either by lowering the contravariant parameter:

```
(refl : (< >, < >) sub :> (<m:int>, < >) sub)
```

or by raising the covariant parameter:

```
(refl : (<m:int>, <m:int>) sub :> (<m:int>, < >) sub)
```

The resulting value can be passed freely through abstraction boundaries that conceal the types involved, eventually being used to coerce a value of type `<m:int>` to its supertype `< >`.

The generality of the interface in Figure 2 places constraints on the implementation. Most notably, since `lift` can transport subtyping evidence through *any* positive context, coercion must pass values through unexamined. For example, `lift` might be used to build a value of type `(s list, t list) sub` from a value of type `(s, t) sub`:

```
let l : (s list, t list) sub = lift {List} s_sub_t
```

but applying `l` cannot involve list traversal, since the subtyping interface says nothing about list structure. A polymorphic interface thus ensures an efficient implementation.

3 Implementations of subtyping

3.1 First-class subtypes via contexts

```
type (-'a, +'b) sub = {N:NEG} → ('b N.t → 'a N.t)
let refl {N:NEG} x = x
let lift {P:POS} s {Q:NEG} x = s {Compose_PN(Q)(P)} x
let (>:) (type b) x f =
  let module M = struct type -'a t = 'a → b end in
  f {M} id x
```

Figure 3: First-class subtypes via negative contexts

```
type (-'a, +'b) sub = {P:POS} → ('a P.t → 'b P.t)
let refl {P:POS} x = x
let lift {P:POS} s {Q:POS} x = s {Compose_PP(P)(Q)} x
let (>:) x f = f {Id} x
```

Figure 4: First-class subtypes via positive contexts

Figure 3 gives an implementation of Figure 2 based on negative contexts that directly follows Liskov & Wing's definition[6]. A value of type `(s, t) sub` is a proof that `t` can be replaced with `s` in any negative context; operationally it must be the identity, as discussed above, and so the two constructors `lift` and `refl` both correspond to the identity function. Figure 4 gives a similar but simpler implementation, based on positive contexts. Apart from the variance annotations, these definitions mirror the standard Leibniz encoding of type equality [21].

3.2 First-class subtypes as an inductive type

Consider an ordinary inductive type, say the Peano natural numbers:

```
type nat = Zero | Suc nat
```

We can write its constructors in the form of a module signature as follows:

```
module type NAT = sig
  type t
  val zero : t
  val suc : t → t
end
```

What it means for the type `nat` to be inductive is that it is an *initial algebra* for this signature: first, it implements the signature by providing `Zero` and `Suc`, and secondly, for any other implementation `M`, we have a function mapping `nat` to `M.t` that maps `Zero` to `M.zero` and `Suc` to `M.suc`:

```
let rec primrec = function
| Zero → M.zero
| Suc n → M.suc (primrec n)
```

[6]Edward Kmett has used this approach in the `magpie` library [14], as we discovered after writing this note.

In defining the type `nat`, we made use of OCaml's built-in support for inductive types. Lacking this, we could have used the initial algebra definition directly, and defined the type `nat` as follows:

```
type nat = {M : NAT} → M.t
```

This corresponds to the Church encoding of the natural numbers [8], in which a natural number is anything that can produce a `M.t` from `M.zero : M.t` and `M.suc : M.t → M.t`. Here and elsewhere we're using the modular implicits extension to OCaml [20] — not for implicit instantiation of arguments, but because modular implicits support higher-kinded quantification with propagation of variance information. Other approaches to higher-kinded polymorphism could perhaps be used instead [22].

This approach to inductive types also makes sense for GADTs, such as the equality GADT defined below [3, 7, 19]:

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

This uses OCaml's GADT syntax, but we can do without it by building a Church encoding of equality [2], using the same technique as before:

```
module type EQ = sig
  type ('a, 'b) t
  val refl : ('a, 'a) t
end
type ('a, 'b) eq = {E : EQ} → ('a, 'b) E.t
```

We can use this encoding to implement the standard operations on the equality GADT by providing a suitable implementation of the `EQ` interface. For instance, we can implement the coercion function

```
val cast : ('a, 'b) eq → 'a → 'b
```

by supplying an implementation of `EQ` using function types:

```
let cast (f : ('a, 'b) eq) x =
  let module L = struct
    type ('a, 'b) t = 'a → 'b
    let refl = fun x → x
  end in
  f {L} x
```

If we modify the signature `EQ` by adding co- and contra-variance markers to the parameters of the type `t`, then we get the Church encoding of first-class subtypes, as shown in Figure 5, from which we can implement the `refl`, `lift` and `(:>)` functions.

