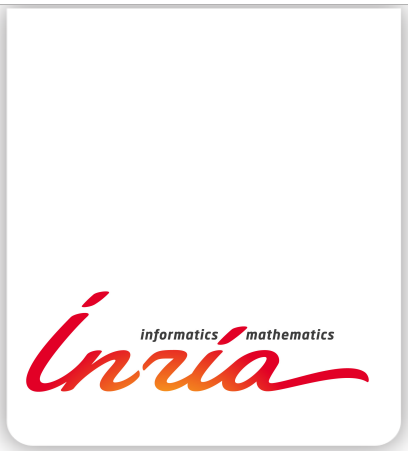
GADT meet subtyping

Gabriel Scherer, Didier Rémy

RESEARCH REPORT
N° 8114
October 2012
Project-Team Gallium

ISSN 0249-6399 ISRN INRIA/RR--8114--FR+ENG



GADT meet subtyping

Gabriel Scherer, Didier Rémy

Project-Team Gallium

Research Report n° 8114 — October 2012 — 33 pages

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.

Key-words: subtyping, datatypes, variance

Part of this work has been done at IRILL

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

GADT avec sous-typage

Résumé : Les types algébriques généralisés (*Generalized Algebraic Datatypes*, GADT) sont maintenant bien compris, mais leur ajout à un langage équipé de sous-typage nous réservait quelques surprises. Qu'est-ce qu'être covariant pour un paramètre de GADT ? La réponse s'avère difficile. Elle met en jeu des propriétés fines de la relation de sous-typage qui soulèvent d'intéressantes problématiques de conception de langage. Nous permettons des annotations de variance dans les définitions de GADT, étudions leur correction, et présentons un algorithme correct et complet pour les vérifier. Notre travail peut s'appliquer à un langage complet inspiré de ML et avec sous-typage explicite, tel que OCaml, ou même à des langages avec des contraintes générales de sous-typage.

Mots-clés : sous-typage, types de données, variance

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. This relaxation uses an intuitive subtyping argument: all occurrences of such type variables can be specialized to $\bot$, and anytime later, all covariant occurrences of the same variable (which are now $\bot$) can be simultaneously replaced by the same arbitrary type τ, which is always a supertype of $\bot$. This relaxation of the value-restriction is implemented in OCaml, where it is surprisingly useful. 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". 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.

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' *)
```

[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.

`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* 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 (eqab : (α, β) eq) : α → β =
  match eqab 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

[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.

[4]This counterexample is due to Jacques Garrigue.

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 $\texttt{t} \leq \tau$ but not $\texttt{t} \geq \tau$. 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`.

Unsound GADT covariance declarations would defeat the purpose of such private types: as soon as the user gets one element of the private type, she could forge values of this type, as illustrated by the code below.

```
module M = struct
  type file_descr = int
  let stdin = 0
  let open = ...
end : sig
  type file_descr = private int
  val stdin : file_descr
  val open : string -> (file_descr, error) sum
end

type +α t =
  | K : priv -> M.file_descr t

let get_eq : α t -> (α, M.file_descr) eq = function
  | K _ -> Refl

let forge : int -> M.file_descr =
  fun (x : int) -> cast (get_eq p) M.stdin
```

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 `M.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 $\texttt{type t} = \texttt{private}\ \tau$ is a new semi-abstract type smaller than τ ($\texttt{t} \leq \tau$ but $\texttt{t} \not\geq \tau$), 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

[5]Except types that are only defined privately in a module and not exported: they exist in

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 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 §4.3. 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 §4.3.

a "closed world" and we can check, for example, that they are never used in a `private` type definition.

On the importance of variance annotations

Being able to specify the variance of a parametrized datatype is important at abstraction boundaries: one may wish to define a program component relying on an *abstract* type, but still make certain subtyping assumptions on this type. Variance assignments provide a framework to specify such a semantic interface with respect to subtyping. When this abstract type dependency is provided by an encapsulated implementation, the system must check that the provided implementation indeed matches the claimed variance properties.

Assume the user specifies an abstract type

```
module type S = sig
  type (+α) collection
  val empty : unit -> α collection
  val app : α collection -> α collection -> α collection
end
```

and then implements it with linked lists

```
module C : S = struct
  type +α collection =
    | Nil of unit
    | Cons of α * α collection
  let empty () = Nil ()
end
```

The type-checker will accept this implementation, as it has the specified variance. On the contrary,

```
type +α collection = (α list) ref
let empty () = ref []
```

would be rejected, as `ref` is invariant. In the following definition:

```
let nil = C.empty ()
```

the right hand-side is not a value, and is therefore not generalized in presence of the value restriction; we get a monomorphic type, $?\alpha$ `t`, where $?\alpha$ is a yet-undetermined type variable. The relaxed value restriction [Gar04] indicates that it is sound to generalize $?\alpha$, as it only appears in covariant positions. Informally, one may unify $?\alpha$ with $\bot$, add an innocuous quantification over α, and then generalize $\forall\alpha.\bot$ `t` into $\forall\alpha.\alpha$ `t` by covariance—assuming a lifting of subtyping to polymorphic type schemes.

The definition of `nil` will therefore get generalized in presence of the relaxed value restriction, which would not be the case if the interface `S` had specified an invariant type.

Related work

When we encountered the question of checking variance annotations on GADT, we expected to find it already discussed in the literature. The work of Simonet and Pottier [SP07] is the closest we could find. It was done in the context of finding good specification for *type inference* of code using GADT, and in this context it is natural to embed some form of constraint solving in the type inference problem. From there, Simonet and Pottier generalized to a rich notion of GADT defined over arbitrary constraints, in presence of a subtyping relation, justified in their setting by potential applications to information flow checking.

They do not describe a particular type system, but a parametrized framework HMG(X), in the spirit of the inference framework HM(X). In this setting, they prove

a general soundness result, applicable to all type systems which satisfy their model requirements. We directly reuse this soundness result, by checking that we respect these requirements and proving that their condition for soundness is met. This allows us to concentrate purely on the static semantics, without having to define our own dynamic semantics to formulate subject reduction and progress results.

Their soundness requirement is formulated in terms of a general constraint entailment problem involving arbitrary constraints. Specializing this to our setting is simple, but expressing it in a form that is amenable to mechanical verification is surprisingly harder—this is the main result of this paper. Furthermore, at their level of generality, the design issues related to subtyping of GADT, in particular the notion of upward and downward-closed type constructors, were not apparent. Our article is therefore not only a specialized, more practical instance of their framework, but also raises new design issues.

The other major related work, by Emir, Kennedy, Russo and Yu [EKRY06], studies the soundness of having subtyping constraints on classes and methods of an object-oriented type system with generics (parametric polymorphism). Previous work [KR05] had already established the relation between the GADT style of having type equality constraints on data constructors and the desirable object-oriented feature of having type equality constraints on object methods. This work extends it to general subtyping constraints and develops a syntactic soundness proof in the context of a core type system for an object-oriented languages with generics.

The general duality between the "sums of data" prominent in functional programming and "record of operations" omnipresent in object-oriented programming is now well-understood. Yet, it is surprisingly difficult to reason on the correspondence between GADT and generalized method constraints; an application that is usually considered to require GADT in a functional style (for example a strongly-typed eval α `expr` datatype and its associated `eval` function) is simply expressed in idiomatic object-oriented style without specific constraints[6], while the simple `flatten` : $\forall\alpha$, α `list list` $\to$ α `list` requires an equality or subtyping constraint when expressed in object-oriented style.

These important differences of style and applications make it difficult to compare our present work with this one. Our understanding of this system is that a subtyping constraint of the form $X \leq Y$ is considered to be a negative occurrence of X, and a positive occurrence of Y; this means that equality constraints (which are conjunctions of a ($\leq$) constraint and a ($\geq$) constraints) always impose invariance on their arguments. Checking correctness of constraints with this notion of variance is simpler than with our upward and downward-closure criterion, but also not as expressive. It corresponds, in our work, to the idea of GADT with subtyping constraint mentioned in the introduction and that we detail in §4.3.

The design trade-off in this related work is different from our setting; the reason why we do not focus on this solution is that it requires explicit annotations at the GADT definition site, and more user annotations in pattern matching in our system where subtyping is explicitly annotated, while convertibility is implicitly inferred by unification. On the contrary, in an OOP system with explicit constraints and implicit subtyping, this solution has the advantage of user convenience.

We can therefore see our present work as a different choice in the design space: we want to allow a richer notion of variance assignment for type *equalities*, at the cost a higher complexity for those checks. Note that the two directions are complementary and are both expressed in our formal framework.

[6] There is a relation between this way of writing a strongly typed eval function and the "finally tagless" approach [Kis] that is known to require only simple ML types.

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

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

Figure 1: Subtyping relation

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 vα t =
  | K1 of τ1[α]
  | ...
  | Kn of τn[α]
```

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 $\overline{\sigma}$ `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 `type` $\overline{v\alpha}$ `t` simply means that the type constructor `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.

Finally, it is routine to show that the rules for reflexivity and transitivity are admissible, by pushing them up in the derivation until the base cases $\mathtt{b} \leqslant \mathtt{c}$, where they can be removed as $(\leqslant)$ is assumed to be reflexive and transitive. Removing reflexivity and transitivity provides us with an equivalent syntax-directed judgment having powerful inversion principles: 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, $\sigma_i \prec_{v_i} \sigma'_i$.

