GADT meet subtyping

Gabriel Scherer and Didier Rémy

INRIA, Rocquencourt*

Abstract. While generalized abstract datatypes (GADT) are now considered well-understood, adding them to a language with a notion of subtyping comes with a few surprises. What does it mean for a GADT parameter to be covariant? The answer turns out to be quite subtle. It involves fine-grained properties of the subtyping relation that raise interesting design questions. We allow variance annotations in GADT definitions, study their soundness, and present a sound and complete algorithm to check them. Our work may be applied to real-world ML-like languages with explicit subtyping such as OCaml, or to languages with general subtyping constraints.

1 Motivation

In languages that have a notion of subtyping, the interface of parametrized types usually specifies a *variance*. It defines the subtyping relation between two instances of a parametrized type from the subtyping relations that hold between their parameters. For example, the type α `list` of immutable lists is expected to be *covariant*: we wish σ `list` $\leq \sigma'$ `list` as soon as $\sigma \leq \sigma'$.

Variance is essential in languages whose programming idioms rely on subtyping, in particular object-oriented languages. Another reason to care about variance is its use in the *relaxed value restriction* [Gar04]: while a possibly-effectful expression, also called an *expansive expression*, cannot be soundly generalized in ML—unless some sophisticated enhancement of the type system keeps track of effectful expressions—it is always sound to generalize type variables that only appear in covariant positions, which may not classify mutable values. Therefore, it is important for extensions of type definitions, such as GADT, to support it as well through a clear and expressive definition of parameter covariance.

For example, consider the following GADT of well-typed expressions:

```
type +α expr =
  | Val : α → α expr
  | Int : int → int expr
  | Thunk : ∀β. β expr * (β → α) → α expr
  | Prod : ∀βγ. β expr * γ expr → (β * γ) expr
```

Is it safe to say that `expr` is covariant in its type parameter? It turns out that, using the subtyping relation of the OCaml type system, the answer is "yes".

* Part of this work has been done at IRILL.

But, surprisingly to us, in a type system with a top type $\top$, the answer would be "no".

The aim of this article is to present a sound and complete criterion to check soundness of parameter variance annotations, for use in a type-checker. We also discuss the apparent fragility of this criterion with respect to changes to the subtyping relation (*e.g.* the presence or absence of a top type, private types, *etc.*), and a different, more robust way to combine GADT and subtyping.

Note Due to space restriction, the present article is only a short version from which many details have been omitted. All proofs of results presented in this version along with auxiliary results, as well as further discussions, can be found in the longer version available online [SR].

Examples

Let us first explain why it is reasonable to say that α `expr` is covariant. Informally, if we are able to coerce a value of type α into one of type α' (we write (`v` :> α') to explicitly cast a value `v` of type α to a value of type α'), then we are also able to transform a value of type α `expr` into one of type α' `expr`. Here is some pseudo-code[1] for the coercion function:

```
let coerce : α expr → α' expr = function
  | Val (v : α) -> Val (v :> α')
  | Int n -> Int n
  | Thunk β (b : β expr) (f : β → α) ->
    Thunk β b (fun x -> (f x :> α'))
  | Prod β γ ((b, c) : β expr * γ expr) ->
    (* if β * γ ≤ α', then α' is of the form
       β' * γ' with β ≤ β' and γ ≤ γ' *)
    Prod β' γ' ((b :> β' expr), (c :> γ' expr))
```

In the `Prod` case, we make an informal use of something we know about the OCaml type system: the supertypes of a tuple are all tuples. By entering the branch, we gain the knowledge that α must be equal to some type of the form $\beta * \gamma$. So from $\alpha \leq \alpha'$ we know that $\beta * \gamma \leq \alpha'$. Therefore, α' must itself be a pair of the form $\beta' * \gamma'$. By covariance of the product, we deduce that $\beta \leq \beta'$ and $\gamma \leq \gamma'$. This allows to conclude by casting at types β' `expr` and γ' `expr`, recursively.

Similarly, in the `Int` case, we know that α must be an `int` and therefore an `int expr` is returned. This is because we know that, in OCaml, no type is above `int`: if `int` $\leq \tau$, then τ must be `int`.

What we use in both cases is reasoning of the form[2]: "if $T[\overline{\beta}] \leq \alpha'$, then I know that α' is of the form $T[\overline{\beta}']$ for some $\overline{\beta}'$". We call this an *upward closure*

[1] The variables β' and γ' of the `Prod` case are never really defined, only justified at the meta-level, making this code only an informal sketch.

[2] We write $T[\overline{\beta}]$ for a type expression T that may contain free occurrences of variables $\overline{\beta}$ and $T[\overline{\sigma}]$ for the simultaneous substitution of $\overline{\sigma}$ for $\overline{\beta}$ in T.

property: when we "go up" from a $T[\overline{\beta}]$, we only find types that also have the structure of T. Similarly, for contravariant parameters, we would need a *downward closure* property: T is downward-closed if $T[\overline{\beta}] \geq \alpha'$ entails that α' is of the form $T[\overline{\beta}']$.

Before studying a more troubling example, we define the classic equality type (α, β) `eq`, and the corresponding casting function `cast` : $\forall\alpha\beta.(\alpha, \beta)$ `eq` $\to \alpha \to \beta$:

```
type (α, β) eq =                 let cast r =
  | Refl : ∀γ. (γ, γ) eq           match r with Refl -> (fun x -> x)
```

Notice that it would be unsound[3] to define `eq` as covariant, even in only one parameter. For example, if we had `type` $(+\alpha, =\beta)$ `eq`, from any $\sigma \leq \tau$ we could subtype (σ, σ) `eq` into (τ, σ) `eq`, allowing to cast any value of type τ back into one of type σ, which is unsound in general.

As a counter-example, the following declaration is incorrect: the type α `t` cannot be declared covariant.

```
type +α t =
  | K : < m : int > → < m : int > t
let v = (K (object method m = 1 end) :> < > t)
```

This declaration uses the OCaml object type `< m : int >`, which qualifies objects having a method `m` returning an integer. It is a subtype of object types with fewer methods, in this case the empty object type `< >`, so the alleged covariance of `t`, if accepted by the compiler, would allow us to cast a value of type `< m : int > t` into one of type `< > t`. However, from such a value, we could wrongly deduce an equality witness `(< >, <m : int>) eq` that allows to cast any empty object of type `< >` into an object of type `< m : int >`, but this is unsound, of course!

```
let get_eq : α t → (α, < m : int >) eq = function
  | K _ -> Refl      (* locally α = < m : int > *)
let wrong : < > -> < m : int > =
  let eq : (< >, < m : int >) eq = get_eq v in
  cast eq
```

It is possible to reproduce this example using a different feature of the OCaml type system named *private type abbreviation*[4]: a module using a type `type t =` τ *internally* may describe its interface as `type t = private` τ. This is a compromise between a type abbreviation and an abstract type: it is possible to cast a value of type `t` into one of type τ, but not, conversely, to construct a value of type `t` from one of type τ. In other words, `t` is a strict subtype of τ: we have `t` $\leq \tau$ but not `t` $\geq \tau$. Take for example `type file_descr = private int`: this semi-abstraction is useful to enforce invariants by restricting the construction of values of type `file_descr`, while allowing users to conveniently and efficiently destruct them for inspection at type `int`. Using an unsound but quite innocent-looking covariant GADT datatype, one is able to construct a function to cast any

[3] This counterexample is due to Jeremy Yallop.
[4] This counterexample is due to Jacques Garrigue.

integer into a `file_descr`, which defeats the purpose of this abstraction—see the extended version of this article for the full example.