3.3 Converting between encodings

Despite the different starting points, the three implementations are interdefinable. In fact, given *any* two implementations `A` and `B` of the subtyping interface, a subtyping witness of type `('a,'b) A.t` can be converted to a witness of type `('a,'b) B.t`.
The `SUB` module type contains the four elements of Figure 2 (`t`, `refl`, `lift`, `>:`):

```
module type SUB =
sig
  type ('a, 'b) t
  val refl : ('a, 'a) t
  val lift : {P:POS} → ('a, 'b) t → ('a P.t, 'b P.t) t
  val (>:) : 'a → ('a, 'b) sub → 'b
end
```

```
module type SUB =
sig
  type (-'a, +'b) t
  val refl : ('a, 'a) t
end

type (-'a, +'b) sub = {S:SUB} → ('a,'b) S.t

let refl {S:SUB} = S.refl

let lift {P:POS} (f : ('a, 'b) sub) =
  let module L = struct
    type ('a,'b) t = ('a P.t,'b P.t) sub
    let refl = refl
  end in f {L}

let (>:) x (f : ('a, 'b) sub) =
  let module L = struct
    type ('a, 'b) t = 'a → 'b
    let refl = fun x → x
  end in f {L} x
```

Figure 5: First-class subtypes as an initial algebra

The function `conv` takes two implementations of `SUB`, `A` and `B`, and converts a value in `A.t` to a value of `B.t`:

```
val conv : {A:SUB} → {B: SUB} → ('a, 'b) A.t → ('a, 'b) B.t
```

As often, implementing `conv` is a matter of finding a suitable implementation of `POS` to pass to `lift`:

```
let conv (type a b) {A:SUB} {B:SUB} (x : (a,b) A.t) =
  let module M = struct type 'a t = (a, 'a) B.t end in
  A.(>:) B.refl (A.lift {M} x)
```

The function `conv` works as follows. The value `x` is a proof that `a` $\leq_A$ `b` — that is, that `a` is an `A`-subtype of `b`. The type `M.t` represents the positive context `a` $\leq_B$ `-`. The call to `lift` then lifts the proof `a` $\leq_A$ `b` through the context `M.t` to produce a proof `(a` $\leq_B$ `a)` $\leq_A$ `(a` $\leq_B$ `b)`. Finally, this proof can be used to coerce a proof of the reflexivity of `B`-subtyping (at type `a`) `a` $\leq_B$ `a` to a proof that `a` $\leq_B$ `b`. That is, from a proof `a` $\leq_A$ `b`, the operations of `A.sub` and `B.sub` produce a proof `a` $\leq_B$ `b`.

4 First-class subtypes: further examples

4.1 Arrays and rows

Here is a function that prints arrays by calling the `name` method of each element:

```
let print_array = Array.iter (fun o → print o#name)
```

To call `name`, `print_array` does not need to know the full element type: it needs only to know that there is a method `name` returning `string`. OCaml gives `print_array` a row type, indicating that the element type may have other methods:

```
val print_array : <name: string; ..> array → unit
```

But rows are sometimes too inflexible. Given two arrays `a`, `b`, of different element types

```
val a : < m : int; name : string > array
val b : < n : bool; name : string > array
```

unification will fail:

```
List.iter print_array [a; b];;
                          ^
Error: This expression has type < n : bool; name : string > array
       but an expression was expected of type
         < m : int; name : string > array
       The second object type has no method n
```