We insist that our equality relation $(=)$ is here a derived concept, defined from the subtyping relation $(\leq)$ as the "equiconvertibility" relation $(\leq \cap \geq)$; in particular, it is not defined as the usual syntactic equality. If we have both $b_1 \leqslant b_2$ and $b_1 \leqslant b_2$ in our relation on base types, for two distinct base types b_1 and b_2, we have $b_1 = b_2$ as types, even though they are syntactically distinct. This choice is inspired by the previous work of Simonet and Pottier.

On the restriction of atomic subtyping The subtyping system demonstrated above is called "atomic". If two head constructors are in the subtyping relation, they are either identical or constant (no parameters). Structure-changing subtyping occurs only at the leaves of the subtyping derivations.

While this simplifies the meta-theoretic study of the subtyping relation, this is too simplifying for real-world type systems that use non-atomic subtyping relations. In our examples using the OCaml type system, `private` type were a source of non-atomic subtyping: if you define $\mathtt{type}\ \alpha\ \mathtt{t_2} = \mathtt{private}\ \alpha\ \mathtt{t_1}$, the head constructors t_1 and t_2 are distinct yet in a subtyping relation. If we want to apply our formal results to the design of such languages, we must be careful to isolate any assumption on this atomic nature of our core formal calculus.

The aspect of non-atomic subtype relations we are interested in is the notion of v-closed constructor. We have used this notion informally in the first section (in OCaml, product types are $+$-closed); we now defined it formally.

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 $\overline{\sigma'}$.

In our core calculus, all type constructors are v-closed for any $v \neq \bowtie$, but we will still mark this hypothesis explicitly when it appears in typing judgments; this let the formal results be adapted more easily to a non-atomic type system.

It would have been even more convincing to start from a non-atomic subtyping relation. However, the formal system of Simonet and Pottier, whose soundness proof we ultimately reuse, restricts subtyping relations between (G)ADT type to atomic subtyping. 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.

2.2 The algebra of variances

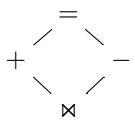
If we know that $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma'}\ \mathtt{t}$, that is $\overline{\sigma}\ \mathtt{t} \prec_+ \overline{\sigma'}\ \mathtt{t}$, and the constructor $\mathtt{t}$ has variable $\overline{v\alpha}$, an inversion principle tells us that for each i, $\sigma_i \prec_{v_i} \sigma'_i$. 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:

$v.w$	$=$	$+$	$-$	$\bowtie$	w
$=$	$=$	$=$	$=$	$\bowtie$	
$+$	$=$	$+$	$-$	$\bowtie$	
$-$	$=$	$-$	$+$	$\bowtie$	
$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$	
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 $\sigma \prec_v \tau$.[7] 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 $v.w \leq v'.w$).

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 `bool` $\leqslant$ `int` and `int` $\not\leqslant$ `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 `bool` $\to$ `bool` into one of type `int` $\to$ `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
```

[7] 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`: $\sigma \prec_w \sigma'$ implies $\sigma \prec_v \sigma'$, which implies $\sigma\ \texttt{t} \leq \sigma'\ \texttt{t}$. 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 $\bowtie \leq +$.

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

Figure 2: Variance assignment

A requirement for type soundness We say that the type $\texttt{type}\ \overline{v\alpha}\ \texttt{t}$ defined by the constructors $(\mathtt{K}_c\ \texttt{of}\ \tau^c[\overline{\alpha}])_{c\in C}$ 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*[8] 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.

In the context of higher-order subtyping [Abe06], where type abstractions are first-class and annotated with a variance ($\lambda v\alpha.\tau$), it is natural to present this check as a kind checking of the form $\Gamma \vdash \tau : \kappa$, where Γ is a context $\overline{v\alpha}$ of type variables *associated with variances*. For example, if $+\alpha \vdash \tau : \star$ is provable, it is sound to consider α covariant in τ. In the context of a simple first-order monomorphic type calculus, this amounts to a *monotonicity check* on the type τ as defined by [EKRY06]. Both approaches use judgments of a peculiar form where the *context* changes when going under a type constructor: to check $\Gamma \vdash \sigma \to \tau$, one checks $\Gamma \vdash \tau$ but $(\Gamma/-) \vdash \sigma$, where $\Gamma/-$ reverses all the variances in the context Γ (turns $(-)$ into $(+)$ and conversely). Abel gives an elegant presentation of this inversion $/$ as an algebraic operation on variances, a quasi-inverse such that $u/v \leq w$ if and only if $u \leq w.v$. This context change is also reminiscent of the *context resurrection* operation of the literature on proof irrelevance (in the style of [Pfe01] for example).

We chose an equivalent but more conventional style where the context of subderivation does not change: instead of a judgment $\Gamma \vdash \tau$ that becomes $(\Gamma/u) \vdash \sigma$ when moving to a subterm of variance u, we define a judgment of the form $\Gamma \vdash \tau : v$, that evolves into $\Gamma \vdash \sigma : (v.u)$. The two styles are equally expressive: our judgment $\Gamma \vdash \tau : v$ holds if and only if $(\Gamma/v) \vdash \tau$ holds in Abel's system—but we found that this one extends more naturally to checking decomposability, as will later be necessary. The inference rules for the judgment $\Gamma \vdash \tau : v$ are defined on Figure 2.

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.

[8] It turns out that this condition is not *necessary* and can be slightly weakened: we will discuss that later (3).

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.*

Proof:

Soundness: $\Gamma \vdash \tau : v$ *implies* $[\![\Gamma \vdash \tau : v]\!]$ By induction on the derivation. In the variable case this is direct. In the $\overline{\sigma}\ \mathtt{t}$ case, for $\overline{\rho}, \overline{\rho}'$ such that $\overline{\rho} \prec_\Gamma \overline{\rho}'$, we get $\forall i, \sigma_i[\overline{\rho}] \prec_{v.w_i} \sigma_i[\overline{\rho}']$ by inductive hypothesis, which allows to conclude, by definition of variance composition, that $(\overline{\sigma}\ \mathtt{t})[\overline{\rho}] \prec_v (\overline{\sigma}\ \mathtt{t})[\overline{\rho}']$.

Completeness: $[\![\Gamma \vdash \tau : v]\!]$ *implies* $\Gamma \vdash \tau : v$ By induction on τ; in the variable case this is again direct. In the $\overline{\sigma}\ \mathtt{t}$ case, given $\overline{\rho} \prec_\Gamma \overline{\rho}'$ such that $(\overline{\sigma}\ \mathtt{t})[\overline{\rho}] \prec_v (\overline{\sigma}\ \mathtt{t})[\overline{\rho}']$ we can deduce by inversion that for each variable α_i of variance w_i in $\tau[\overline{\alpha}]$ we have $\sigma_i[\overline{\rho}] \prec_{v.w_i} \sigma_i[\overline{\rho}']$, which allows us to inductively build the subderivations $\Gamma \vdash \sigma_i : v.w_i$. ■

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

Lemma 3 *If $\Gamma \vdash \tau : v$ and $\Gamma' \vdash \tau : v$ both hold, then $(\Gamma \vee \Gamma') \vdash \tau : v$ also holds.*

Corollary 1 (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 $\Delta \leq \Gamma$.*

Inversion of subtyping We have mentioned in 2.1 the inversion properties of our subtyping relation. From $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}$ we can deduce subtyping relations on the type parameters σ_i, σ_i'. This can be generalized to any type expression $\tau[\overline{\alpha}]$:

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 $\overline{\sigma} \prec_\Gamma \overline{\sigma}'$.*

Proof: The reverse implication, is a direct application of the soundness of the variance judgment.

The direct implication is proved by induction on $\tau[\overline{\alpha}]$. The variable case is direct: if $\alpha[\sigma] \prec_v \alpha[\sigma']$ holds then for Γ equal to $(v\alpha)$ we indeed have $v\alpha \vdash \alpha : v$ and $\sigma \prec_\Gamma \sigma'$.

In the $\overline{\tau}\ \mathtt{t}$ case, we have that $(\overline{\tau}\ \mathtt{t})[\overline{\sigma}] \prec_v (\overline{\tau}\ \mathtt{t})[\overline{\sigma}']$. Suppose the variance of $\overline{\alpha}\ \mathtt{t}$ is $\overline{w\alpha}$: by inversion on the head type constructor $\mathtt{t}$ we deduce that for each i, $\tau_i[\overline{\sigma}] \prec_{v.w_i} \tau_i[\overline{\sigma}']$. Our induction hypothesis then gives us a family of contexts $(\Gamma_i)_{i\in I}$ such that for each i we have $\Gamma_i \vdash \tau_i : v.w_i$. Furthermore, $\overline{\sigma} \prec_{\Gamma_i} \overline{\sigma}'$ holds for all Γ_i, which means that $\overline{\sigma} \prec_{\wedge_{i\in I}\,\Gamma_i} bs'$. Let's define Γ as $\wedge_{i\in I}\,\Gamma_i$. By construction we have $\Gamma \geq \Gamma_i$, so by monotonicity (Lemma 2) we have $\Gamma \vdash \tau_i : v.w_i$ for each i. This allows us to conclude $\Gamma \vdash \overline{\tau}\ \mathtt{t} : v$ as desired. ■

Note that this would work even for type constructors that are not v-closed: we are not comparing a $\tau[\overline{\sigma}]$ to any type τ', but to a type $\tau[\overline{\sigma}']$ sharing the same structure—the head constructors are always the same.

For any given pair $\overline{\sigma}, \overline{\sigma}'$ such that $\tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$ we can produce a context Γ such that $\overline{\sigma} \prec_\Gamma \overline{\sigma}'$. But is there a common context that would work for any pair? Indeed, that is the lowest possible context, the principal context Γ such that $\Gamma \vdash \tau : v$.

Corollary 2 (Principal inversion) *If Δ is principal for $\Delta \vdash \tau : v$, then for any type sequences $\overline{\sigma}$ and $\overline{\sigma}'$, the subtyping relation $\tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$ implies $\overline{\sigma} \prec_\Delta \overline{\sigma}'$.*

Proof: Let Δ be the principal context such that $\Delta \vdash \tau : v$ holds. For any $\overline{\sigma}, \overline{\sigma}'$ such that $\tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$, by inversion (Theorem 1) we have some Γ such that $\Gamma \vdash \tau : v$ and $\overline{\sigma} \prec_\Gamma \overline{\sigma}'$. By definition of Δ, $\Delta \leq \Gamma$ so $\overline{\sigma} \prec_\Delta \overline{\sigma}'$ also holds. That is, $\overline{\sigma} \prec_\Delta \overline{\sigma}'$ holds for any $\overline{\sigma}, \overline{\sigma}'$ such that $\tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$. ■

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 $\texttt{K of}\ \tau$ 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:

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

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. The idea is that we will force again the codomain to be exactly α `expr`, but allow additional type equations such as $\alpha = \beta * \gamma$ in this example:

$$\texttt{| Prod :}\ \forall \alpha.\ \forall \beta\gamma[\alpha = \beta * \gamma].\ \beta\ \texttt{expr} * \gamma\ \texttt{expr} \to \alpha\ \texttt{expr}$$

This restricted form justifies the name of *guarded* algebraic datatype. The $\forall \beta\gamma[D].\tau$ notation, coming from Simonet and Pottier, is a *constrained type scheme*: β, γ may only be instantiated with type parameters respecting the constraint D. Note that, as β and γ do not appear in the codomain anymore, we may equivalently change the outer universal into an existential on the left-hand side of the arrow:

$$\texttt{| Prod : } \forall\alpha.\ (\exists\beta\gamma[\alpha = \beta * \gamma].\ \beta\ \texttt{expr} * \gamma\ \texttt{expr}) \to \alpha\ \texttt{expr}$$

In the general case, a GADT definition for $\overline{\alpha}\ \texttt{t}$ is composed of a set of constructor declarations, each of the form:

$$\texttt{| K : } \forall\overline{\alpha}.\ (\exists\overline{\beta}[\overline{\alpha} = \overline{\sigma}[\overline{\beta}]].\,\tau[\overline{\alpha}, \overline{\beta}]) \to \overline{\alpha}\ \texttt{t}$$

or, reusing the classic notation,

$$\texttt{| K of } \exists\overline{\beta}[\overline{\alpha} = \overline{\sigma}[\overline{\beta}]].\,\tau[\overline{\alpha}, \overline{\beta}]$$

Without loss of generality, we can conveniently assume that the variables $\overline{\alpha}$ do not appear in the parameter type τ anymore: if some α_i appears in τ, one may always pick a fresh existential variable β, add the constraint $\alpha = \beta$ to D, and consider $\tau[\beta/\alpha]$. Let us re-express the introductory example in this form, that is, $\texttt{K of } \exists\overline{\beta}[\overline{\alpha} = \overline{\sigma}[\overline{\beta}]].\,\tau[\overline{\beta}]$:

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

If all constraints between brackets are of the simple form $\alpha_i = \beta_i$ (for distinct variables α_i and β_i), as for the constructor `Thunk`, then we have a constructor with existential types as described by Laüfer and Odersky [OL92]. If furthermore there are no other existential variables than those equated with a type parameter, as in the `Val` case, we have an usual algebraic type constructor; of course the whole type is "simply algebraic" only if each of its constructors is algebraic.

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 $\texttt{type } \overline{v\alpha}\ \texttt{t}$ 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 $\overline{\alpha}\ \texttt{t}$, 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 $\overline{\alpha}\ \texttt{t}, \overline{\alpha'}\ \texttt{t}'$, the types $(\tau_1 \to \tau_2)$, $\tau_1 * \tau_2$, $\overline{\sigma}\ \texttt{t}$ and $\overline{\sigma'}\ \texttt{t}'$ 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[9]: 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.

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:

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

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.

A remark on the non-completeness Note that while the criterion REQ-SP is *sound*, it is not *complete*—even in the simple ADT case.

For a constructor `K of` τ of $\overline{\sigma}$ `t`, the justification for the fact that, under the hypothesis $\overline{\sigma}$ `t` $\leq \overline{\sigma'}$ `t`, we should have $\tau[\overline{\sigma}] \leq \tau[\overline{\sigma'}]$ is the following: given a value v of type $\tau[\overline{\sigma}]$, we can build the value `K` v at type $\overline{\sigma}$ `t`, and coerce it to $\overline{\sigma'}$ `t`. We can then deconstruct this value by matching the constructor K, whose argument is of type $\tau[\overline{\sigma'}]$. But this whole computation, (`match` (`K` v `:>` $\overline{\sigma'}$ `t`) `with K` $x \ \to\ x$), reduces to v, so for value reduction to hold we need to also have $v : \tau[\overline{\sigma'}]$.

For this whole argument to work we need a value at type $\tau[\overline{\sigma}]$. In fact, if the type $\tau[\overline{\sigma}]$ is not inhabited, it can fail to satisfy REQ-SP and still be sound: this criterion is not complete. See the following example in a consistent system with an uninhabited type $\bot$:

```
type +α t =
  | T of int
  | Empty of ⊥ * (α → bool)
```

Despite α occurring in a contravariant position in the dead `Empty` branch (which violates the soundness criterion of Simonet and Pottier), under the assumption that the $\bot$ type really is uninhabited we know that this `Empty` constructor will never be used in closed code, and the contravariant occurrence will therefore never make a program "go wrong". The definition is correct, yet rejected by REQ-SP, which is therefore incomplete.

[9] 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.

Deciding type inhabitation in the general case is a very complex question, which is mostly orthogonal to the presence and design of GADT in the type system. There is, however, one clear interaction between the type inhabitation question and GADT. If a GADT $\overline{\alpha}$ `t` is instantiated with type variables $\overline{\sigma}$ that satisfy none of the constraints $D[\alpha]$ of its constructors `K of` $\exists\overline{\beta}[\overline{D}].\tau$, then we know that $\overline{\sigma}$ `t` is not inhabited. This is related to the idea of "domain information" that we discuss in the Future Work section (§5).

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:

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

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

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

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 $\rho_1 \leq \rho_1'$ 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-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.*

Our final decomposability criterion, given below in Figure 3, requires both correct variances and a decomposability property, so it will be neither monotonous nor anti-monotonous with respect to the context argument.
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.*, `int*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: β `ref` $* \beta$ `ref` is upward-closed (assuming the type constructor `ref` is invariant and upward-closed): if $(\sigma$ `ref` $* \sigma$ `ref`$) \leq \sigma'$, then by upward closure of the product, σ' is of the form $\sigma'_1 * \sigma'_2$, and by its covariance σ `ref` $\leq \sigma'_1$ and σ `ref` $\leq \sigma'_2$. Now by invariance of `ref` we have $\sigma'_1 = \sigma$ `ref` $= \sigma'_2$, and therefore σ' is equal to σ `ref` $* \sigma$ `ref`, which is an instance[10] of β `ref` $* \beta$ `ref`.

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

However, when a variable appears twice, the two witnesses corresponding to the two occurrences may not coincide. (Consider for example $\beta * \beta \leq$ `bool` $*$ `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

[10] 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.

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"[11] 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					

The following lemma uses zipping to merge together the results of the decomposition of several subterms $(\sigma_i)_i$ into a "simultaneous decomposition".

Definition 4 (Simultaneous decomposition) Given a context Γ, and families of type expressions $(\sigma_i)_{i\in I}$ and variances $(v_i)_{i\in I}$ and $(v'_i)_{i\in I}$, we define the following "simultaneous closure property" $\Gamma \Vdash (T_i : v_i \rightsquigarrow v'_i)_{i\in I}$ defined as :

$$\forall(\overline{\rho} : \Gamma), \overline{\sigma}', \ (\forall i \in I, \sigma_i[\overline{\rho}] \prec_{v_i} \sigma'_i) \implies \exists(\overline{\rho}' : \Gamma), \ \overline{\rho} \prec_\Gamma \overline{\rho}' \wedge (\forall i \in I, \sigma_i[\overline{\rho}'] \prec_{v'_i} \sigma'_i)$$

Lemma 5 (Soundness of zipping) *Suppose we have families of type expressions $(T_i[\overline{\beta}])_{i\in I}$, contexts $(\Gamma_i)_{i\in I}$ and variances $(v_i)_{i\in I}$ and $(v'_i)_{i\in I}$ such that $\curlywedge_{i\in I} \Gamma_i$ exists and for all i we have both $\Gamma_i \vdash T_i : v_i$ and $\Gamma_i \Vdash T_i : v_i \rightsquigarrow v'_i$. Then, we have $(\curlywedge_{i\in I} \Gamma_i) \Vdash (\sigma_i : v_i \rightsquigarrow v'_i)_{i\in I}$.*

Proof: Without loss of generality, we can consider that there are only two type expressions $T_1[\overline{\beta}]$ and $T_2[\overline{\beta}]$, and that the free variables $\overline{\beta}$ is reduced to a single variable β. Let w_1, w_2 be the respective variances of β in Γ_1, Γ_2. We know that $(w_1 \curlywedge w_2)$ exists and is equal to the variance w of the variable β in $\Gamma_1 \curlywedge \Gamma_2$.

Our further assumptions are $\Gamma_i \vdash T_i : v_i$ (**1**) and $\Gamma_i \Vdash T_i : v_i \rightsquigarrow v'_i$ (**2**) for i in $\{1, 2\}$. The expansion of (2) is:

$$\forall i \in I, \forall \rho, \sigma'_i, \quad T_i[\rho] \prec_{v_i} \sigma'_i \implies \exists \rho', \ \rho \prec_{\Gamma_i} \rho' \ \wedge \ T_i[\rho'] \prec_{v'_i} \sigma'_i \qquad (\mathbf{3})$$

Our goal is to prove $\Gamma_1 \curlywedge \Gamma_2 \Vdash (\sigma_i : v_i \rightsquigarrow v'_i)_{i\in I}$, which is equivalent to:

$$\forall \rho, \overline{\sigma}', \quad (\forall i \in I, (T_i[\rho] \prec_{v_i} \sigma'_i)) \implies \exists \rho', \ \rho \prec_\Gamma \rho' \ \wedge \ \forall i \in I, T_i[\rho'] \prec_{v'_i} \sigma'_i \quad (\mathbf{4})$$

Assume given ρ and (σ'_1, σ'_2) such that $T_1[\rho] \prec_{v_1} \sigma'_1$ (**5**) and $T_2[\rho] \prec_{v_2} \sigma'_2$ (**6**). Applying (3) with i equal to 1 and (5) ensures the existence of a ρ'_1 such that $\rho \prec_{w_1} \rho'_1$ (**7**) and $T_1[\rho'_1] \prec_{v_1} \sigma'_1$ (**8**). Similarly, there exists ρ'_2 such that $\rho \prec_{w_2} \rho'_2$ (**9**) and $T_2[\rho'_2] \prec_{v_2} \sigma'_2$ (**10**). To establish (4), it remains to build a single ρ' that satisfies $\rho \prec_w \rho'$ (**11**), $T_1[\rho'] \prec_{v_1} \sigma'_1$ (**12**) and $T_2[\rho'] \prec_{v_2} \sigma'_2$ (**13**), simultaneously. We reason by case analysis on w_1 and w_2 (restricted to the cases where the zip exists).

If both w_1 and w_2 are $(\bowtie)$, we take either ρ'_1 or ρ'_2 for: since w is $\bowtie$, we (11) trivially holds; Furthermore, $T[\rho'] = T[\rho'_1] = T[\rho'_2]$ by irrelevance of v_i and (1); therefore (12) and (13) follow from (8) and (10).

If only one of the w_i is $(\bowtie)$, we'll suppose that it is w_1. We then take ρ_2 for ρ'. Since $\bowtie_1 \curlywedge w_2$ is w_2, (11) follows from (9); Furthermore, $T[\rho'] = T[\rho'_1]$ by irrelevance of

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

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

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

Figure 3: Syntactic decomposablity

v_1 and (1) while $T_2[\rho'] = T_2[\rho'_2]$ holds by construction; Hence, as in the previous case, (12) and (13) follow from (8) and (10).

Finally, if both w_1 and w_2 are (=), then (7) and (9) implies $\rho'_1 = \rho = \rho'_2$. We take ρ for ρ' and all three conditions are obviously satisfied. ■

This lemma admits a kind of converse lemma stating completeness of zipping, that says that if $\Gamma \Vdash (T_i : v_i \rightsquigarrow v'_i)_i$ holds, then Γ is indeed related to a zip of contexts $(\Gamma_i)_i$ that pairwise decompose each of the $(T_i)_i$. However, the proof of completeness is more delicate and we prove it separately in §3.5.

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 rely on the zip soundness (Lemma 5) to merge sub-derivations into larger ones, so in addition to decomposability, the judgments simultaneously ensures that v is a correct 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.

Rule sc-Var is very similar to vc-Var, except that the condition $w \geq v$ is replaced by a stronger equality $w = v$. The reason why the variance-checking judgment has an inequality $w \geq v$ is to make it monotonous in the environment—as requested by its corresponding semantics criterion (Definition 2). Therefore, the condition $w \geq v$ is necessary for completeness—and admissible. On the contrary, the present judgment ensures, according the semantic criterion (Definition 5), that both the variance is correct (monotonous in the environment) and the type is decomposable, a property which is *anti-monotonic* in the environment (Lemma 4). Therefore, the semantics criterion $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ is invariant in Γ and, correspondingly, the variable rule must use a strict equality.