The difference between the former, correct `Prod` case and those two latter situations with unsound variance is the notion of upward closure. The types $\alpha * \beta$ and `int` used in the correct example were upward-closed. On the contrary, the private type `file_descr` has a distinct supertype `int`, and similarly the object type `< m:int >` has a supertype `< >` with a different structure (no method `m`).

In this article, we formally show that these notions of upward and downward-closure are the key to a sound variance check for GADT. We start from the formal development of Simonet and Pottier [SP07], which provides a general soundness proof for a language with subtyping and a very general notion of GADT expressing arbitrary constraints—rather than only type equalities. By specializing their correctness criterion, we can express it in terms of syntactic checks for closure and variance, that are simple to implement in a type-checker.

The problem of non-monotonicity

There is a problem with those upward or downward closure assumptions: while they hold in core ML, with strong inversion theorems, they are non-monotonic properties: they are not necessarily preserved by extensions of the subtyping lattice. For example, OCaml has a concept of *private types*: a type specified by `type t = private` τ is a new semi-abstract type smaller than τ (`t` $\leq \tau$ but `t` $\not\geq \tau$), that can be defined a posteriori for any type. Hence, no type is downward-closed *forever*. That is, for any type τ, a new, strictly smaller type may always be defined in the future.

This means that closure properties of the OCaml type system are relatively weak: no type is downward-closed[5] (so instantiated GADT parameters cannot be contravariant), and arrow types are not upward-closed as their domain should be downward-closed. Only purely positive algebraic datatypes are upward-closed. The subset of GADT declarations that can be declared covariant today is small, yet, we think, large enough to capture a lot of useful examples, such as α `expr` above.

Giving back the freedom of subtyping

It is disturbing that the type system should rely on non-monotonic properties: if we adopt the correctness criterion above, we must be careful in the future not to enrich the subtyping relation too much.

Consider `private` types for example: one could imagine a symmetric concept of a type that would be strictly *above* a given type τ; we will name those types `invisible` types (they can be constructed, but not observed). Invisible types

[5] Except types that are only defined privately in a module and not exported: they exist in a "closed world" and we can check, for example, that they are never used in a `private` type definition.

and GADT covariance seem to be working against each other: if the designer adds one, adding the other later will be difficult.

A solution to this tension is to allow the user to *locally* guarantee negative properties about subtyping (what is *not* a subtype), at the cost of selectively abandoning the corresponding flexibility. Just as object-oriented languages have `final` classes that cannot be extended any more, we would like to be able to define some types as `public` (respectively `visible`), that cannot later be made `private` (resp. `invisible`). Such declarations would be rejected if the defining type already has subtypes (*e.g.* an object type), and would forbid further declarations of types below (resp. above) the defined type, effectively guaranteeing downward (resp. upward) closure. Finally, upward or downward closure is a semantic aspect of a type that we must have the freedom to publish through an interface: abstract types could optionally be declared `public` or `visible`.

Another approach: subtyping constraints

Getting fine variance properties out of GADT is difficult because they correspond to type equalities which, to a first approximation, use their two operands both positively and negatively. One way to get an easy variance check is to encourage users to *change* their definitions into different ones that are easier to check. For example, consider the following redefinition of α `expr` (in a speculative extension of OCaml with subtyping constraints):

```
type +α expr =
| Val : ∀α.α → α expr
| Int : ∀α[α≥int].int → α expr
| Thunk : ∀β. β expr * (β → α) → α expr
| Prod : ∀αβγ[α≥β * γ]. (β expr * γ expr) → α expr
```

It is now quite easy to check that this definition is covariant, since all type equalities $\alpha = T_i[\overline{\beta}]$ have been replaced by inequalities $\alpha \geq T_i[\overline{\beta}]$ which are preserved when replacing α by a subtype $\alpha' \geq \alpha$—we explain this more formally in §4.2. This variation on GADT, using subtyping instead of equality constraints, has been studied by Emir *et al* [EKRY06] in the context of the `C♯` programming language.

But isn't such a type definition less useful than the previous one, which had a stronger constraint? We will discuss this choice in more detail in §4.2.

Related work

Simonet and Pottier [SP07] have studied GADT in a general framework HMG(X), inspired by HM(X). They were interested in typing inference using constraints, so considered GADT with arbitrary constraints rather than type equalities, and considered the case of subtyping with applications to information flow security in mind. We instantiate their general framework, which allows us to reuse their dynamic semantics and syntactic proofs of soundness, and concentrate only on

the static semantics, proving that we meet the requirements they impose on the parametrized models.

Their soundness criterion is formulated in very general terms as a constraint entailment problem. In contrast, our specialized study of the case of equality and subtyping led to a refined, more syntactic, criterion. This provides a more practically implementable check for type definitions, and reveals the design issues surrounding v-closed constructors that were not apparent in their work.

Emir, Kennedy, Russo and Yu [EKRY06] studied the soundness of an object-oriented calculus with subtyping constraints on classes and methods. Previous work [KR05] had established the correspondence between equality constraints on methods in an object-oriented style and GADT constraints on type constructors in functional style. Through this correspondence, their system matches our presentation of GADT with subtyping constraints and easier variance assignment, mentioned in the introduction (§1) and detailed in §4.2. They do not encounter the more delicate notion of upward and downward closure.

Those two approaches (subtyping constraints with easy variance check, stronger equality constraints with more delicate variance check) are complementary and have different convenience trade-offs. In their system with explicit-constraint definitions and implicit subtyping, the subtyping-constraint solution is the most convenient, while our ML setting provides incentives to study the other solution.

2 A formal setting

We define a core language for Algebraic Datatypes (ADT) and, later, Generalized Algebraic Datatypes (GADT), that is an instance of the parametrized HMG(X) system of Simonet and Pottier [SP07]. We refine their framework by using variances to define subtyping, but rely on their formal description for most of the system, in particular the static and dynamic semantics. We ultimately rely on their type soundness proof, by rigorously showing (in the next section) that their requirements on datatype definitions for this proof to hold are met in our extension with variances.

2.1 Atomic subtyping

Our type system defines a subtyping relation between ground types, parametrized by a reflexive transitive relation between base constant types (`int`, `bool`, etc.). Ground types consist of a set of base types `b`, function types $\tau_1 \to \tau_2$, product types $\tau_1 * \tau_2$, and a set of algebraic datatypes $\overline{\sigma}$ `t`. (We write $\overline{\sigma}$ for a sequence of types $(\sigma_i)_{i \in I}$.) We use prefix notation for datatype parameters, as is the usage in ML. Datatypes may be user-defined by toplevel declarations of the form:

```
type ᾱv t =
  | K₁ of τ¹[ᾱ]
  | ...
  | Kₙ of τⁿ[ᾱ]
```

$$\frac{}{\sigma \leq \sigma} \qquad \frac{\sigma_1 \leq \sigma_2 \qquad \sigma_2 \leq \sigma_3}{\sigma_1 \leq \sigma_3} \qquad \frac{\mathtt{b} \leqslant \mathtt{c}}{\mathtt{b} \leq \mathtt{c}} \qquad \frac{\sigma \geq \sigma' \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

$$\frac{\sigma \leq \sigma' \qquad \tau \leq \tau'}{\sigma * \tau \leq \sigma' * \tau'} \qquad \frac{\mathtt{type}\ \overline{v\alpha}\ \mathtt{t} \qquad \forall i,\ \sigma_i \prec_{v_i} \sigma'_i}{\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}}$$

Fig. 1. Subtyping relation

This is a disjoint sum: the constructors $\mathtt{K}_c$ represent all possible cases and each type $\tau^c[\overline{\alpha}]$ is the domain of the constructor $\mathtt{K}_c$. Applying it to an argument e of a corresponding ground type $\tau[\overline{\sigma}]$ constructs a term of type $\overline{\sigma}\ \mathtt{t}$. Values of this type are deconstructed using pattern matching clauses of the form $\mathtt{K}_c\ x \to e$, one for each constructor.

