GADTs Meet Subtyping*

Gabriel Scherer and Didier Rémy

INRIA, Rocquencourt

Abstract. While generalized algebraic datatypes (GADTs) 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 Introduction

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 with parametric polymorphism whose programming idioms rely on subtyping, in particular object-oriented languages, or languages with structural datatypes such as extensible records and variants, dependently typed languages with inductive types (to represent positivity requirements), or additional information in types such as permissions, effects, *etc.* A last 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, as they may not classify mutable data. Therefore, it is important for extensions of type definitions, such as generalized algebraic datatypes (GADTs), 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 exp =
  | Val : α → α exp
  | Int : int → int exp
  | Thunk : ∀β. β exp * (β → α) → α exp
  | Prod : ∀βγ. β exp * γ exp → (β * γ) exp
```

* Part of this work has been done at IRILL.

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 554–573, 2013.

Is it safe to say that `exp` is covariant in its type parameter? It turns out that, using the subtyping relation of the OCaml type system, the answer is "yes". But, surprisingly to us, in a type system with a top type $\top$, the answer would be "no". We introduce this example in details in §2—and present some interesting counter-examples of incorrect variance annotations.

Verifying variance annotations for simple algebraic datatypes is straightforward: it suffices to check that covariant type variables appear only positively and contravariant variables only negatively in the types of the arguments of the datatype constructors. GADTs can be formalized as extensions of datatypes where constructors have typed arguments, but also a set of existential variables and equality constraints. Then, the simple check of algebraic datatypes apparently becomes a searching problem: witnesses for existentials must be found so as to satisfy the equality constraints. That is, there is a natural correctness criterion (already present in previous work); however, it is expressed in a "semantic" form that is not suitable for a simple implementation in a type checker. We present this semantic criterion in §3 after reviewing the formal framework of variance-based subtyping.

The main contribution of our work, described in §4, is to develop a syntactic criterion that ensures the semantics criterion. Our solution extends the simple check of algebraic datatypes in a non-obvious way by introducing two new notions. First, *upward and downward-closure* of type constructors explains how to check that a single equality constraint is still satisfiable in presence of variance (but also raises interesting design issues for the subtyping relation). Second, *zipping* explains when witnesses exist for existential variables, that is, when multiple constraints using the same existential may soundly be used without interfering with each other. These two properties are combined into a new syntactic judgment of *decomposability* that is central to our syntactic criterion. We prove that our syntactic criterion is sound and complete with respect to the semantic criterion. The proof of soundness is relatively direct, but completeness is much harder.

We discuss the implication of our results in §5, in particular the notion of upward and downward-closure properties of type constructors, on the design of a subtyping relation. We also contrast this approach, motivated by the needs of a language of a ML family, with a different and mostly orthogonal approach taken by existing object-oriented languages, namely `C♯` and Scala, where a natural notion of GADTs involves subtyping constraints, rather than equality constraints. We can re-evaluate our syntactic criterion in this setting: it is still sound, but the question of completeness is left open.

In summary, we propose a syntactic criterion for checking the soundness of variance annotations of GADTs with equality constraints in a language with subtyping. Our work is directly applicable to the OCaml language, but our approach can also be transposed to languages with general subtyping constraints, and raises interesting design questions. A long version of the present article, containing the detailed proofs and additional details and discussion, is available online [SR].

2 Examples

Let us first explain why it is reasonable to say that α `exp` 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 `exp` into one of type α' `exp`. Here is some pseudo-code[1] for the coercion function:

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

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 $\gamma \leq \gamma'$. We may thus conclude by casting at types β' `exp` and γ' `exp`, recursively.

Similarly, in the `Int` case, we know that α must be an `int` and therefore an `int exp` 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* 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 =
  | Refl : ∀γ. (γ, γ) eq

let cast r =
  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 a cast from any value of type τ back into one of type σ, which is unsound in general.

[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.

[3] This counterexample is due to Jeremy Yallop.

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

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

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 `bad`, if accepted by the compiler, would allow us to cast a value of type `< m : int > bad` into one of type `< > bad` and thus have the above value `v` of type `<> bad`. However, if such a value `v` existed, we could produce 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 bad → (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 s =` τ *internally* may describe its interface as `type s = private` τ. This is a compromise between a type abbreviation and an abstract type: it is possible to cast a value of type `s` into one of type τ, but not, conversely, to construct a value of type `s` from one of type τ. In other words, `s` is a strict subtype of τ: we have `s` $\leq \tau$ but not `s` $\geq \tau$. 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 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`).

Finally, the need for covariance of α `exp` can be justified either by applications using subtyping on data (for example object types or polymorphic variants), or by the relaxed value restriction. If we used the `Thunk` constructor to delay a computation returning an object of type `< m : int >`, that is itself of type `< m : int > exp`, we may need to see it as a computation returning the empty object `< >`. We could also wish to define an abstract interface through a module boundary that would not expose any implementation detail about the datatype; for example, using `Product` to implement a `list` interface.

[4] This counterexample is due to Jacques Garrigue.