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 5 (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 6 (Soundness of syntactic decomposability)
If the judgment $\Gamma \vdash \tau : v \Rightarrow v'$ *holds, then* $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ *is holds.*

Proof: The proof is by induction on the derivation $\Gamma \vdash \tau : v \Rightarrow v'$ (**1**). Expanding $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$, we must show both $[\![\Gamma \vdash \tau : v]\!]$, or equivalently $\Gamma \vdash \tau : v$ (**2**) and $\Gamma \Vdash \tau : v \rightsquigarrow v'$ (**3**), which itself expands to:

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

Let $\overline{\rho}$, τ' be such that $\tau[\overline{\rho}] \prec_v \tau'$. We must exhibit a sequence $\overline{\rho}'$ such that $\overline{\rho} \prec_\Gamma \overline{\rho}'$ (**4**) and $\tau[\overline{\rho}'] \prec_{v'} \tau'$ (**5**). Cases where the derivation of (1) ends with SC-TRIV and SC-VAR cases are direct: take $\overline{\rho}$ and $(\dots, \sigma', \dots)$ for $\overline{\rho}'$, respectively.

In the remaining cases, the derivation ends with Rule SC-CONSTR and τ is of the form $\overline{\sigma}\ \mathtt{t}$.

- The v-closure assumption of the left premise ensures that τ' is itself of the form $\overline{\sigma}'\ \mathtt{t}$ for some sequence of closed types $\overline{\sigma}'$. By inversion on the variance $\overline{w\alpha}$ of the head constructor $\mathtt{t}$, we deduce $\sigma_i[\overline{\rho}] \prec_{v.w_i} \sigma_i'$ for all i (**6**).
- The middle premise is the zipping assumption on the contexts $\Gamma = \curlywedge_{i\in I} \Gamma_i$ (**7**).
- The right premises gives us subderivations $\Gamma_i \vdash \sigma_i : v.w_i \Rightarrow v'.w_i$. This implies $\Gamma_i \vdash \sigma_i : v.w_i$, for all i, which implies $\Gamma \vdash \overline{\sigma}\ \mathtt{t} : v$, *i.e.* (2). By induction hypothesis, this also implies $\Gamma_i \vdash \sigma_i : v.w_i$ (**8**) and $\Gamma_i \Vdash \sigma_i : v.w_i \Rightarrow v'.w_i$ for all i (**9**).

We may now apply zip soundness (Lemma 5) with hypotheses (7), (8) and (9), which gives us the simultaneous decomposition $\Gamma \Vdash (\sigma_i' : v.w_i \rightsquigarrow v'.w_i)_{i\in I}$. Expanding this property (Definition 4), we may apply to (6) to get to get a witness $\overline{\rho}'$ such that both $\overline{\rho}' \prec_\Gamma \overline{\rho}$, *i.e.* our first goal (4), and $(\forall i \in I, \sigma_i[\overline{\rho}'] \prec_{v'.w_i} \sigma_i')$, which implies $(\overline{\sigma}\ \mathtt{t})[\overline{\rho}'] \prec_{v'} \overline{\sigma}'\ \mathtt{t}$, *i.e.* our second goal (5). ∎

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. The proof requires several auxiliary lemmas; it is the subject of the next subsection.

3.5 Completeness of syntactic decomposability

We first show a few auxiliary results that will serve in the proof of zip completeness, and later, to reconnect our closure-checking criterion (Definition 5) with the full criterion of Simonet and Pottier (REQ-SP).

Lemma 7 (Intermediate value) *Let* $\tau[\overline{\alpha}]$ *be a type expression and* $\overline{\rho}_1$, $\overline{\rho}_2$, $\overline{\rho}_3$ *three type families such that* $\tau[\overline{\rho}_1] \leq \tau[\overline{\rho}_2] \leq \tau[\overline{\rho}_3]$ *and* $\overline{\rho}_1 \prec_\Gamma \overline{\rho}_3$ *holds for some* Γ. *Then, there exists a type family* $\overline{\rho}_2'$ *such that both* $\overline{\rho}_1 \prec_\Gamma \overline{\rho}_2' \prec_\Gamma \overline{\rho}_3$ *and* $\tau[\overline{\rho}_2'] = \tau[\overline{\rho}_2]$ *hold.*

Proof: We reuse the notations of the definition and assume $\tau[\overline{\rho}_1] \leq \tau[\overline{\rho}_2] \leq \tau[\overline{\rho}_3]$ (**1**) and $\overline{\rho}_1 \prec_\Gamma \overline{\rho}_3$ (**2**). We just have to exhibit ρ_2' such that both $\overline{\rho}_1 \prec_\Gamma \overline{\rho}_2' \prec_\Gamma \overline{\rho}_3$(**3**) and $\tau[\overline{\rho}_2'] = \tau[\overline{\rho}_2]$ (**4**) hold. Let Δ be the most general variance of τ, *i.e.* the lowest context such that $\Delta \vdash \tau : +$ (**5**) holds. By principal inversion (Corollary 2) applied to (1) twice thanks to (5), we have $\overline{\rho}_1 \prec_\Delta \overline{\rho}_2 \prec_\Delta \overline{\rho}_3$ (**6**).

If $\Delta \geq \Gamma$, the result is immediate, as $\overline{\rho}_1 \prec_\Gamma \overline{\rho}_2 \prec_\Gamma \overline{\rho}_3$ follows from (6) by anti-monotonicity and both (3) and (4) hold when we take $\overline{\rho}_2$ for $\overline{\rho}'_2$. Otherwise, we reason on each variable of the context Γ independently. We may assume, *w.l.o.g.*, that τ is defined over a single free variable α, and Γ and Δ are single variances v_Δ, v_Γ with $v_\Delta \not\geq v_\Gamma$. We reason by case analysis on the possible variances for (v_Δ, v_Γ), which are $\{(\_\_, =), (+, -), (-, +), (\bowtie, \_\_)\}$.

If v_Γ is $(=)$, the hypotheses (2) and (1) become $\rho_1 = \rho_3$ and $\tau[\rho_1] \leq \tau[\rho_2] \leq \tau[\rho_1]$, which implies $\tau[\rho_1] = \tau[\rho_2]$. Thus, taking ρ_1 for ρ'_2 satisfies (3) and (4).

If v_Δ is $(\bowtie)$, then by irrelevant of $\bowtie$ and (5), we have $\tau[\rho_1] = \tau[\rho_2]$. Thus, taking ρ_1 for ρ'_2 satisfies (3) and (4), as above.

Finally, the cases $(+, -)$ and $(-, +)$ are symmetric and we will only work out the first one, *i.e.* v_Δ is $(+)$ and v_Γ is $(-)$. From $\tau[\rho_1] \leq \tau[\rho_3]$ (which follows from (1) by transitivity) and (5), we have $\rho_1 \prec_{v_\Delta} \rho_3$, *i.e.* $\rho_1 \leq \rho_3$. Since the hypothesis (2) becomes $\rho_1 \geq \rho_3$, we have $\rho_1 = \rho_3$. Then, taking ρ_1 for ρ'_2, (3) trivially holds while (4) follows from (1). ■

The next lemma connects the monotonicity of the variance-checking judgment (checking variance at a lower context provides more information, and is therefore harder) and the anti-monotonicity of the decomposability formula (decomposing to a higher context provides more information, and is therefore harder): for a fixed type expression, the contexts at which you can check variance are higher than the contexts at which you can decompose. This property, however, only holds for non-trivial decomposability results (otherwise any context can decompose): we must decompose from a v to a v' that do not verify $v \geq v'$, and no variable of the typing context must be irrelevant.

Lemma 8 *Let $\tau[\overline{\alpha}]$ be a type and v and v' be variances such that $v \not\geq v'$. If $\Gamma \Vdash \tau : v \rightsquigarrow v'$ and Δ is the most general context such that $\Delta \vdash \tau : v$, then, for each non-irrelevant variable α of Δ, we have $\Gamma(\alpha) \leq \Delta(\alpha)$.*

Proof: We show that the Γ is lower than the lowest possible Δ, *i.e.* it is the most general context such that $\Delta \vdash \tau : v$ holds. Without loss of generality, we consider the case where τ has a single, non-irrelevant variable α, and Γ and Δ are singleton contexts over a single variance, respectively w_Γ and w_Δ, with $w_\Delta \neq (\bowtie)$.

Therefore, we assume $(w_\Gamma \alpha) \Vdash \tau : v \rightsquigarrow v'$ (**1**) and $(w_\Delta \alpha) \vdash \tau : v$ (**2**). We prove that $w_\Gamma \leq w_\Delta$. We actually show that for any ρ_1, ρ_2 such that $\rho_1 \prec_\Delta \rho_2$ we also have $\rho_1 \prec_\Gamma \rho_2$ (**3**).