The sequence $\overline{v\alpha}$ is a binding list of type variables α_i along with their *variance annotation* v_i, which is a marker among the set $\{+, -, =, \bowtie\}$. We may associate a relation a relation $(\prec_v)$ between types to each variance v:

- $\prec_+$ is the *covariant* relation $(\leq)$;
- $\prec_-$ is the *contravariant* relation $(\geq)$, the symmetric of $(\leq)$;
- $\prec_=$ is the *invariant* relation $(=)$, defined as the intersection of $(\leq)$ and $(\geq)$;
- $\prec_\bowtie$, is the *irrelevant* relation $(\bowtie)$, the full relation such that $\sigma \bowtie \tau$ holds for all types σ and τ.

Given a reflexive transitive relation $(\leqslant)$ on base types, the subtyping relation on ground types $(\leq)$ is defined by the inference rules of Figure 1, which, in particular, give their meaning to the variance annotations $\overline{v\alpha}$. The judgment $\mathtt{type}\ \overline{v\alpha}\ \mathtt{t}$ simply means that the type constructor $\mathtt{t}$ has been previously defined with the variance annotation $\overline{v\alpha}$. Notice that the rules for arrow and product types can be subsumed by the rule for datatypes, if one consider them as special datatypes (with a specific dynamic semantics) of variance $(-, +)$ and $(+, +)$, respectively. For this reason, the following definitions will not explicitly detail the cases for arrows and products.

As usual in subtyping systems, we could reformulate our judgment in a syntax-directed way, to prove that it admits good inversion properties: if $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}$ and $\mathtt{type}\ \overline{v\alpha}\ \mathtt{t}$, then one can deduce that for each i, $\sigma_i \prec_{v_i} \sigma'_i$.

On the restriction of atomic subtyping Our typing relation reproduces a simplification that is present in the formulation of Simonet and Pottier: it is *atomic* in the sense that two non-constant type constructors in the subtyping relation are always identical. We are confident their proof (and then our formal setting) can be extended to cover the non-atomic case, but we have left this extension to future work.

Richer type systems, for example if they have bottom or top types or, in the case of the OCaml type system, `private` types and object types, have a non-atomic subtyping relations. To be able to extend our work to such settings, we have carefully marked each use of atomic subtyping in the formal development

with an hypothesis of v-closure, defined below. In the case of atomic subtyping, all types are v-closed.

Definition 1 (Constructor closure). *A type constructor $\overline{\alpha}$ $\mathtt{t}$ is v-closed if, for any type sequence $\overline{\sigma}$ and type τ such that $\overline{\sigma}$ $\mathtt{t} \prec_v \tau$ hold, then τ is necessarily equal to $\overline{\sigma}'$ $\mathtt{t}$ for some $\overline{\sigma}'$.*

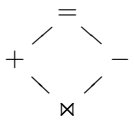
2.2 The algebra of variances

If we know that $\overline{\sigma}$ $\mathtt{t} \leq \overline{\sigma}'$ $\mathtt{t}$, that is $\overline{\sigma}$ $\mathtt{t} \prec_+ \overline{\sigma}'$ $\mathtt{t}$, and the constructor $\mathtt{t}$ has variable $\overline{v\alpha}$, an inversion principle tells us that for each i, $\sigma_i \prec_{v_i} \sigma'_i$. But what if we only know $\overline{\sigma}$ $\mathtt{t} \prec_u \overline{\sigma}'$ $\mathtt{t}$ for some variance u different from $(+)$? If u is $(-)$, we get the reverse relation $\sigma_i \succ_{v_i} \sigma'_i$. If u is $(\bowtie)$, we get $\sigma_i \bowtie \sigma'_i$, that is, nothing. This outlines a *composition* operation on variances $u.v_i$, such that if $\overline{\sigma}$ $\mathtt{t} \prec_u \overline{\sigma}'$ $\mathtt{t}$ then $\sigma_i \prec_{u.v_i} \sigma'_i$ holds. It is defined by the following table:

$v.w$	$=$	$+$	$-$	$\bowtie$
$=$	$=$	$=$	$=$	$\bowtie$
$+$	$=$	$+$	$-$	$\bowtie$
$-$	$=$	$-$	$+$	$\bowtie$
$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$
v				

This operation is associative and commutative. Such an operator, and the algebraic properties of variances explained below, have already been used by other authors, for example [Abe06].

There is a natural order relation between *variances*, which is the *coarser-than* order between the corresponding relations: $v \leq w$ if and only if $(\prec_v) \supseteq (\prec_w)$; *i.e.* if and only if, for all σ and τ, $\sigma \prec_w \tau$ implies $\sigma \prec_v \tau$.[6] This reflexive, partial order is described by the following lattice diagram:



That is, all variances are smaller than $=$ and bigger than $\bowtie$.

From the order lattice on variances we can define join $\vee$ and meet $\wedge$ of variances: $v \vee w$ is the biggest variance such that $v \vee w \leq v$ and $v \vee w \leq v$; conversely, $v \wedge w$ is the lowest variance such that $v \leq v \wedge w$ and $w \leq v \wedge w$. Finally, the composition operation is monotonous: if $v \leq v'$ then $w.v \leq w.v'$ (and $v.w \leq v'.w$).

[6] The reason for this order reversal is that the relations occur as hypotheses, in negative position, in definition of subtyping: if we have $v \leq w$ and $\mathtt{type}$ $v\alpha$ $\mathtt{t}$, it is safe to assume $\mathtt{type}$ $w\alpha$ $\mathtt{t}$: $\sigma \prec_w \sigma'$ implies $\sigma \prec_v \sigma'$, which implies σ $\mathtt{t} \leq \sigma'$ $\mathtt{t}$. One may also see it, as Abel notes, as an "information order": knowing that $\sigma \prec_+ \tau$ "gives you more information" than knowing that $\sigma \prec_\bowtie \tau$, therefore $\bowtie \leq +$.

We will frequently manipulate vectors $\overline{v\alpha}$, of variable associated with variances, which correpond to the "context" Γ of a type declaration. We extend our operation pairwise on those contexts: $\Gamma \vee \Gamma'$ and $\Gamma \wedge \Gamma'$, and the ordering between contexts $\Gamma \leq \Gamma'$. We also extend the variance-dependent subtyping relation ($\prec_v$), which becomes an order ($\prec_\Gamma$) between vectors of type of the same length: $\overline{\sigma} \prec_{\overline{v\alpha}} \overline{\sigma}'$ holds when for all i we have $\sigma_i \prec_{v_i} \sigma'_i$.