```
module Exp : sig
  type α exp
  val inj : α -> α exp
  val pair : α exp -> β exp -> (α * β) exp
  val fst : (α * β) exp -> α exp
end
```

What would then be the type of `Exp.inj []`? In presence of the value restriction, this application cannot be generalized, and we get a weak polymorphic type $?\alpha$ `list Exp.exp` for some non-generalized inference variable $?\alpha$. If we change the interface to express that `Exp.exp` is covariant, then we get the expected polymorphic type $\forall\alpha.\alpha$ `list Exp.exp`.

3 A Formal Setting

3.1 The Subtyping Relation

Ground types consist of base type $\mathtt{q}$, types τ $\mathtt{p}$, function types $\tau_1 \to \tau_2$, product types $\tau_1 * \tau_2$, and a set of algebraic datatypes $\overline{\sigma}$ $\mathtt{t}$. We also write σ and ρ for types, $\overline{\sigma}$ for a sequence of types $(\sigma_i)_{i\in I}$, and we use prefix notation for datatype parameters, as is the usage in ML. Datatypes may be user-defined by toplevel declarations of the form:

$$\mathtt{type}\ \overline{v\alpha}\ \mathtt{t} = \mathtt{K}_1\ \mathtt{of}\ \tau^1[\overline{\alpha}] \mid \ldots\ \mathtt{K}_n\ \mathtt{of}\ \tau^n[\overline{\alpha}]$$

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 $\mathtt{K}_c$ 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. Variances range in the set $\{+, -, =, \bowtie\}$. We may associate 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 $(\bowtie)$, *i.e.* 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, SUB-FUN and SUB-PROD, can be subsumed by the rule for datatypes SUB-CONSTR. Indeed, one can consider them as special datatypes (with a specific

$$\textsc{sub-Refl}\ \frac{}{\sigma \leq \sigma} \qquad \textsc{sub-Trans}\ \frac{\sigma_1 \leq \sigma_2 \qquad \sigma_2 \leq \sigma_3}{\sigma_1 \leq \sigma_3} \qquad \textsc{sub-Fun}\ \frac{\sigma \geq \sigma' \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

$$\textsc{sub-Prod}\ \frac{\sigma \leq \sigma' \qquad \tau \leq \tau'}{\sigma * \tau \leq \sigma' * \tau'} \qquad \textsc{sub-Constr}\ \frac{\texttt{type}\ \overline{v\alpha}\ \texttt{t} \qquad \forall i,\ \sigma_i \prec_{v_i} \sigma'_i}{\overline{\sigma}\ \texttt{t} \leq \overline{\sigma'}\ \texttt{t}} \qquad \textsc{sub-P}\ \frac{\sigma \leq \sigma'}{\sigma\ \texttt{p} \leq \sigma'\ \texttt{p}} \qquad \textsc{sub-PQ}\ \frac{}{\sigma\ \texttt{p} \leq \texttt{q}}$$

Fig. 1. Subtyping relation

dynamic semantics) of variance $(-,+)$ and $(+,+)$, respectively. For this reason, the following definitions will not explicitly detail the cases for arrows and products.

The rules SUB-P and SUB-PQ were added for the explicit purpose of introducing some amount of non-atomic subtyping in our relation. For two fixed type constructors `p` (unary) and `q` (nullary), we have σ `p` $\leq$ `q` for any σ. Note that `q` is not a top type as it is not above all types, only above the σ `p`. Of course, we could add other such type constructors, but those are enough to make the system interesting and representative of complex subtype relation.

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

The non-atomic rule SUB-PQ ensures that our subtyping relation is not "too structured" and is a meaningful choice for a formal study applicable to real-world languages with possibly top or bottom types, private types, record width subtyping, etc. In particular, the type constructor `p` is *not* upward-closed (and conversely `q` is not downward-closed), as used informally in the examples and defined for arbitrary variances in the following way:

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

3.2 The Algebra of Variances

If we know that $\overline{\sigma}$ `t` $\leq \overline{\sigma'}$ `t`, that is $\overline{\sigma}$ `t` $\prec_+ \overline{\sigma'}$ `t`, and the constructor `t` has variable $\overline{v\alpha}$, an inversion principle tells us that $\sigma_i \prec_{v_i} \sigma'_i$ for each i. But what if we only know $\overline{\sigma}$ `t` $\prec_u \overline{\sigma'}$ `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 $\overline{\sigma}$ `t` $\prec_u \overline{\sigma'}$ `t` then $\sigma_i \prec_{u.v_i} \sigma'_i$ holds. It is defined by the table in figure 3.2.

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 $\sigma \prec_v \tau$.[5] This reflexive, partial order is described by the lattice diagram in figure 3.2. All variances are smaller than $=$ and bigger than $\bowtie$.

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

Fig. 2. Variance composition table

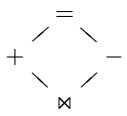

Fig. 3. Variance order diagram

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 $v \vee w \leq w$; 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 monotone: if $v \leq v'$ then $w.v \leq w.v'$ and $v.w \leq v'.w$.