Using first-class subtypes it is possible to combine iterations over arrays whose element types belong to the same subtyping hierarchy.

```
type +'a arr = Arr : 'x array * ('x, 'a) sub → 'a arr
let aiter f (Arr (a,sub)) = Array.iter (fun s → f (s >: sub)) a
List.iter
  (aiter (fun o → print o#name)) [Arr (a,refl); Arr (b,refl)]
```

4.2 Selective abstraction

A third class of examples arises from selective abstraction, where an abstract type comes with a proof of a property about that type. For example, here is a module that exports a type `t` along with a proof that `t` is a subtype of `int`:

```
module M:
  sig type t
  val t_sub_int : (t,int) sub
  (*...*)
end
```

Outside the module, values of type `t` can be coerced to `int`, but not vice versa. This approach supports a style similar to refinement types, in which abstraction boundaries distinguish values of a type for which some additional predicate has been shown to hold.

These are known as *partially abstract types* [6], and are available as a language feature in OCaml [13] and Moby [11]. However, implementing these via first-class subtypes allows more flexibility: for example, they allow some of the methods of an object type to be hidden from the exposed interface, and also support the dual of private types (called *invisible types* [16]), and *zero cost-coercions* [5], where coercions in both directions are available, but actual type equality is not exposed.

4.3 Bounded quantification

Dual to abstraction, combining first-class subtypes with OCaml's first-class polymorphism encodes bounded quantification. For example, the type $\forall \alpha \leq t.\ \alpha \to t$ might be written as follows:

```
type s = { f: 'a. ('a, t) sub → 'a → t }
```

4.4 Proofs of variance

Finally, first-class subtypes can express proofs of variance. For example, the covariance of `list` can be represented by a value of the following type:

$$\texttt{('a,'b) sub} \to \texttt{('a list, 'b list) sub}$$

4.5 Unsoundness in Java

Amin and Tate [1] present an encoding of first-class subtypes in Java, which they use to demonstrate a soundness bug. They use Java's bounded quantification to define a type `Constrain<A, B>` which is well-formed only when $\mathtt{A} \leq \mathtt{B}$. Then, the type $\exists \mathtt{X} \geq \mathtt{T}.\ \mathtt{Constrain\langle U,X\rangle}$ corresponds to a subtyping witness `(U,T) sub`: if this type is inhabited, then some $\mathtt{X} \geq \mathtt{T}$ exists making `Constrain<U, X>` well-formed, giving $\mathtt{U} \leq \mathtt{X} \leq \mathtt{T}$. Unfortunately, in Java (unlike OCaml), the value `null` inhabits every reference type, giving an invalid subtyping witness that allows any type to be coerced to any other.

5 Discussion

The encodings given here are useful for exploratory work, for demonstrating soundness, and for showcasing OCaml's expressivity. However, direct language support would make first-class subtypes more usable. Scherer and Rémy [16] discuss design issues and related work (e.g. [10, 17]).

The encodings suffer from some awkwardness, since contexts must be applied explicitly, unlike the equalities revealed by pattern matching with GADTs, which the type checker applies implicitly.

Our encodings share another issue with similar encodings of GADTs [21]: they lack inversion principles. Given `('a, 'b) sub`, our encodings can be used to derive `('a list, 'b list) sub`, from the covariance of the `list` type constructor. However, going the other direction, from `('a list, 'b list) sub` to `('a, 'b) sub`, is equally valid but not expressible with our encodings.

With language support for subtype witnesses, coercions would still be explicit, but constraints in scope could be implicitly lifted through contexts, and inversion principles could be applied.

Acknowledgements We thank François Pottier, Leo White, the ML 2017 reviewers, and the ML & OCaml 2017 post-proceedings reviewers for helpful comments.

References

[1] Nada Amin & Ross Tate (2016): *Java and Scala's type systems are unsound: the existential crisis of null pointers*. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pp. 838–848, doi:10.1145/2983990.2984004.

[2] Robert Atkey (2012): *Relational Parametricity for Higher Kinds*. In Patrick Cégielski & Arnaud Durand, editors: *Computer Science Logic (CSL'12)*, *LIPIcs* 16, doi:10.4230/LIPIcs.CSL.2012.46.

[3] Arthur I. Baars & S. Doaitse Swierstra (2002): *Typing Dynamic Typing*. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, ACM, doi:10.1145/581478.581494.