2.3 Variance assignment in ADTs

A counter-example To have a sound type system, some datatype declarations must be rejected. Assume (only for this example) that we have two base types `int` and `bool` such that $\texttt{bool} \leqslant \texttt{int}$ and $\texttt{int} \not\leqslant \texttt{bool}$. Consider the following type declaration:

```
type (+α, +β) t =
  | Fun of α → β
```

If it were accepted, we could build type the following program that deduces from the $(+\alpha)$ variance that $(\texttt{bool}, \texttt{bool})\ \texttt{t} \leq (\texttt{int}, \texttt{bool})\ \texttt{t}$; that is, we could turn the identity function of type $\texttt{bool} \to \texttt{bool}$ into one of type $\texttt{int} \to \texttt{bool}$ and then turns an integer into a boolean:

```
let three_as_bool : bool =
  match (Fun (fun x -> x) : (bool, bool) t :> (int, bool) t) with
    | Fun (danger : int → bool) -> danger 3
```

A requirement for type soundness We say that the type $\texttt{type}\ \overline{v\alpha}\ \texttt{t}$ defined by the constructors $(\texttt{K}_c\ \texttt{of}\ \tau^c[\overline{\alpha}])_{c \in C}$ is *well-signed* if

$$\forall c \in C, \forall \overline{\sigma}, \forall \overline{\sigma}', \quad \overline{\sigma}\ \texttt{t} \leq \overline{\sigma}'\ \texttt{t} \implies \tau^c[\overline{\sigma}] \leq \tau^c[\overline{\sigma}']$$

The definition of $(+\alpha, +\beta)\ \texttt{t}$ is not well-signed because we have $(\bot, \bot)\ \texttt{t} \leq (\texttt{int}, \bot)\ \texttt{t}$ according to the variance declaration, but we do not have the corresponding conclusion $(\texttt{int} \to \bot) \leq (\bot \to \bot)$.

This is a simplified version, specialized to simple algebraic datatypes, of the soundness criterion of Simonet and Pottier. They proved that this condition is *sufficient*[7] for soundness: if all datatype definitions accepted by the type-checker are well-signed, then both subject reduction and progress hold—for their static and dynamic semantics, using the subtyping relation ($\leq$) we have defined.

A judgment for variance assignment When reformulating the well-signedness requirement of Simonet and Pottier for simple ADT, in our specific case where the subtyping relation is defined by variance, it becomes a simple check on the variance of type definitions. Our example above is unsound as its claims α covariant while it in fact appears in negative position in the definition.

[7] It turns out that this condition is not *necessary* and can be slightly weakened: we discuss this in the extended version of this article.

$$\textsc{vc-Var}\ \frac{w\alpha \in \Gamma \qquad w \geq v}{\Gamma \vdash \alpha : v} \qquad \textsc{vc-Constr}\ \frac{\Gamma \vdash \texttt{type}\ \overline{w\alpha}\ \texttt{t} \qquad \forall i,\ \Gamma \vdash \sigma_i : v.w_i}{\Gamma \vdash \overline{\sigma}\ \texttt{t} : v}$$

Fig. 2. Variance assignment

We define a judgment to check the variance of a type expression. Given a context Γ of the form $\overline{v\alpha}$, that is, where each variable is annotated with a variance, the judgment $\Gamma \vdash \tau : v$ checks that the expression τ varies along v when the variables of τ vary along their variance in Γ. For example, $(+\alpha) \vdash \tau[\alpha] : +$ holds when $\tau[\alpha]$ is covariant in its variable α. The inference rules for the judgment $\Gamma \vdash \tau : v$ are defined on Figure 2.

The parameter v evolves when going into subderivations: when checking $\Gamma \vdash \tau_1 \to \tau_2 : v$, contravariance is expressed by checking $\Gamma \vdash \tau_1 : (v.-)$. Previous work (on variance as [Abe06] and [EKRY06], but also on irrelevance as in [Pfe01]) used no such parameter, but modified the context instead, checking $\Gamma/- \vdash \tau_1$ for some "variance cancellation" operation $vw/$ (see [Abe06] for a principled presentation). Our own inference rules preserve the same context in the whole derivation and can be more easily adapted to the decomposability judgment $\Gamma \vdash \tau : v \Rightarrow v'$ that we introduce in §3.4.

A semantics for variance assignment This syntactic judgment $\Gamma \vdash \tau : v$ corresponds to a semantics property about the types and context involved, which formalizes our intuition of "when the variables vary along Γ, the expression τ varies along v". We also give a few formal results about this judgment.

Definition 2 (Interpretation of the variance checking judgment). *We write $[\![\Gamma \vdash \tau : v]\!]$ for the property: $\forall \overline{\sigma}, \overline{\sigma}',\ \overline{\sigma} \prec_\Gamma \overline{\sigma}' \implies \tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$.*

Lemma 1 (Correctness of variance checking). *$\Gamma \vdash \tau : v$ is provable if and only if $[\![\Gamma \vdash \tau : v]\!]$ holds.*

Lemma 2 (Monotonicity). *If $\Gamma \vdash \tau : v$ is provable and $\Gamma \leq \Gamma'$ then $\Gamma' \vdash \tau : v$ is provable.*

Lemma 3 (Principality). *For any type τ and any variance v, there exists a minimal context Δ such that $\Delta \vdash \tau : v$ holds. That is, for any other context Γ such that $\Gamma \vdash \tau : v$, we have $\Delta \leq \Gamma$.*

We can generalize inversion of head type constructors (§2.1) to whole type expressions. The most general inversion is given by the principal context.

Theorem 1 (Inversion).

For any type $\tau[\overline{\alpha}]$, variance v, and type sequences $\overline{\sigma}$ and $\overline{\sigma}'$, the subtyping relation $\tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$ holds if and only if the judgment $\Gamma \vdash \tau : v$ holds for some context Γ such that $\overline{\sigma} \prec_\Gamma \overline{\sigma}'$.

Furthermore, if $\tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$, then $\overline{\sigma} \prec_\Delta \overline{\sigma}'$ holds, where Δ is the more general context such that $\Delta \vdash \tau : v$ holds.

Checking variance of type definitions We have all the machinery in place to explain the checking of ADT variance declarations. The well-signedness criterion of Simonet and Pottier gives us a general semantic characterization of which definitions are correct: a definition $\texttt{type}\ \overline{v\alpha}\ \texttt{t} = (\texttt{K}_c\ \texttt{of}\ (\tau^c[\overline{\alpha}])_{c\in C}$ is correct if, for each constructor c, we have:

$$\forall\overline{\sigma}, \forall\overline{\sigma}', \quad \overline{\sigma}\ \texttt{t} \leq \overline{\sigma}'\ \texttt{t} \implies \tau[\overline{\sigma}] \leq \tau[\overline{\sigma}']$$

By inversion of subtyping, $\overline{\sigma}\ \texttt{t} \leq \overline{\sigma}'\ \texttt{t}$ implies $\sigma_i \prec_{v_i} \sigma'_i$ for all i. Therefore, it suffices to check that:

$$\forall\overline{\sigma}, \forall\overline{\sigma}', \quad (\forall i, \sigma_i \prec_{v_i} \sigma'_i) \implies \tau[\overline{\sigma}] \leq \tau[\overline{\sigma}']$$

This is exactly the semantic property corresponding to the judgment $\overline{v\alpha} \vdash \tau : (+)$! That is, we have reduced soundness verification of an algebraic type definition to a mechanical syntactic check on the constructor argument type.