From $\rho_1 \prec_\Gamma \rho_2$, we can deduce $\tau[\rho_1] \prec_v \tau[\rho_2]$ (**4**). Applying (1), we get a ρ' such that $\rho_1 \prec_\Gamma \rho'$ (**5**) and $\tau[\rho'] \prec_{v'} \tau[\rho_2]$ (**6**). We then reason by case analysis on $v \not\geq v'$, considering the different cases $\{(\bowtie, \_\_), (+, -), (-, +), (\_\_, =)\}$.

If v is $(\bowtie)$, the most general w_Δ is $\bowtie$, a case we explicitly ruled out: there is nothing to prove.

If v' is $(=)$, then (6) implies $\tau[\rho'] = \tau[\rho_2]$, and in particular $\rho' = \rho_2$; our goal (3) follows from (5).

If (v, v') is $(+, -)$ (we won't repeat the symmetric case $(-, +)$), then (4) and (6) becomes $\tau[\rho_1] \leq \tau[\rho_2]$ and $\tau[\rho_2] \leq \tau[\rho']$. Given (5), the intermediate value lemma (Lemma 7) ensures the existence of a ρ'' such that $\rho \prec_\Gamma \rho'' \prec_\Gamma \rho'$ and $\tau[\rho''] = \tau[\rho_2]$. From there, we deduce $\rho'' = \rho_2$, our goal (3) follows from (5), as in the previous case. ■

Finally, the following auxiliary lemmas will be useful in the proof of completeness.

Lemma 9 *If the principal variance w such that $(w\alpha) \vdash \tau[\alpha] : v$ holds is not the irrelevant variance $\bowtie$, then $\tau[\rho_1] = \tau[\rho_2]$ implies $\rho_1 = \rho_2$.*

Proof: Whatever v is, $\tau[\rho_1] = \tau[\rho_2]$ implies both $\tau[\rho_1] \prec_v \tau[\rho_2]$ and its converse $\tau[\rho_2] \prec_v \tau[\rho_1]$. This holds in particular for the principal variance w such that $(w\alpha) \vdash \tau[\alpha] : v$. Moreover, by principal inversion (Corrolary 2) applied twice, we have both $\rho_1 \prec_w \rho_2$ and $\rho_2 \prec_w \rho_1$. If w is distinct from $\bowtie$ this implies $\rho_1 = \rho_2$. ■

Lemma 10 *If $\Gamma \Vdash \tau : v \rightsquigarrow (=)$ holds for some Γ, and Δ is the most general context such that $\Delta \vdash \tau : v$ holds, then $\Delta \Vdash \tau : v \rightsquigarrow (=)$ also hold.*

Proof: Assume $\Gamma \Vdash \tau : v \rightsquigarrow (=)$, *i.e.*

$$\forall \overline{\rho\sigma}', \tau[\overline{\rho}] \prec_v \overline{\sigma}' \implies \exists \overline{\rho}',\ \overline{\rho} \prec_\Gamma \overline{\rho}' \wedge \tau[\overline{\rho}'] = \sigma' \qquad (\mathbf{1})$$

Assume that Δ is principal for $\Delta \vdash \tau : v$ (**2**). We show $\Delta \Vdash \tau : v \rightsquigarrow (=)$, *i.e.*

$$\forall \overline{\rho\sigma}', \tau[\overline{\rho}] \prec_v \overline{\sigma}' \implies \exists \overline{\rho}',\ \overline{\rho} \prec_\Delta \overline{\rho}' \wedge \tau[\overline{\rho}'] = \sigma' \qquad (\mathbf{3})$$

Let $\overline{\rho}$, $\overline{\sigma}'$ be such that have $\tau[\overline{\rho}] \prec_v \sigma'$ (**4**). By (1), there exists $\overline{\rho}'$ such that $\tau[\overline{\rho}'] = \sigma'$ (**5**). To prove (3), it only remains to prove that $\overline{\rho} \prec_\Delta \overline{\rho}'$ (**6**). Given (5), the inequality (4) becomes $\tau[\overline{\rho}] \prec_v \tau[\overline{\rho}']$ (**7**). Then (6) follows by principal inversion (Corrolary 2) applied to (7), given (2). ■

Lemma 11 *Let Δ be the most general context such that $\Delta \vdash \tau : v$ holds. If v is $(=)$, then only variances $(=)$ or $(\bowtie)$ may appear in Δ. If v is $\bowtie$, then only $(\bowtie)$ may appear in Δ.*

Proof: If v is $(\bowtie)$, the context Γ with all variances set to $\bowtie$ satifies $\Gamma \vdash \tau : v$ (as $[\![\Gamma \vdash \tau : v]\!]$ holds). By principality we have $\Delta \leq \Gamma$, so Δ also has only irrelevant variances as $(\bowtie)$ is the minimal variance.

If v is $(=)$, we handle each variable of the context independently, that is we can assume, *w.l.o.g.*, that τ has only one variable α. So Δ is of the form $(w\alpha)$ and we know that for any ρ, ρ' such that $\rho \prec_w \rho'$ we have $\tau[\rho] = \tau[\rho']$ (**1**). If w is not $(\bowtie)$, by lemma 9 applied to (1), we have $\rho = \rho'$ for any $\rho \prec_w \rho'$, which means that w is $(=)$. Summing up, we have shown that w is either $(\bowtie)$ or $(=)$. Reasoning similarly in the general case, any variance w of Δ is either $(\bowtie)$ or $(=)$. ■

We can now prove the converse of the zip soundness (Lemma 5) that is the core of the future proof of completeness of the decomposability judgment $\Gamma \vdash \tau : v \Rightarrow (=)$.

Theorem 2 (Zip completeness) *Given any context Γ, a family of type expressions $(T_i[\overline{\sigma}])_{i\in I}$ and a family of variances $(v_i)_{i\in I}$, if the simultaneous decomposition $\Gamma \Vdash (T_i : v_i \rightsquigarrow (=))_{i\in I}$ holds, then there exists a family of contexts $(\Gamma_i)_{i\in I}$ such that $\Gamma \leq \curlywedge_{i\in I} \Gamma_i$ and both $\Gamma_i \vdash T_i : v_i$ and $\Gamma_i \Vdash T_i : v_i \rightsquigarrow (=)$ hold for all i.*

If furthermore $\Gamma \vdash T_i : v_i$ holds for all i, then $\curlywedge_{i\in I} \Gamma_i$ is precisely Γ.

Proof: Let us assume the simultaneous decomposition $\Gamma \Vdash (T_i : v_i \rightsquigarrow (=))_{i\in I}$, which expands to:

$$\forall \overline{\sigma}', \overline{\rho},\ (\forall i, T_i[\overline{\rho}] \prec_{v_i} \sigma_i') \implies \exists \overline{\rho}',\ \overline{\rho} \prec_\Gamma \overline{\rho}' \wedge (\forall i, T_i[\overline{\rho}'] = \sigma_i') \qquad (\mathbf{1})$$

We construct a family of contexts $(\Gamma_i)_{i\in I}$ such that the following holds:

$$\Gamma \leq \bigwedge_i \Gamma_{i\in I} \ (\mathbf{2}) \qquad \forall i,\ \Gamma_i \vdash T_i : v_i \ (\mathbf{3}) \qquad \forall i, \Gamma_i \Vdash T_i : v_i \rightsquigarrow (=) \ (\mathbf{4})$$

where (4) is equivalent to

$$\forall i,\ \forall \sigma_i', \overline{\rho},\ \left(T_i[\overline{\rho}] \prec_{v_i} \sigma_i' \implies \exists \overline{\rho}',\ \overline{\rho} \prec_{\Gamma_i} \overline{\rho}' \wedge T_i[\overline{\rho}'] = \sigma_i'\right) \qquad (\mathbf{5})$$

The first step is to move from the entailment (1) of the form $\forall \overline{\rho}, (\forall i \in I, \dots) \implies (\exists \overline{\rho}', (\forall i \in I, \dots))$ to the weaker form $(\forall i \in I, \forall \overline{\rho}, \dots \implies \exists \overline{\rho}', \dots)$, but closer to (5). More precisely, we show that

$$\forall i,\ \forall \sigma_i', \overline{\rho},\ \left(T_i[\overline{\rho}] \prec_{v_i} \sigma_i' \implies \exists \overline{\rho}',\ \overline{\rho} \prec_{\Gamma} \overline{\rho}' \wedge T_i[\overline{\rho}'] = \sigma_i'\right) \qquad (\mathbf{6})$$

that is, $\forall i, \Gamma \Vdash T_i : v_i \rightsquigarrow (=)$ (**7**). Let i, σ_i', and $\overline{\rho}$ be such that $T_i[\overline{\rho}] \prec_{v_i} \sigma_i'$ (**8**). We show that there exists a $\overline{\rho}'$ such that $T_i[\overline{\rho}] = \sigma_i'$ (**9**) and $\overline{\rho} \prec_\Gamma \overline{\rho}'$ (**10**). Let us extend our type σ_i' to a family $\overline{\sigma}'$, defined by taking σ_j' equal to $T_j[\overline{\rho}]$ for j in $I \setminus \{i\}$. By construction, we have $(\forall j \in I, T_j[\overline{\rho}] \prec_{v_j} \sigma_j')$. Therefore, we may apply (1) to get a $\overline{\rho}'$ such that $\overline{\rho} \prec_\Gamma \overline{\rho}'$, *i.e.* our first goal (10), and $(\forall j, T_j[\overline{\rho}'] = \sigma_i')$, which implies our second goal (9) when j is i. This proves (6).

We now prove that we can refine this to have $\overline{\rho} \prec_{\Gamma_i} \overline{\rho}'$ for $(\Gamma_i)_{i\in I}$ such that $\Gamma \leq \bigwedge_i \Gamma_i$.

Let Δ_1 and Δ_2 be the most general contexts such both that $\Delta_1 \vdash T_1 : v_1$ and $\Delta_2 \vdash T_2 : v_2$ hold (**11**). Let (2)', (3)', and (4)' be obtained by replacing Γ_i's by Δ_i's in our three goals (2), (3), and (4). In fact (3)' is just (11). By Lemma 10 applied to (7) twice, given (11), we have both $\Delta_1 \Vdash T_1 : v_1 \rightsquigarrow (=)$ and $\Delta_2 \Vdash T_2 : v_2 \rightsquigarrow (=)$, that is, (4).