[4] Thibaut Balabonski, François Pottier & Jonathan Protzenko (2016): *The Design and Formalization of Mezzo, a Permission-Based Programming Language*. *ACM Trans. Program. Lang. Syst.* 38(4), pp. 14:1–14:94, doi:10.1145/2837022.

[5] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones & Stephanie Weirich (2014): *Safe Zero-cost Coercions for Haskell*. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, ACM, doi:10.1145/2628136.2628141.

[6] Luca Cardelli & Peter Wegner (1985): *On Understanding Types, Data Abstraction, and Polymorphism*. *ACM Comput. Surv.* 17(4), pp. 471–523, doi:10.1145/6041.6042.

[7] James Cheney & Ralf Hinze (2003): *First-Class Phantom Types*. Technical Report, Cornell University.

[8] Alonzo Church (1940): *A Formulation of the Simple Theory of Types*. *The Journal of Symbolic Logic* 5(2), doi:10.2307/2266170. Available at `http://www.jstor.org/stable/2266170`.

[9] Julien Cretin & Didier Rémy (2012): *On the Power of Coercion Abstraction*. In: *POPL 2012: 39th ACM SIGPLAN-SIGACT Symposium on Principle Of Programming Languages*, Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, ACM, Philadelphia, United States, doi:10.1145/2103656.2103699.

[10] Burak Emir, Andrew Kennedy, Claudio Russo & Dachuan Yu (2006): *Variance and Generalized Constraints for C# Generics*. In: *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, Springer-Verlag, doi:10.1007/11785477_18.

[11] Kathleen Fisher & John Reppy (2000): *Extending Moby with Inheritance-Based Subtyping*. In Elisa Bertino, editor: *ECOOP 2000 — Object-Oriented Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 83–107, doi:10.1007/3-540-45102-1_5.

[12] Jacques Garrigue (2004): *Functional and Logic Programming: 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004. Proceedings*, chapter Relaxing the Value Restriction. Springer Berlin Heidelberg.

[13] Jacques Garrigue (2006): *Private Row Types: Abstracting the Unnamed*. In Naoki Kobayashi, editor: *Programming Languages and Systems: 4th Asian Symposium, APLAS 2006*, Springer Berlin Heidelberg, doi:10.1007/11924661_3.

[14] Edward Kmett (2010): *Magpie*. `https://github.com/ekmett/magpie/`. See also `https://issues.scala-lang.org/browse/SI-4040`.

[15] Barbara H. Liskov & Jeannette M. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Trans. Program. Lang. Syst.* 16(6), doi:10.1145/197320.197383.

[16] Gabriel Scherer & Didier Rémy (2013): *GADTs Meet Subtyping*. In Matthias Felleisen & Philippa Gardner, editors: *22nd European Symposium on Programming, ESOP 2013*, *Lecture Notes in Computer Science* 7792, Springer, doi:10.1007/978-3-642-37036-6_30.

[17] Benoit Vaugon (2016): *Subtyping by Constraint Saturation, Theory and Implementation*. Theses, Université Paris-Saclay. Available at `https://pastel.archives-ouvertes.fr/tel-01356695`.

[18] Philip Wadler (2015): *Propositions As Types*. *Commun. ACM* 58(12), doi:10.1145/2699407.

[19] Stephanie Weirich (2004): *Functional Pearl: type-safe cast*. *Journal of Functional Programming* 14, doi:10.1017/S0956796804005179.

[20] Leo White, Frédéric Bour & Jeremy Yallop (2015): *Modular Implicits*. ACM Workshop on ML 2014 post-proceedings, doi:10.4204/EPTCS.198.2.

[21] Jeremy Yallop & Oleg Kiselyov (2010): *First-class modules: hidden power and tantalizing promises*. ACM SIGPLAN Workshop on ML. Baltimore, Maryland, United States.

[22] Jeremy Yallop & Leo White (2014): *Lightweight Higher-Kinded Polymorphism*. In Michael Codish & Eijiro Sumii, editors: *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan. Proceedings*, doi:10.1007/978-3-319-07151-0_8.