This syntactic criterion is very close to the one implemented in actual type checkers, which do not need to decide general subtyping judgments—or worse solve general subtyping constraints—to check variance of datatype parameters. Our aim is now to find a similar syntactic criterion for the soundness of variance annotations on guarded algebraic datatypes, rather than simple algebraic datatypes.

2.4 Variance annotations in GADT

A general description of GADT When used to build terms of type $\overline{\alpha}\ \texttt{t}$, a constructor $\texttt{K}\ \texttt{of}\ \tau$ behaves like a function of type $\forall\overline{\alpha}.(\tau \to \overline{\alpha}\ \texttt{t})$. Remark that the codomain is exactly $\overline{\alpha}\ \texttt{t}$, the type $\texttt{t}$ instantiated with parametric variables. GADT arise by relaxing this restriction, allowing to specify constructors with richer types of the form $\forall\overline{\alpha}.(\tau \to \overline{\sigma}\ \texttt{t})$. See for example the declaration of constructor `Prod` in the introduction:

$$\texttt{|}\ \texttt{Prod}\ \texttt{:}\ \forall\beta\gamma.\ \beta\ \texttt{expr} * \gamma\ \texttt{expr} \to (\beta * \gamma)\ \texttt{expr}$$

Instead of being just α `expr`, the codomain is now $(\beta * \gamma)$ `expr`. We moved from simple algebraic datatypes to so-called *generalized* algebraic datatypes. This approach is natural and convenient for the users, so it is exactly the syntax chosen in languages with explicit GADT support, such as Haskell and OCaml, and is reminiscent of the inductive datatype definitions of dependently typed languages.

However, for formal study of GADT, a different formulation based on equality constraints is preferred. We will use the following equivalent presentation, in the syntax of Simonet and Pottier. The idea is that instead of having $(\beta * \gamma)\ \texttt{t}$ as codomain of the constructor `Prod`, we will force it to be $\alpha\ \texttt{t}$ again, by adding an explicit type equality $\alpha = \beta * \gamma$.

```
type α expr =
  | Val of ∃β[α = β]. β
  | Int of [α = int]. int
  | Thunk of ∃βγ[α = γ]. β expr ∗ (β → γ)
  | Prod of ∃βγ[α = β ∗ γ]. β expr ∗ γ expr
```

In the rest of the paper, we extend our former core language with such guarded algebraic datatypes. This impacts the typing rules (which are precisely defined in Simonet and Pottier), but not the notion of subtyping, which is defined on (GADT) type constructors with variance `type` $\overline{v\alpha}$ `t` just as it previously was on simple datatypes. What needs to be changed, however, is the soundness criterion for checking the variance of type definitions.

The correctness criterion Simonet and Pottier [SP07] define a general framework HMG(X) to study type systems with GADT where the type equalities in bounded quantification are generalized to an arbitrary constraint language. They make few assumptions on the type system used, mostly that it has function types $\sigma \to \tau$, user-definable (guarded) algebraic datatypes $\overline{\alpha}$ `t`, and a subtyping relation $\sigma \leq \tau$ (which may be just equality, in languages without subtyping).

They use this general type system to give static semantics (typing rules) to a fixed untyped lambda-calculus equipped with datatype construction and pattern matching operations. They are able to prove a type soundness result under just some general assumptions on the particular subtyping relation ($\leq$). Here are the three requirements to get their soundness result:

1. Incomparability of distinct types: for all types $\tau_1, \tau_2, \overline{\sigma}, \overline{\sigma}'$ and distinct datatypes $\overline{\alpha}$ `t`, $\overline{\alpha}'$ `t`$'$, the types $(\tau_1 \to \tau_2)$, $\tau_1 * \tau_2$, $\overline{\sigma}$ `t` and $\overline{\sigma}'$ `t`$'$ must be pairwise incomparable (both $\not\leq$ and $\not\geq$) — this is where our restriction to an atomic subtyping relation, discussed in §2.1, comes from.
2. Decomposability of function and product types: if $\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2$ (respectively $\tau_1 * \tau_2 \leq \sigma_1 * \sigma_2$), we must have $\tau_1 \geq \sigma_1$ (resp. $\tau_1 \leq \sigma_1$) and $\tau_2 \leq \sigma_2$.
3. Decomposability of datatypes[8]: for each datatype $\overline{\alpha}$ `t` and all type vectors $\overline{\sigma}$ and $\overline{\sigma}'$ such that $\overline{\sigma}$ `t` $\leq \overline{\sigma}'$ `t`, we must have $(\exists\overline{\beta}[D[\overline{\sigma}]]\tau) \leq (\exists\overline{\beta}[D[\overline{\sigma}']]\tau)$ for each constructor `K of` $\exists\overline{\beta}[D[\overline{\beta}, \overline{\alpha}]].\, \tau[\overline{\beta}]$.

Those three criteria are necessary for the soundness proof. We will now explain how variance of type parameters impact those requirements, that is, how to match a GADT implementation against a variance specification. With our definition of subtyping based on variance, and the assumption that the datatype $\overline{v\alpha}$ `t` we are defining indeed has variance $\overline{v\alpha}$, is the GADT decomposability requirement (item 3 above) satisfied by all its constructors? If so, then the datatype definition is sound and can be accepted. Otherwise, the datatype definition does not match the specified variance, and should be rejected by the type checker.

[8] This is an extended version of the soundness requirement for algebraic datatypes: it is now formulated in terms guarded existential types $\exists\overline{\beta}[D]\tau$ rather than simple argument types τ.

3 Checking variances of GADT

For every type definition, we need to check that the decomposability requirement of Simonet and Pottier holds. Remark that it is expressed for each GADT constructor independently of the other constructors for the same type: we can check one constructor at a time.

Assume we check a fixed constructor `K` of argument type $\exists\overline{\beta}[D[\overline{\alpha},\overline{\beta}]].\,\tau[\overline{\beta}]$. Simonet and Pottier prove that their requirement is equivalent to the following formula, which is more convenient to manipulate:

$$\forall\overline{\sigma},\overline{\sigma}',\overline{\rho},\ \left(\overline{\sigma}\ \mathtt{t} \le \overline{\sigma}'\ \mathtt{t} \wedge D[\overline{\sigma},\overline{\rho}] \implies \exists\overline{\rho}',\ D[\overline{\sigma}',\overline{\rho}'] \wedge \tau[\overline{\rho}] \le \tau[\overline{\rho}']\right) \quad \text{(REQ-SP)}$$

The purpose of this section is to extract a practical criterion equivalent to this requirement. It should not be expressed as a general constraint satisfaction problem, but rather as a syntax-directed and decidable algorithm that can be used in a type-checker—without having to implement a full-blown constraint solver.