We often 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 we have $\sigma_i \prec_{v_i} \sigma'_i$ for all i.

3.3 A Judgment for Variance of Type Expressions

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 4.

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 §4.4.

[5] 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 `type` $v\alpha$ `t`, it is safe to assume `type` $w\alpha$ `t`, since $\sigma \prec_w \sigma'$ implies $\sigma \prec_v \sigma'$, which implies σ `t` $\leq \sigma'$ `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 $\bowtie \leq +$.

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

Fig. 4. Variance assignment

A semantics for variance assignment. This syntactic judgment $\Gamma \vdash \tau : v$ corresponds to a semantic 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 (§3.1) 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 $\tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma'}]$ holds, then $\overline{\sigma} \prec_\Delta \overline{\sigma'}$ holds, where Δ is the minimal context such that $\Delta \vdash \tau : v$.*

3.4 Variance Annotations in ADTs

As a preparation for the difficult case of GADTs, we first present our approach in the well-understood case of algebraic datatypes. We exhibit a semantic criterion that justifies the correctness of a variance annotation; then, we propose an equivalent syntactic judgment. Of course, we recover the usual criterion that covariant variables should only occur positively.

In general, an ADT definition of the form

$$\mathtt{type}\ \overline{v\alpha}\ \mathtt{t} = \Big|_{c \in C}\ \mathtt{K}_c\ \mathtt{of}\ \tau^c[\overline{\alpha}]$$

cannot be accepted with any variance $\overline{v\alpha}$ $\mathtt{t}$. For example, the declaration ($\mathtt{type}\ v\alpha\ \mathtt{inv} = \mathtt{Fun\ of}\ \alpha \to \alpha$) is only sound when v is invariant. Accepting a variance assignment $\overline{v\alpha}$ determines the relations between closed types $\overline{\sigma}$ and $\overline{\sigma'}$ under which the relation $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma'}\ \mathtt{t}$ is correct.

In the definition of $+\alpha$ `exp` we justified the covariance of `exp` by the existence of a coercion function. We now formalize this idea for the general case. To check the correctness of $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}$ we check the existence of a coercion term that turns a closed value q of type $\overline{\sigma}\ \mathtt{t}$ into one of type $\overline{\sigma}'\ \mathtt{t}$ that is equal to q up to type information. We actually search for coercions of the form:

$$\mathtt{match}\ (q : \overline{\sigma}\ \mathtt{t})\ \mathtt{with}\ |_{c \in C}\ \mathtt{K}_c(x : \tau^c[\overline{\sigma}]) \to \mathtt{K}_c(x :> \tau^c[\overline{\sigma}'])$$

Note that erasing types gives an η-expansion of the sum type, *i.e.* this is really a coercion. Hence, such a coercion exists if and only if it is well-typed, that is, each cast of the form $(x : \tau^c[\overline{\sigma}] :> \tau^c[\overline{\sigma}'])$ is itself well-typed. This gives our semantic criterion for ADTs.

Definition 3 (Semantic soundness criterion for ADTs)
We accept the ADT definition of $\overline{v\alpha}\,\mathtt{t}$ *with constructors* $(\mathtt{K}_c\ \mathtt{of}\ \tau^c[\overline{\alpha}])_{c \in C}$ *if*

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

The syntactic criterion for ADTs. We notice that this criterion is exactly the semantic interpretation of the variance checking judgment (Definition 2): the type $\mathtt{type}\ \overline{v\alpha}\ \mathtt{t}$ is accepted if and only if the judgment $\overline{v\alpha} \vdash \tau^c : (+)$ is derivable for each constructor type $\tau^c[\overline{\alpha}]$.

This syntactic criterion coincides with the well-known alogrithm implemented in type checkers[6]: checking positive occurences of a variable α corresponds to a proof obligation of the form $\overline{v\alpha} \vdash \alpha : +$, which is valid only when α has variance $(+)$ or $(=)$ in Γ; checking negative occurences correspond to a proof obligation $\overline{v\alpha} \vdash \alpha : -$, *etc.* This extends seamlessly to irrelevant variables, which must appear only under irrelevant context $\overline{v\alpha} \vdash \alpha : \bowtie$—or not at all.

3.5 Variance Annotations in GADTs

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

$$\mathtt{|\ Prod\ :}\ \forall\beta\gamma.\ \beta\ \mathtt{exp} * \gamma\ \mathtt{exp} \to (\beta * \gamma)\ \mathtt{exp}$$

Instead of being just α `exp`, the codomain is now $(\beta * \gamma)$ `exp`. 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 GADTs support, such as Haskell and OCaml,