Hence Δ_1 and Δ_2 are correct choices for Γ_1 and Γ_2 if they also satisfy the goal (2)', *i.e.* $\Delta_1 \wedge \Delta_2 \geq \Gamma$. We now study when remaining goal (2)' holds and, when it does not, propose a different choice for Γ_1 and Γ_2 that respect all three goals.

W.l.o.g., we assume that I is reduced to $\{1, 2\}$ and that there is only one free variable β in T_1, T_2. Since we focus on a single variable of the context we name w_1 and w_2 the variances of β in Δ_1 and Δ_2, respectively. We now reason by case analysis on the variances w_1 and w_2.

If both of them are $\bowtie$, we have $\Delta_1 \wedge \Delta_2 = (\bowtie \beta)$, so we do not necessarily have $\Gamma \leq \Delta_1 \wedge \Delta_2$. Instead, we make a different choice for G_2. Namely, we pick Γ for Γ_2 and keep Δ_1 for Γ_1. As Δ_2 is $\bowtie$ we have $\Delta_2 \leq \Gamma_2$, so by monotonicity of the variance checking judgment we have $\Gamma_2 \vdash T_2 : v_2$ from (11), and we still have $\Gamma_2 \Vdash T_2 : v_2 \rightsquigarrow (=)$ from (7). Hence (3) and (4) are reestablished. Finally, we have $\Gamma_1 \wedge \Gamma_2 = \Gamma$, so in particular $\Gamma \leq \Gamma_1 \wedge \Gamma_2$, *i.e.* (2).

If only one of the w_i is $\bowtie$, we may assume, *w.l.o.g.*, that it is w_1. Then $\Delta_1 \wedge \Delta_2$ is Δ_2 and we only need to show that $\Gamma \leq \Delta_2$ (**12**). From (7), we have $\Gamma \Vdash T_2 : v_2 \Rightarrow (=)$. We then make a case analysis on v_2: if v_2 is not $(=)$, then by since Δ_2 is most general and $w_2 \neq \bowtie$, we may apply Lemma 8 to get (12); Otherwise, v_2 is $(=)$; Lemma 11 applied to (11) implies that Δ_2 is itself $(=\beta)$, and (12) trivially holds.

Finally, if none of the w_i is $\bowtie$, we first prove that they are both $(=)$. In fact, we only prove that w_1 is $(=)$ (**13**), as the other case follows by symmetry. To prove (13), we assume that ρ'' be such that $\rho \prec_{w_1} \rho''$ and we show that $\rho = \rho''$ (**14**) holds. By (11), we have $T_1[\rho] \prec_{v_1} T_1[\rho'']$. By reflexivity, we have $T_2[\rho] \prec_{v_2} T_2[\rho]$. We can use those two inequalities to invoke our simultaneous decomposability hypothesis (1) with $T_1[\rho'']$ for σ_1' and $T_2[\rho]$ for σ_2' to get a ρ' such that both $T_1[\rho'] = T_1[\rho'']$ and $T_2[\rho'] = T_2[\rho]$ hold. By Lemma 9 applied with (11), this implies both $\rho' = \rho''$ and $\rho' = \rho$, and therefore (14).

Therefore, the only remaining case is when w_1 and w_2 are both $(=)$. Then $\Delta_1 \curlywedge \Delta_2$ is $(=\beta)$, which is the highest single-variable context. So our goal (2)' trivially holds.

Of these several cases, one $(\bowtie, \bowtie)$ has $\Gamma_1 \curlywedge \Gamma_2 = \Gamma$ directly, and in the others Γ_1 and Γ_2 were defined as the most general contexts such that $\Gamma_1 \vdash T_1 : v_1$ and $\Gamma_2 \vdash T_2 : v_2$. If we add the further hypothesis that for each i, $\Gamma \vdash T_i : v_i$ holds, then by principality of the Γ_i, we have that $\Gamma_i \leq \Gamma$ for each i. This implies that we have $(\curlywedge_{i\in I} \Gamma_i) \leq \Gamma$ (when it is defined, $\curlywedge$ coincides with the lowest upper bound $\wedge$). By combination with (2), we get $(\curlywedge_{i\in I} \Gamma_i) = \Gamma$. ∎

Lemma 12 (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.

Proof: Assume $[\![\Gamma \vdash \tau : v \Rightarrow v']\!]$ holds for $v' \in \{=, \bowtie\}$, *i.e.* $\Gamma \vdash \tau : v$ (**1**) and $\Gamma \Vdash \tau : v \rightsquigarrow v'$ (**2**), which expands to

$$\forall(\overline{\rho} : \Gamma), \tau',\ \tau[\overline{\rho}] \prec_v \tau' \implies \exists(\overline{\rho}' : \Gamma),\ \overline{\rho} \prec_\Gamma \overline{\rho}' \ \wedge\ \tau[\overline{\rho}'] \prec_{v'} \tau' \qquad (\mathbf{3})$$

We show $\Gamma \vdash \tau : v \Rightarrow v'$ (**4**) by structural induction on τ

If $v \geq v'$ holds, then (4) directly follows from Rule SC-TRIV. This applies in particular when $v' = \bowtie$. Hence, we only need to consider the remaining cases where v' is $(=)$ and $v \not\geq v'$

We now reason by cases on τ.

Case τ is a variable α. (3) becomes

$$\forall\rho, \tau',\ \rho \prec_v \tau' \implies \exists\rho',\ \rho \prec_\Gamma \rho' \ \wedge\ \rho' = \tau'$$

This means that if $\rho \prec_v \tau'$ holds then $\rho \prec_\Gamma \tau'$ also holds: the variance $w\alpha \in \Gamma$ satisfies $v \geq w$. Since, the hypothesis (1) implies $v \leq w$, we have $v = w$. Therefore, (4) follows by Rule SC-VAR.

Case τ is of the form $\overline{\sigma}$ $\mathtt{t}$. By inversion, the derivation of (1) must end with rule VC-CONSTR, hence we have $\Gamma \vdash \sigma_i : v.w_i$ (**5**) for each $i \in I$ with $\Gamma \vdash \mathtt{type}\ \overline{w\alpha}\ \mathtt{t}$ (**6**).

Let us show that $\Gamma \Vdash (\sigma_i : v.w_i \rightsquigarrow =.w_i)_{i\in I}$ (**7**), *i.e.*

$$\forall\overline{\rho}, \overline{\sigma}',\ (\forall i \in I, \sigma_i[\overline{\rho}] \prec_{v_i} \sigma_i') \implies \exists\overline{\rho}',\ \overline{\rho} \prec_\Gamma \overline{\rho}' \wedge (\forall i \in I, \sigma_i[\overline{\rho}'] \prec_{v_i'} \sigma_i')$$

Let $\overline{\rho}$ and $\overline{\sigma}'$ be such that $\sigma_i[\overline{\rho}] \prec_{v_i} \sigma_i'$ holds for all i in I. From this and (5), we have $(\overline{\sigma}\ \mathtt{t})[\overline{\rho}] \prec_v \overline{\sigma}'\ \mathtt{t}$. By applitcation of (2), there exists $\overline{\rho}'$ such that $\overline{\rho} \prec_\Gamma \overline{\rho}'$ and $(\overline{\sigma}\ \mathtt{t})[\overline{\rho}'] \prec_= \overline{\sigma}'\ \mathtt{t}$. By inversion of subtyping , this implies $\sigma_i[\overline{\rho}'] \prec_{=.w_i} \sigma_i'$, for all i in I. This proves (7). We also note that the constructor $\mathtt{t}$ is v-closed (**8**).

To prove our goal (4), we construct a family $(\Gamma_i)_{i\in I}$ of contexts that satisfies $\curlywedge_{i\in I} \Gamma_i = \Gamma$ (**9**) and subderivations $\Gamma_i \vdash \sigma_i : v.w_i \Rightarrow =.w_i$ (**10**), since then the conclusion (4) follows by an application of rule SC-CONSTR with (6), (9), and (10).

We will handle separately the arguments σ_j that are irrelevant, *i.e.* when w_j is $\bowtie$, from the rest. Let $I_\bowtie$ be the set of indices with irrelevant variances and $I_{\not\bowtie}$ the others.

For any $i \in I_\bowtie$, $v.w_i$ and $=.w_i$ are both $\bowtie$ so the condition (10), which becomes $\Gamma_i \vdash \sigma_i : \bowtie \Rightarrow \bowtie$, is void of content $(\sigma_i[\overline{\rho}] \prec_\bowtie \sigma_i' \implies \exists\overline{\rho}',\ \sigma_i[\overline{\rho}'] \prec_\bowtie \sigma_i'$ is always true). More precisely, let Γ_i be the irrelevant context having only irrelevant variances. Then (10) follows by Rule SC-TRIV.

Since the decomposability constraints for $i \in I_\bowtie$ such that $w_i = \bowtie$ are trivial, (7) is equivalent to $\Gamma \Vdash (\sigma_i : v.w_i \rightsquigarrow =.w_i)_{i\in I_{\not\bowtie}}$ (**11**).

For each $i \in I_{\not\bowtie}$, $(=.w_i)$ equals $(=)$, so (11) becomes $\Gamma \Vdash (\sigma_i : v.w_i \rightsquigarrow (=))_{i \in I_{\not\bowtie}}$. By zip completeness (Theorem 2), there is a family $(\Gamma_i)_{i \in I_{\not\bowtie}}$ such that $\Gamma \leq \curlywedge_{i \in I_{\not\bowtie}} \Gamma_i$ and both $\Gamma_i \vdash \sigma_i : v.w_i$ and $\Gamma_i \Vdash \sigma_i : v.w_i \rightsquigarrow (=)$, *i.e.* $[\![\Gamma_i \vdash \sigma_i : v.w_i \Rightarrow (=)]\!]$ (**12**) hold for any $i \in I_{\not\bowtie}$, Furthermore, since we also have (5), we can strengthen our result into $\curlywedge_{i \in I_{\not\bowtie}} \Gamma_i = \Gamma$. By induction hypothesis applied to (12), we have (10) for $i \in I_{\not\bowtie}$.

We have two families of contexts over domains $I_{\bowtie}$ and $I_{\not\bowtie}$ that partition I; we can union them in a family $(\Gamma_i)_{i \in I}$ that has subderivations $\forall i \in I, \Gamma_i \vdash \sigma_i : v.w_i \Rightarrow v'.w_i$. As the contexts in $I_{\bowtie}$ are all irrelevant, they are neutral for the zipping operation: $\curlywedge_{i \in I} \Gamma_i$ is equal to $\curlywedge_{i \in I_{\not\bowtie}} \Gamma_i$, that is Γ. This proves (9) while (10) has already been proved separately for $i \in I_{\bowtie}$ and $i \in I_{\not\bowtie}$. ■