3.1 Expressing decomposability

If we specialize REQ-SP to the `Prod` constructor of the α `expr` example datatype, *i.e.* `Prod of` $\exists\beta\gamma[\alpha = \beta * \gamma]\beta$ `expr` $* \gamma$ `expr`, we get:

$$\forall\sigma,\sigma',\rho_1,\rho_2,\\ \left(\sigma\ \mathtt{expr} \le \sigma'\ \mathtt{expr} \wedge \sigma = \rho_1 * \rho_2 \implies \exists\rho_1',\rho_2',(\sigma' = \rho_1' * \rho_2' \wedge \rho_1 * \rho_2 \le \rho_1' * \rho_2')\right)$$

We can substitute equalities and use the (assumed) covariance to simplify the subtyping constraint σ `expr` $\le \sigma'$ `expr` into $\sigma \le \sigma'$:

$$\forall\sigma',\rho_1,\rho_2,\ \left(\rho_1 * \rho_2 \le \sigma' \implies \exists\rho_1',\rho_2',\ (\sigma' = \rho_1' * \rho_2' \wedge \rho_1 \le \rho_1' \wedge \rho_2 \le \rho_2')\right)\ \mathbf{(1)}$$

This is the *upward closure* property mentioned in the introduction. This transformation is safe only if any supertype σ' of a product $\rho_1 * \rho_2$ is itself a product, *i.e.* is of the form $\rho_1' * \rho_2'$ for some ρ_1' and ρ_2'.

More generally, for a type $\Gamma \vdash \sigma$ and a variance v, we are interested in a closure property of the form

$$\forall(\overline{\rho} : \Gamma),\sigma',\quad \sigma[\overline{\rho}] \prec_v \sigma' \implies \exists(\overline{\rho}' : \Gamma),\ \sigma' = \sigma[\overline{\rho}']$$

Here, the context Γ represents the set of existential variables of the constructor (β and γ in our example). We can easily express the condition $\rho_1 \le \rho_1'$ and $\rho_2 \le \rho_2'$ on the right-hand side of the implication by considering a context Γ annotated with variances $(+\beta,+\gamma)$, and using the context ordering ($\prec_\Gamma$). Then, (1) is equivalent to:

$$\forall(\overline{\rho} : \Gamma),\sigma',\quad \sigma[\overline{\rho}] \prec_v \sigma' \implies \exists(\overline{\rho}' : \Gamma),\ \overline{\rho} \prec_\Gamma \overline{\rho}' \wedge \sigma' = \sigma[\overline{\rho}']$$

Our aim is now to find a set of inference rules to check decomposability; we will later reconnect it to REQ-SP. In fact, we study a slightly more general relation, where the equality $\sigma[\overline{\rho}'] = \sigma'$ on the right-hand side is relaxed to an arbitrary relation $\sigma[\overline{\rho}'] \prec_{v'} \sigma'$:

Definition 3 (Decomposability). *Given a context Γ, a type expression $\sigma[\overline{\beta}]$ and two variances v and v', we say that σ is* decomposable *under Γ from variance v to variance v', which we write $\Gamma \Vdash \sigma : v \rightsquigarrow v'$, if the property*

$$\forall(\overline{\rho} : \Gamma), \sigma', \ \sigma[\overline{\rho}] \prec_v \sigma' \implies \exists(\overline{\rho}' : \Gamma), \ \overline{\rho} \prec_\Gamma \overline{\rho}' \ \wedge \ \sigma[\overline{\rho}'] \prec_{v'} \sigma'$$

holds.

We use the symbol $\Vdash$ rather than $\vdash$ to highlight the fact that this is just a logic formula, not the semantics criterion corresponding to an inductive judgment, nor a syntactic judgment—we will introduce one later in section 3.4.

Remark that, due to the *positive* occurrence of the relation $\prec_\Gamma$ in the proposition $\Gamma \Vdash \tau : v \rightsquigarrow v'$ and the anti-monotonicity of $\prec_\Gamma$, this formula is "anti-monotonous" with respect to the context ordering $\Gamma \leq \Gamma'$. This corresponds to saying that we can still decompose, but with less information on the existential witness $\overline{\rho}'$.

Lemma 4 (Anti-monotonicity). *If $\Gamma \Vdash \tau : v \rightsquigarrow v'$ holds and $\Gamma' \leq \Gamma$, then $\Gamma' \Vdash \tau : v \rightsquigarrow v'$ also holds.*

In the following subsections, we study the subtleties of decomposability.

3.2 Variable occurrences

In the `Prod` case, the type whose decomposability was considered is $\beta * \gamma$ (in the context β, γ). In this very simple case, decomposability depends only on the type constructor for the product. In the present type system, with very strong invertibility principles on the subtyping relation, both upward and downward closures hold for products—and any other head type constructor. In the general case, we require that this specific type constructor be upward-closed.

In the general case, the closure of the head type constructor alone is not enough to ensure decomposability of the whole type. For example, in a complex type expression with subterms, we should consider the closure of the type constructors appearing in the subterms as well. Besides, there are subtleties when a variable occurs several times.

For example, while $\beta * \gamma$ is decomposable from (+) to (=), $\beta * \beta$ is not: $\bot * \bot$ is an instantiation of $\beta * \beta$, and a subtype of, *e.g.*, $\texttt{int} * \texttt{bool}$, but it is not equal to $(\beta * \beta)[\gamma']$ for any γ'. The same variable occurring twice in covariant position (or having one covariant and one invariant or contravariant occurence) breaks decomposability.

On the other hand, two invariant occurrences are possible: $\beta\ \texttt{ref} * \beta\ \texttt{ref}$ is upward-closed (assuming the type constructor `ref` is invariant and upward-closed): if $(\sigma\ \texttt{ref} * \sigma\ \texttt{ref}) \leq \sigma'$, then by upward closure of the product, σ' is of the form $\sigma'_1 * \sigma'_2$, and by its covariance $\sigma\ \texttt{ref} \leq \sigma'_1$ and $\sigma\ \texttt{ref} \leq \sigma'_2$. Now by invariance of `ref` we have $\sigma'_1 = \sigma\ \texttt{ref} = \sigma'_2$, and therefore σ' is equal to $\sigma\ \texttt{ref} * \sigma\ \texttt{ref}$, which is an instance[9] of $\beta\ \texttt{ref} * \beta\ \texttt{ref}$.

[9] We use the term *instance* to denote the replacement of all the free variables of a type expression under context by closed types—not the specialization of an ML type scheme.

Finally, a variable may appear in irrelevant positions without affecting closure properties; $\beta * (\beta\ \texttt{irr})$ (where `irr` is an upward-closed irrelevant type, defined for example as $\texttt{type}\ \alpha\ \texttt{irr} = \texttt{int}$) is upward closed: if $\sigma * (\sigma\ \texttt{irr}) \leq \sigma'$, then σ' is of the form $\sigma'_1 * (\sigma'_2\ \texttt{irr})$ with $\sigma \leq \sigma'_1$ and $\sigma \bowtie \sigma'_2$, which is equiconvertible to $\sigma'_1 * (\sigma'_1\ \texttt{irr})$ by irrelevance, an instance of $\beta * (\beta\ \texttt{irr})$.

3.3 Context zipping

The intuition to think about these different cases is to consider that, for any σ', we are looking for a way to construct a "witness" $\overline{\sigma}'$ such that $\tau[\overline{\sigma}'] = \sigma'$ from the hypothesis $\tau[\overline{\sigma}] \prec_v \sigma'$. When a type variable appears only once, its witness can be determined by inspecting the corresponding position in the type σ'. For example in $\alpha * \beta \leq \texttt{bool} * \texttt{int}$, the mapping $\alpha \mapsto \texttt{bool}, \beta \mapsto \texttt{int}$ gives the witness pair $\texttt{bool}, \texttt{int}$.