[6] One should keep in mind that this criterion suffers the usual bane of static typing, it can reject programs that do not go wrong: $\mathtt{type}\ -\alpha\ \mathtt{weird} = \mathtt{K}\ \mathtt{of}\ \alpha * \bot$. For more details, see the beginning of the §4 in the long version of this article.

and is reminiscent of the inductive datatype definitions of dependently typed languages.

However, for the formal study of GADTs, a different formulation based on equality constraints is preferred. We use the following equivalent presentation, already present in previous works [SP07]. We force the codomain of the constructor `Prod` to be α `t` again, instead of $(\beta * \gamma)$ `t`, by adding an explicit equality constraint $\alpha = \beta * \gamma$.

$$
\begin{array}{l}
\texttt{type } \alpha \texttt{ exp} = \\
\quad \mid \texttt{Val of } \exists \beta[\alpha = \beta].\, \beta \\
\quad \mid \texttt{Int of } [\alpha = \texttt{int}].\, \texttt{int} \\
\quad \mid \texttt{Thunk of } \exists \beta\gamma[\alpha = \gamma].\, \beta\ \texttt{exp} * (\beta \to \gamma) \\
\quad \mid \texttt{Prod of } \exists \beta\gamma[\alpha = \beta * \gamma].\, \beta\ \texttt{exp} * \gamma\ \texttt{exp}
\end{array}
$$

In the rest of the paper, we extend our former core language with such definitions. This does not impact 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 ADT type constructors. What needs to be changed, however, is the soundness criterion for checking the variance of type definitions

The correctness criterion. We must adapt our semantic criterion for datatype declarations (Definition 3) from simple ADTs to GADTs. Again, we check under which relations between $\overline{\sigma}$ and $\overline{\sigma}'$ the subtyping relation $\overline{\sigma}$ `t` $\leq \overline{\sigma}'$ `t` holds for some GADT definition $\overline{v\alpha}$ `t`.

The difference is that a constructor $\mathtt{K}_c$ that had an argument of type $\tau^c[\overline{\alpha}]$ in the simple ADT case, now has the more complex type $\exists\overline{\beta}[D[\overline{\alpha}, \overline{\beta}]].\tau^c[\overline{\beta}]$, for a set of existential variables $\overline{\beta}$ and a set of equality constraints D—of the form $(\alpha_i = T_i[\overline{\beta}])_{i\in I}$ for a family of type expressions $(T_i[\overline{\beta}])_{i\in I}$. Given a closed value q of type $\overline{\sigma}$ `t`, the coercion term is:

$$\texttt{match } (q : \overline{\sigma}\ \texttt{t}) \texttt{ with } |_{c\in C}\ \mathtt{K}_c(x : \tau^c[\overline{\rho}_c]) \to \mathtt{K}_c(x :> \tau^c[\overline{\rho}'_c])$$

We do not need to consider the dead cases: we only match on the constructors for which there exists an instantiation $\overline{\rho}_c$ of the existential variables $\overline{\beta}$ such that the constraint $D[\overline{\sigma}, \overline{\rho}]$, *i.e.* $\bigwedge_{i\in I} \sigma_i = T_i[\overline{\rho}_c]$, holds. To type-check this term, we need to find another instantiation $\overline{\rho}'_c$ that verifies the constraints $D[\overline{\sigma}', \overline{\rho}']$. This coercion type-checks only when $\tau^c[\overline{\rho}_c] \leq \tau^c[\overline{\rho}'_c]$ holds. This gives our semantic criterion for GADTs:

Definition 4 (Semantic soundness criterion for GADTs). *We accept the GADT definition of* `type` $\overline{v\alpha}$ `t` *with constructors* $(K_c$ `of` $\exists\overline{\beta}[D[\overline{\alpha}, \overline{\beta}]].\tau^c[\overline{\alpha}])_{c\in C}$*, if for all* c *in* C *we have:*

$$\forall\overline{\sigma}, \overline{\sigma}', \overline{\rho},\ \left(\overline{\sigma}\ \texttt{t} \leq \overline{\sigma}'\ \texttt{t} \wedge D[\overline{\sigma}, \overline{\rho}] \implies \exists\overline{\rho}',\ D[\overline{\sigma}', \overline{\rho}'] \wedge \tau[\overline{\rho}] \leq \tau[\overline{\rho}']\right) \qquad (\textsc{Req})$$

As for ADTs, this criterion ensures soundness: if, under some variance annotation, a datatype declaration satisfies it, then the implied subtyping relations are all expressible as coercions in the language, and therefore correct. Whereas the

simpler ADT criterion was already widely present in the literature, this one is less known; it is however present in the previous work of Simonet and Pottier [SP07] (presented as a constraint entailment problem).

Another way to understand this criterion would be to define constrained existential types of the form $\exists\overline{\beta}[D[\overline{\alpha},\overline{\beta}]].\tau[\overline{\beta}]$ as first-class types and, with the right notion of subtyping for those, require that $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}$ imply $(\exists\overline{\beta}[D[\overline{\sigma},\overline{\beta}]].\tau[\overline{\beta}]) \leq (\exists\overline{\beta}[D[\overline{\sigma}',\overline{\beta}]].\tau[\overline{\beta}])$. The (easy) equivalence between those two presentations is detailed in the work of Simonet and Pottier [SP07].