Remark 1 (Note (8)) *The head constructor* `t` *is closed in our system with atomic subtyping, but the situation is in fact a bit stronger than that: the statement of v-closure of* $\overline{\alpha}$ `t` *can be formulated in term of decomposability* $\Gamma \Vdash \overline{\alpha}$ `t` $: v \rightsquigarrow (=)$. *It is very close from our decomposability hypothesis* $\Gamma \Vdash \overline{\sigma}$ `t` $: v \rightsquigarrow (=)$, *but uses variables* $\overline{\alpha}$ *instead of full type expressions* $\overline{\sigma}$. *We conjecture that the decomposability hypothesis (with* $=$ *on the right) implies v-closure in a much larger set of subtyping systems that just atomic subtyping: it suffices that the subtyping relation is defined only in term of head constructors.*

3.6 Back to the correctness criterion

Remember the correctness criterion REQ-SP of Simonet and Pottier:

$$\forall \overline{\sigma}, \overline{\sigma}', \overline{\rho}, \quad (\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t} \wedge D[\overline{\sigma}, \overline{\rho}] \implies \exists \overline{\rho}',\ D[\overline{\sigma}', \overline{\rho}'] \wedge \tau[\overline{\rho}] \leq \tau[\overline{\rho}']) \qquad (\mathbf{1})$$

We now show how the closure judgment $\Gamma \vdash \tau : v \Rightarrow v'$ can be used to verify that this criterion holds: we will express this criterion in an equivalent form that uses the interpretation of our judgments.

The first step is to rewrite the property $\overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}$ using the variance annotation $\overline{v\alpha}$ of `t`. Again, we are taking the variance annotation for the datatype `t` as granted (this is why we can use it in this reasoning step), and checking that the definitions of the constructors of `t` are sound with respect to this annotation.

$$\forall \overline{\sigma}, \overline{\sigma}', \overline{\rho}, \quad ((\forall i, \sigma_i \prec_{v_i} \sigma'_i) \wedge D[\overline{\sigma}, \overline{\rho}] \implies \exists \overline{\rho}',\ D[\overline{\sigma}', \overline{\rho}'] \wedge \tau[\overline{\rho}] \leq \tau[\overline{\rho}']) \qquad (\mathbf{2})$$

Since, the constraint $D[\overline{\alpha}, \overline{\beta}]$ is a set of equalities of the form $\alpha_i = T_i[\overline{\beta}]$ (where T_i is a type), (2) is actually:

$$\forall \overline{\sigma}, \overline{\sigma}', \overline{\rho},\ (\forall i, \sigma_i \prec_{v_i} \sigma'_i) \wedge (\forall i, \overline{\sigma}_i = T_i[\overline{\rho}]) \implies \exists \overline{\rho}',\ (\forall i, T_i[\overline{\rho}'] = \sigma'_i) \wedge \tau[\overline{\rho}] \leq \tau[\overline{\rho}']$$

Substituting the equalities and, in particular, removing the quantification on the $\overline{\sigma}$, which are fully determined by the equality constraints $\overline{\sigma} = \overline{T}[\overline{\rho}]$, we get:

$$\forall \overline{\sigma}', \overline{\rho}, \quad (\forall i, T_i[\overline{\rho}] \prec_{v_i} \sigma'_i) \implies \exists \overline{\rho}',\ (\forall i, T_i[\overline{\rho}'] = \sigma'_i) \wedge \tau[\overline{\rho}] \leq \tau[\overline{\rho}'] \qquad (\mathbf{3})$$

By inversion (Theorem 1), we may replace the goal $\tau[\overline{\rho}] \leq \tau[\overline{\rho}']$ by the formula $\exists \Gamma, (\Gamma \vdash \tau : +) \wedge (\overline{\rho} \prec_\Gamma \overline{\rho}')$. Moreover, since $\Gamma \vdash \tau : +$ always for for some Γ, we may move this quantification in front. Hence, (3) is equivalent to:

$$\exists \Gamma,\ \bigwedge \begin{cases} \Gamma \vdash \tau : + \\ \forall \overline{\sigma}', \overline{\rho},\ (\forall i, T_i[\overline{\rho}] \prec_{v_i} \sigma'_i) \implies \exists \overline{\rho}',\ (\forall i, T_i[\overline{\rho}'] = \sigma'_i) \wedge \overline{\rho} \prec_\Gamma \overline{\rho}' \end{cases} \qquad (\mathbf{4})$$

We may recognize in second clause the simultaneous decomposability judgment (Definition 3) $\Gamma \vdash (T_i : v_i \rightsquigarrow =)_{i \in I}$. Hence, (4) is in fact:

$$\exists \Gamma,\ \Gamma \vdash \tau : +\ \wedge\ \Gamma \vdash (T_i : v_i \rightsquigarrow =)_{i \in I} \qquad (\mathbf{5})$$

Then, comes the delicate step of this series of equivalent rewriting:

$$\exists \Gamma, (\Gamma_i)_{i\in I},\ \bigwedge \begin{cases} \Gamma \vdash \tau : + \\ \Gamma = \curlywedge_{i\in I} \Gamma_i \ \wedge\ \forall i,\ \left(\Gamma_i \vdash T_i : v_i \ \wedge\ \Gamma_i \vdash T_i : v_i \rightsquigarrow =\right) \end{cases} \tag{6}$$

The reverse imiplication from (6) to (5) is is the zip soundness (Lemma 5).

The direct implication, from (5) to (6) is more involved: let Γ_0 be such that $\Gamma_0 \vdash \tau : +$. By zip completeness (Theorem 2), with the hypotheses of (4), there exists a family $(\Gamma_i)_{i\in I}$ satisfying the typing, zipping and decomposability of second line of (6) with $\Gamma_0 \leq \curlywedge_{i\in I} \Gamma_i$. We take $\curlywedge_{i\in I} \Gamma_i$ for Γ. Then, from $\Gamma_0 \leq \Gamma$ we get $\Gamma \vdash \tau : +$ by monotonicity (Lemma 2).

As a last step, the last conjuncts of (6) are equivalent to $\forall i, \Gamma_i \vdash T_i : v_i \Rightarrow (=)$ by interpretation of syntactic decomposability (Definition 5) and soundness and completeness of zipping (lemmas 6 and 12). Therefore, (6) is equivalent to:

$$\exists \Gamma, (\Gamma_i)_{i\in I}, \quad \Gamma \vdash \tau : (+) \quad \wedge \quad \Gamma = \bigcurlywedge_{i\in I} \Gamma_i \quad \wedge \quad \forall i \in I, \Gamma_i \vdash T_i : v_i \Rightarrow (=) \tag{7}$$

which is our final criterion.

Pragmatic evaluation of this 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 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 here. 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 Upward and downward closure in a ML type system

In the type system we have used so far, *all types* are both upward and downward-closed. Indeed, thanks to the simplicity of our subtyping relation, we have a very strong inversion principle: two ground types in a subtyping relation necessarily have exactly the same structure. We have therefore completely determined a sound variance check for a simple type system with GADT.

This simple resolution, 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 most types. For example, products are no longer downward-closed: $\Gamma \vdash \sigma * \tau \geq \bot$ does not imply 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[12]. However, object types and polymorphic variant have subtyping, so they are, in general, neither upward nor downward-closed. Finally, subtyping is also used in private type definitions, that 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 reasonably large set of use cases.

4.2 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`.

Such "closure specifications" are part of the semantic properties of a type and would, as such, sometimes need to be exposed through module boundaries. It is important that the specification language for abstract types allow to say that a type is upward-closed (respectively downward-closed). These new ways to classify types raise some software engineering questions. When is it desirable to define types as `upward-closed`? The user must balance its ability to define semi-abstract version of the type against its use in a GADT—and potentially other type-system features that would make use of negative reasoning on the subtyping relation. We do not yet know how to answer this question and believe that more practice is necessary to get a clearer picture of the trade-off involved.

4.3 Subtyping constraints and variance assignment

We will now revisit our previous example, using the guarded existential notation:

[12] 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.