However, when a variable appears twice, the two witnesses corresponding to the two occurrences may not coincide. (Consider for example $\beta * \beta \leq \texttt{bool} * \texttt{int}$.) If a variable β_i appears in several *invariant* occurrences, the witness of each occurrence is forced to be equal to the corresponding subterm of $\tau[\overline{\sigma}]$, that is σ_i, and therefore the various witnesses are themselves equal, hence compatible. On the contrary, for two covariant occurrences (as in the $\beta * \beta$ case), it is possible to pick a σ' such that the two witnesses are incompatible—and similarly for one covariant and one invariant occurrence. Finally, an irrelevant occurrence will never break closure properties, as all witnesses (forced by another occurrence) are compatible.

To express these merging properties, we define a "zip"[10] operation $v_1 \curlywedge v_2$, that formally expresses which combinations of variances are possible for several occurrences of the same variable; it is a partial operation (for example, it is not defined in the covariant-covariant case, which breaks the closure properties) with the following table:

$v \curlywedge w$	$=$	$+$	$-$	$\bowtie$	w
$=$	$=$			$=$	
$+$				$+$	
$-$				$-$	
$\bowtie$	$=$	$+$	$-$	$\bowtie$	
v					

3.4 Syntactic decomposability

Equipped with the zipping operation, we introduce a judgment $\Gamma \vdash \tau : v \Rightarrow v'$ to express decomposability, syntactically, defined by the inference rules on Figure 3.

We sometimes need to merge sub-derivations into larger ones, so in addition to decomposability, the judgments simultaneously ensures that v is a correct

[10] The idea of context merging and the term "zipping" are inspired by Montagu and Remy [MR09]

$$\textsc{sc-Triv}\ \frac{v \geq v' \qquad \Gamma \vdash \tau : v}{\Gamma \vdash \tau : v \Rightarrow v'} \qquad\qquad \textsc{sc-Var}\ \frac{w\alpha \in \Gamma \qquad w = v}{\Gamma \vdash \alpha : v \Rightarrow v'}$$

$$\textsc{sc-Constr}\ \frac{\Gamma \vdash \mathtt{type}\ \overline{w\alpha}\ \mathtt{t} : v\text{-closed} \qquad \Gamma = \curlywedge_i \Gamma_i \qquad \forall i,\ \Gamma_i \vdash \sigma_i : v.w_i \Rightarrow v'.w_i}{\Gamma \vdash \overline{\sigma}\ \mathtt{t} : v \Rightarrow v'}$$

Fig. 3. Syntactic decomposablity

variance for τ under Γ. Actually, in order to understand the details of this judgment, it is quite instructive to compare it with the variance-checking judgment $\Gamma \vdash \tau : v$ defined on Figure 2.

The first thing to notice is that the present rules are not completely syntax-directed: we first check whether $v \geq v'$ holds, and if not, we apply syntax-directed inference rules; existence of derivations is still easily decidable. If $v \geq v'$ holds, satisfying the semantics criterion is trivial: $\tau[\overline{\sigma}] \prec_v \tau'$ implies $\tau[\overline{\sigma}] \prec_{v'} \tau'$, so taking $\overline{\sigma}$ for $\overline{\sigma}'$ is always a correct witness, which is represented by Rule sc-Triv. The other rules then follow the same structure as the variance-checking judgment.

Rule sc-Var is very similar to vc-Var, except that the condition $w \geq v$ is replaced by a stronger equality $w = v$. This difference comes from the fact that the semantics condition for closure checking (Definition 2) includes both a variance check, which is monotonic in the context (Lemma 2) and the decomposability property, which is anti-monotonic (Lemma 4), so the present judgment must be invariant with respect to the context.

The most interesting rule is sc-Constr. It checks first that the head type constructor is v-closed (according to Definition 1); then, it checks each subtype for decomposability from v to v' *with compatible witnesses*, that is, in an environment family Γ_i that can be zipped into a unique environment Γ.

In order to connect the syntactic and semantics versions of decomposability, we define the interpretation $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ of syntactic decomposability.

Definition 4 (Interpretation of syntactic decomposability).
We write $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ for the conjunction of properties $[\![\Gamma \vdash \tau : v]\!]$ and $\Gamma \Vdash \tau : v \rightsquigarrow v'$.

Note that our interpretation $\Gamma \vdash \tau : v \Rightarrow v'$ does not coincide with our previous decomposability formula $\Gamma \Vdash \tau : v \rightsquigarrow v'$, because of the additional variance-checking hypothesis that makes it composable. The distinction between those two notions of decomposition is not useful to have a sound criterion, but is crucial to be complete with respect to the criterion of Simonet and Pottier, which imposes no variance checking condition.

Lemma 5 (Soundness of syntactic decomposability).
If the judgment $\Gamma \vdash \tau : v \Rightarrow v'$ holds, then $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ is holds.

Completeness is the general case is however much more difficult and we only prove it when the right-hand side variance v' is $(=)$. In other words, we take back the generality that we have introduced in §3.1 when defining decomposability.

Lemma 6 (Completeness of syntactic decomposability). *If* $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ *holds for* $v' \in \{=, \bowtie\}$*, then* $\Gamma \vdash \tau : v \Rightarrow v'$ *is provable.*

Lemma 6 is an essential piece to finally turn the correctness criterion REQ-SP of Simonet and Pottier into a purely syntactic criterion.

Theorem 2 (Algorithmic criterion). *The* REQ-SP *criterion is equivalent to*

$$\exists \Gamma, (\Gamma_i)_i, \quad \Gamma \vdash \tau : (+) \quad \wedge \quad \Gamma = \bigwedge_i \Gamma_i \quad \wedge \quad \forall i,\, \Gamma_i \vdash T_i : v_i \Rightarrow (=)$$

This presentation of the correctness criterion only relies on syntactic judgments. It is pragmatic in the sense that it suggests a simple and direct implementation, as a generalization of the check currently implemented in type system engines — which are only concerned with the $\Gamma \vdash \tau : +$ part.

To compute the contexts Γ and $(\Gamma_i)_{i \in I}$ existentially quantified in this formula, one can use a variant of our syntactic judgments where the environment Γ is not an input, but an output of the judgment; in fact, one should return for each variable α the *set* of possible variances for this judgment to hold. For example, the query $(? \vdash \alpha * \beta \texttt{ ref} : +)$ should return $(\alpha \mapsto \{+, =\}; \beta \mapsto \{=\})$. Defining those algorithmic variants of the judgments is routine, and we have not done it hereby lack of space. The sets of variances corresponding to the decomposability of the $(T_i)_{i \in I}$ $(? \vdash T_i : v_i \Rightarrow (=))$ should be zipped together and intersected with the possibles variances for τ, returned by $(? \vdash \tau : +)$. The algorithmic criterion is satisfied if and only if the intersection is not empty; this can be decided in a simple and efficient way.

4 Closed-world vs. open-world subtyping

4.1 A better control on upward and downward-closure

As explained in the introduction, the problem with the upward and downward closure properties is that they are not monotonic: enriching the subtyping lattice of our type system does not preserve them. While the core language has a nice variance check for GADT, adding private types in particular destroys the downward-closure property of the whole type system.