4 Checking Variances of GADT

4.1 Expressing Decomposability

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

$$\forall\sigma,\sigma',\rho_1,\rho_2,\\ \big(\sigma\ \mathtt{exp} \leq \sigma'\ \mathtt{exp} \wedge \sigma = \rho_1 * \rho_2 \implies \exists\rho_1',\rho_2', (\sigma' = \rho_1' * \rho_2' \wedge \rho_1 * \rho_2 \leq \rho_1' * \rho_2')\big)$$

We can substitute equalities and use the (user-defined) covariance to simplify the subtyping constraint σ `exp` $\leq \sigma'$ `exp` into $\sigma \leq \sigma'$:

$$\forall\sigma',\rho_1,\rho_2,\ \big(\rho_1 * \rho_2 \leq \sigma' \implies \exists\rho_1',\rho_2',\ (\sigma' = \rho_1' * \rho_2' \wedge \rho_1 \leq \rho_1' \wedge \rho_2 \leq \rho_2')\big) \quad (1)$$

This is the *upward closure* property mentioned in the introduction. The preceeding 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 following form, where the notation $(\overline{\rho} : \Gamma)$ simply classifies type vectors $\overline{\rho}$ that have exactly one type ρ_i for each variable in Γ:

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

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 $\rho_1 \leq \rho_1'$ and $\rho_2 \leq \rho_2'$ 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. 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 5 (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 following property holds:*

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

We use the symbol $\Vdash$ rather than $\vdash$ to highlight the fact that this is just a logic formula, not the semantic interpretation of a syntactic judgment—we will introduce one later in section 4.4.

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-monotone" 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.

4.2 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. In the general case, we require that this specific type constructor be upward-closed.

In general, 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.*, $\texttt{int} * \texttt{bool}$, which is not an instance[7] of $\beta * \beta$. 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 of $\beta \ \texttt{ref} * \beta \ \texttt{ref}$.

[7] 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 $\beta * (\beta\ \texttt{irr})$.

4.3 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* 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					

4.4 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 5. We also define its semantic interpretation $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$. The judgment and its interpretation were co-designed, so keeping the interpretation in mind is the best way to understand the subtleties of the inference rules. We use zipping, which requires correct variances, to merge sub-derivations into larger ones, so, in addition to decomposability, the interpretation also ensures that v is a correct

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

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

Fig. 5. Syntactic decomposablity

variance for τ under Γ. This subtlety is why we have two different properties for decomposability, $\Gamma \Vdash \tau : v \rightsquigarrow v'$ and $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$.

Definition 6 (Interpretation of syntactic decomposability)
We write $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ *for the conjunction of properties* $[\![\Gamma \vdash \tau : v]\!]$ *and* $\Gamma \Vdash \tau : v \rightsquigarrow v'$.

To understand the inference rules, 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 $\Gamma \Vdash \tau : v \rightsquigarrow v'$ (Definition 5) 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 semantic 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 that each subtype is decomposable from v to v', *with compatible witnesses*, that is, in an environment family $(\Gamma_i)_{i \in I}$ that can be zipped into a unique environment Γ.

Lemma 5 (Soundness of syntactic decomposability)
If the judgment $\Gamma \vdash \tau : v \Rightarrow v'$ *holds, then* $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ *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 §4.1 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 semantic criterion Req into a purely syntactic form.

Theorem 2 (Algorithmic criterion). *Given a variance annotation* $(v_i\alpha_i)_{i\in I}$ *and a constructor declaration of type* $(\exists\overline{\beta}\,[\bigwedge_{i\in I}\alpha_i = T_i[\overline{\beta}]]\,.\,\tau[\overline{\beta}])$*, the soundness criterion* Reg *for this constructor is equivalent to*

$$\exists\Gamma, (\Gamma_i)_{i\in I}, \quad \Gamma \vdash \tau : (+) \quad\wedge\quad \Gamma = \bigwedge_{i\in I}{\!\!\uparrow}\,\Gamma_i \quad\wedge\quad \forall i \in I,\, \Gamma_i \vdash T_i : v_i \Rightarrow (=)$$