Our proposed solution to this tension is to give the user the choice to locally strengthen negative knowledge about the subtyping relation by abandoning some flexibility. Just as object-oriented languages have a concept of `final` classes that cannot be extended, we would like to allow to define `downward-closed` datatypes, whose private counterparts cannot be declared, and `upward-closed` datatypes that cannot be made `invisible`: defining `type t = private` τ would be rejected by the type-checker if τ was itself declared `downward-closed`.

4.2 Subtyping constraints and variance assignment

Consider our introductory example α `expr` of strongly typed expressions (§1). A simple way to get such a type to be covariant would be, instead of proving delicate, non-monotonic upward-closure properties on the tuple type involved in the equation $\alpha = \beta * \gamma$, to *change* this definition so that the resulting type is obviously covariant:

```
type +α expr =
  | Val of ∃β[α ≥ β]. β
  | Int of [α ≥ int]. int
  | Thunk of ∃βγ[α ≥ γ]. β expr * (β → γ)
  | Prod of ∃βγ[α ≥ β * γ]. β expr * γ expr
```

We have turned each equality constraint $\alpha = T[\overline{\beta}]$ into a subtyping constraint $\alpha \geq T[\overline{\beta}]$. For a type α' such that $\alpha \leq \alpha'$, we get by transitivity that $\alpha' \geq T[\overline{\beta}]$. This means that α `expr` trivially satisfies the correctness criterion from Simonet and Pottier. Formally, instead of checking $\Gamma \vdash T_i : v_i \Rightarrow (=)$, we are now checking $\Gamma \vdash T_i : v_i \Rightarrow (+)$, which is significantly easier to satisfy: when v_i is itself $+$ we can directly apply the SC-Triv rule.

While we now have a different datatype, which gives us a weaker subtyping assumption when pattern-matching, we are still able to write the classic function `eval` : α `expr` $\to \alpha$, because the constraints $\alpha \geq \tau$ are in the right direction to get an α as a result.

```
let rec eval : α expr → α = function
  | Val β (v : β) -> (v :> α)
  | Int (n : int) -> (n :> α)
  | Thunk βγ ((v : β expr), (f : β → γ)) ->
    (f (eval v) :> α)
  | Prod β γ ((b : β expr), (c : γ expr)) ->
    ((eval b, eval c) :> α)
```

However, allowing subtyping constraints in GADT has some disadvantages. If the language requires subtyping casts to be explicit, this would make pattern matching of GADT syntactically heavier than with current GADT where equalities constraints are used implicitly. Subtyping constraints need also be explicit in the type declaration, forcing the user out of the convenient "generalized codomain type" syntax.

From a theoretical standpoint, we think there is value in exploring both directions: experimenting with GADT using subtyping constraints, and with fine-grained closure properties for equality constraints. Both designs allow to reason in an open world setting, by being resilient to extensions of the subtyping relation. Whether it is possible to expose those features to the expert language user (*e.g.* library designers) without forcing all users to pay the complexity burden remains to be seen.

5 Future Work

Extension of the formal exposition to non-atomic subtyping As remarked in §2.1 during the definition of our formal subtyping relation, the soundness proof of Simonet and Pottier is restricted to atomic subtyping. We conjecture that their work can be extended to non-atomic subtyping, and furthermore that our results would extend seamlessly in this setting, thanks to our explicit use of the v-closure hypothesis.

Experiments with v-closure of type constructors as a new semantic property In a language with non-atomic subtyping such as OCaml, we need to distinguish v-closed and non-v-closed type constructors. This is a new semantic property that, in particular, must be reflected through abstraction boundaries: we should be able to say about an abstract type that it is v-closed, or not say anything.

How inconvenient in practice is the need to expose those properties to have good variance for GADT? Will the users be able to determine whether they want to enforce v-closure for a particular type they are defining?

Experiments with subtyping constraints in GADT In §4.2, we have presented a different way to define GADT with weaker constraints (simple subtyping instead of equality) and stronger variance properties. It is interesting to note that, for the few GADT examples we have considered, using subtyping constraints rather than equality constraints was sufficient for the desired applications of the GADT.

However, there are cases were the strong equality relying on fine-grained closure properties is required. We need to consider more examples of both cases to evaluate the expressiveness trade-off in, for example, deciding to add only one of these solutions to an existing type system.

Completeness of variance annotations with domain information For simple algebraic datatypes, variance annotations are "enough" to say anything we want to say about the variance of datatypes. Essentially, all admissible variance relations between datatypes can be described by considering the pairwise variance of their parameters, separately.

This does not work anymore with GADT. For example, with only this notion of variance, all we can soundly say about the equality type (α, β) `eq` is that it must be invariant in both its parameters. In particular, the well-known trick of "factoring out" GADT by using the `eq` type in place of equality constraint does not preserve variances.

We think it would possible to regain some "completeness", and in particular re-enable factoring by `eq`, by considering *domain information*, that is information on constraints that must hold for the type to be inhabited. If we restricted the subtyping rule with conclusion $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}$ to only cases where $\overline{\sigma}\ \mathtt{t}$ and $\overline{\sigma}'\ \mathtt{t}$ are inhabited—with a separate rule to conclude subtyping in the non-inhabited case—we could have a finer variance check, as we would only need to show that the criterion of Simonet and Pottier holds between two instances of the inhabited domain, and not any instance. If we stated that the domain of the

type (α, β) `eq` is restricted by the constraint $\alpha = \beta$, we could soundly declare the variance $(\bowtie\alpha, \bowtie\beta)$ `eq` on this domain—which no longer prevents from factoring out GADT by equality types.

Conclusion

Checking the variance of GADT is surprisingly more difficult (and interesting) than we initially thought. We have studied a novel criterion of upward and downward closure of type expressions and proposed a corresponding syntactic judgment that is easily implementable. We presented a core formal framework to prove both its correctness and its completeness with respect to the more general criterion of Simonet and Pottier.

This closure criterion exposes important tensions in the design of a subtyping relation, for which we previously knew of no convincing example in the context of ML-derived programming languages. We have suggested new language features to help alleviate these tensions, whose convenience and practicality is yet to be assessed by real-world usage.

Considering extension of GADT in a rich type system is useful in practice; it is also an interesting and demanding test of one's type system design.

References

Abe06. Andreas Abel. Polarized subtyping for sized types. *Mathematical Structures in Computer Science*, 2006. Special issue on subtyping, edited by Healfdene Goguen and Adriana Compagnoni.

EKRY06. Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C# generics. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, 2006.

Gar04. Jacques Garrigue. Relaxing the value restriction. In *In International Symposium on Functional and Logic Programming, Nara, LNCS 2998*, 2004.

KR05. Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005. URL: `http://research.microsoft.com/pubs/64040/gadtoop.pdf`.

MR09. Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009. URL: `http://gallium.inria.fr/~remy/modules/Montagu-Remy@popl09:fzip.pdf`.

Pfe01. Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th IEEE Symposium on Logic in Computer Science (LICS 2001), 16-19 June 2001, Boston University, USA, Proceedings*, 2001.

SP07. Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), January 2007.

SR. Gabriel Scherer and Didier Rémy. Gadt meet subtyping. Long version, available electronically. URL: `http://gallium.inria.fr/~remy/gadts/`.