The three parts of this formula can be explained to a user, as soon as the underlying semantic phenomenons (variable interference through zipping, and upward- and downward-closure) have been understood—there is no way to get around that. They are best read from right to left. The last part on the $(T_i)_{i\in I}$ is the decomposability requirement that failed in our example with `< m : int >`: the type expressions equated with a covariant variable should be upward-closed, and those equated with a contravariant one downward-closed. The zipping part checks that the equations do not create interference through shared existential variables, as in `type (+α, =β) eq = Refl of` $\exists\gamma[\alpha = \gamma, \beta = \gamma]$. Finally, the variance check corresponds to the classic variance check on argument types of ADTs. One can verify that in presence of a simple ADT, this new criterion reduces to the simple syntactic criterion.

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 corresponds to 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 $(? \vdash \alpha * \beta$ `ref` $: +)$ should return $(\alpha \mapsto \{+, =\}; \beta \mapsto \{=\})$. Defining those algorithmic variants of the judgments is routine. 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 possible 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.

5 Discussion

5.1 Upward and Downward Closure in a ML Type System

In the type system we have used so far, all type constructors but p and q are both upward and downward-closed. This simple situation, however, does not hold in general: richer subtyping relations will have weaker invertibility properties. As soon as a bottom type $\bot$ is introduced, for example, such that that for all type σ we have $\bot \leq \sigma$, downward-closure fails for all types – but $\bot$ itself. For example, products are no longer downward-closed: $\Gamma \vdash \sigma * \tau \geq \bot$ does not implies that $\bot$ is equal to some $\sigma' * \tau'$. Conversely, if one adds a top type $\top$, bigger than all other types, then most type are not upward-closed anymore.

In OCaml, there is no $\bot$ or $\top$ type[8]. However, object types and polymorphic variants have subtyping, so they are, in general, neither upward nor downward-closed. Finally, subtyping is also used in private type definitions, which were demonstrated in the example. Our closure-checking relation therefore degenerates into the following, quite unsatisfying, picture:

- no type is downward-closed because of the existence of private types;
- no object type but the empty object type is upward-closed;
- no arrow type is upward-closed because its left-hand-side would need to be downward-closed;
- datatypes are upward-closed if their components types are.

From a pragmatic point of view, the situation is not so bad; as our main practical motivation for finer variance checks is the relaxed value restriction, we care about upward-closure (covariance) more than downward-closure (contravariance). This criterion tells us that covariant parameters can be instantiated with covariant datatypes defined from sum and product types (but no arrow), which would satisfy a reasonable set of use cases.

5.2 A Better Control on Upward and Downward-Closure

There is a subtle design question here. Decomposability is fundamentally a `negative` statement on the subtyping relation, guaranteeing that some types have no supertypes of a different structure. It is therefore not necessarily preserved by addition to the subtyping relation – our system, informally, is `non-monotone` in the subtyping relation.

This means that 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 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 `downward-closed` (respectively `upward-closed`), that cannot later be made `private` (resp. `invisible`). Such declarations would be rejected if the defining type, for example an object type, already has subtypes (resp. supertypes), and would forbid further declarations of types below (resp. above) the defined type, effectively guaranteeing downward (resp. upward) closure.

[8] A bottom type would be admissible, but a top type would be unsound in OCaml, as different types may have different runtime representations. Existential types, that may mix values of different types, are constructed explicitly through a boxing step.

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 `upward-closed` or `downward-closed`.

5.3 Subtyping Constraints and Variance Assignment

We will now revisit our example of strongly typed expressions in the introduction. 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 exp =
  | Val of ∃β[α ≥ β]. β
  | Int of [α ≥ int]. int
  | Thunk of ∃βγ[α ≥ γ]. β exp * (β → γ)
  | Prod of ∃βγ[α ≥ β * γ]. β exp * γ exp
```

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 `exp` trivially satisfies the correctness criterion Req. 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. Note that this only works in the easy direction: while $\Gamma \vdash T_i : (+) \Rightarrow (+)$ is easy to check, $\Gamma \vdash T_i : (+) \Rightarrow (-)$ is just as hard as $\Gamma \vdash T_i : (+) \Rightarrow (=)$. In particular, an equality $(\sigma = \sigma')$ is already equivalent to a pair of inequalities $(\sigma \leq \sigma' \ \wedge \ \sigma \geq \sigma')$.

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

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

This variation on GADTs, using subtyping instead of equality constraints, has been studied by Emir *et al* [EKRY06] in the context of the `C♯` programming language—it is also expressible in Scala. However, using subtyping constraints in GADTs has important practical drawbacks in a ML-like language. While typed object-oriented programming languages tend to use explicit polymorphism and implicit subtyping, ML uses implicit polymorphism and explicit subtyping (when present). Thus in ML, equality constraints can be implicitly used while subtyping constraints must be explicitly used: unification-based inference favors bidirectional equality over unidirectional subtyping. This makes GADT definitions based on single subtyping constraints less convenient to use, because of

the corresponding syntactic burden, and this is probably the reason why the notion of GADTs found in functional languages use only equality constraints. Subtyping constraints need also be explicit in the type declaration, forcing the user out of the convenient "generalized codomain type" syntax.

Finally, weakening equality constraints into a subtyping constraint in one direction is not always possible; sometimes the strictly weaker expressivity of the type forbids important uses. One must then use an equality constraint, and use our decomposability-based reasoning to justify the variance annotation. Consider the following example:

```
type +α tree =
  | Node of ∃β[α = β list]. (β tree) list
let append : α tree * α tree → α tree = function
  | Node β1 (l1 : β1 tree list), Node β2 (l2 : β2 tree list) ->
      Node (List.append l1 l2)
```

We know that the two arguments of `append` have the same type α `tree`. When matching on the `Node` constructors, we learn that α is equal to both β_1 `list` and β_2 `list`, from which we can deduce that β_1 is equal to β_2 by non-irrelevance of `list`. The concatenation of the lists `l1` and `l2` type-checks because this equality holds. If we used a type system without the decomposability criterion, we would need to turn the constructor constraint into $\exists\beta[\alpha \geq \beta$ `list`$]$ to preserve covariance of α `tree` . We wouldn't necessarily have β_1 and β_2 equal anymore, so `(List.append l1 l2)`, hence the definition of `append` would not type-check. We would need decomposability-based reasoning to deduce, from $\alpha \geq \beta$ `list` and the fact that `list` is upward-closed, that in fact $\alpha = \beta'$ `list` for some β'.

This demonstrates that single subtyping constraints and our novel decomposability check on equality constraints are of incomparable expressivity: each setting handles programs that the other cannot type-check. From a theoretical standpoint, we think there is value in exploring the combination of both systems: using subtyping constraints rather than equalities, but also using decomposability to deduce stronger equalities when possible.

Note that while our soundness result directly transposes to a type-system with decomposability conditions on subtyping rather than equality constraints, our completeness result is special-cased on equality constraints. Completeness in the case of subtyping constraints is an open question.

6 Related Work

Simonet and Pottier [SP07] have studied GADTs in a general framework HMG(X), inspired by HM(X). They were interested in type inference using constraints, so considered GADTs with arbitrary constraints rather than type equalities, and considered the case of subtyping with applications to information flow security in mind. Their formulation of the checking problem for datatype declarations, as a constraint-solving problem, is exactly our semantic criterion and is not amenable to a direct implementation. Correspondingly, they did not encounter

any of the new notions of upward and downward-closure and variable interference (zipping) discussed in the present work. They define a dynamic semantics and prove that this semantic criterion implies subject reduction and progress. However, we cannot directly reuse their soundness result as they work in a setting where all constructors are upward- and downward-closed (their subtyping relation is atomic). We believe this is only an artifact of their presentation and their proof should be easily extensible to our setting.

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 surprisingly non-obvious correspondence, their system matches our presentation of GADTs with subtyping constraints and easier variance assignment, detailed in §5.3. They provide several usage examples and a full soundness proof using a classic syntactic argument. However, they do not consider the more delicate notions of decomposability, and their system therefore cannot handle some of the examples presented here.

7 Future Work

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 GADTs? Will the users be able to determine whether they want to enforce v-closure for a particular type they are defining?

Completeness of variance annotations with domain information. The way we present GADTs using equality constraints instead of the codomain syntax is well-known to practictioners, under the form of a "factoring" transformation where an arbitrary GADT is expressed as a simple ADT, using the equality GADT (α, β) `eq` as part of the constructor arguments to reify equality information.

This transformation does not work anymore with our current notion of GADTs in presence of subtyping. Indeed, all we can soundly say about the equality type (α, β) `eq` is that it must be invariant in both its parameters; using $(\alpha, T_i[\overline{\beta}])$ `eq` as part of a constructor type would force the paramter α to be invariant.

We think it would possible to 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 Seq 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 GADTs by equality types.

8 Conclusion

Checking the variance of GADTs 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 a natural semantic criterion.

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 extensions of GADTs 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. Abel, A.: Polarized subtyping for sized types. Mathematical Structures in Computer Science (2006); Goguen, H., Compagnoni, A. (eds.) Special issue on subtyping

EKRY06. Emir, B., Kennedy, A., Russo, C.V., Yu, D.: Variance and Generalized Constraints for C# Generics. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 279–303. Springer, Heidelberg (2006)

Gar04. Garrigue, J.: Relaxing the Value Restriction. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 196–213. Springer, Heidelberg (2004)

KR05. Kennedy, A., Russo, C.V.: 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), `http://research.microsoft.com/pubs/64040/gadtoop.pdf`

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

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

SR. Scherer, G., Rémy, D.: GADTs meet subtyping. Long version, available electronically, `http://gallium.inria.fr/~remy/gadts/`