```
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
```

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[13]: 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)
```

We conjecture that moving from an equality constraint on the GADT type parameters to a subtyping constraint (bigger than, or smaller than, according to the desired variance for the parameter) is often unproblematic in practice. In the examples we have studied, such a change did not stop functions from type-checking—we only needed to add some explicit coercions.

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. This is related to practical implementation questions, as languages based on inference by unification tend to favor equality over subtyping, bidirectional coercions over unidirectional ones. 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

[13] Note that the formal proofs of the precedent section were, in some cases, specialized to the equality constraint. More precisely, our decomposability criterion is still sound when extended to arbitrary subtyping constraints, but its completeness is unknown and left to future work.

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.

On the relaxed value restriction Regarding the relaxed value restriction, which is our initial practical motivation to investigate variance in presence of GADT, there is also future work to be done to verify that it is indeed compatible with this refined notion of variance. While the syntactic proof of soundness of the relaxation doesn't involve subtyping directly, the "informal justification" for value restriction uses the admissibility of a global bottom type $\bot$ to generalize a covariant unification variable; in presence of downward-closed type, there is no such general $\bot$ type (only one for non-downward-closed types). We conjecture that the relaxed value restriction is still sound in this case, because the covariance criterion is really used to rule out mutable state rather than subtype from a $\bot$ type; but it will be necessary to study the relaxation justification in more details to formally establish this result.

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 §4.3, 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.

On the implementation side, we suspect that adding subtyping constraints to a type system that already supports GADT and private types should not require large engineering efforts (in particular, it does not implies supporting the most general forms of bounded polymorphism). Matching on a GADT α `t` already introduces local type equalities of the form $\alpha = T[\overline{\beta}]$ in pattern matching clauses. Jacques Garrigue suggested that adding an equality of the form $\alpha =$ `private` $T[\overline{\beta}]$ should correspond to GADT equations of the form $\alpha \leq T[\overline{\beta}]$, and lower bounds could be represented using the dual notion of `invisible` types. Regardless of implementation difficulties, in a system with only explicit subtyping coercion, such subtyping constraints would still require more user annotations.

Mathematical structures for variance studies There has been work on more structured presentation of GADT as part of a categorical framework ([GJ08] and [HF11]). This is orthogonal to the question of variance and subtyping, but it may be interesting to re-frame the current result in this framework.

Parametrized types with variance can also be seen as a sub-field of order theory with very partial orders and functions with strong monotonicity properties. Finally, we have been surprised to find that geometric intuitions were often useful to direct our formal developments. It is possible that existing work in these fields would allow us to streamline the proofs, which currently are rather low-level and tedious.

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, the equality type (α, β) `eq` cannot be accurately described by considering variation of each of its parameters independently. We would like to say that (α, β) `eq` $\leq (\alpha', \beta')$ `eq` holds as soon as $\alpha = \beta$ and $\alpha' = \beta'$. With the simple notion of variance we currently have, all we can soundly say about `eq` is that it must be invariant in both its parameters—which is considerably weaker. In particular, the well-known trick of "factoring out" GADT by using the `eq` type in place of equality constraint does not preserve variances: equality constraints allow fine-grained variance considerations based on upward or downward-closure, while the equality type instantly makes its parameters invariant.

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 $\overline{\sigma}$ `t` $\leq \overline{\sigma}'$ `t` to only cases where $\overline{\sigma}$ `t` and $\overline{\sigma}'$ `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.

[GJ08] Neil Ghani and Patricia Johann. Foundations for structured programming with gadts. In *Proceedings of Principles and Programming Languages (POPL), 2008*, pages 297–308, 2008.

[HF11] Makoto Hamana and Marcelo Fiore. A foundation for gadts and inductive families: dependent polynomial functor approach. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, 2011. URL: `http://www.cs.gunma-u.ac.jp/~hamana/Papers/dep.pdf`.

[Kis] Oleg Kiselyov. Typed tagless interpretations and typed compilation. URL: `http://okmij.org/ftp/tagless-final/index.html`.

[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`.

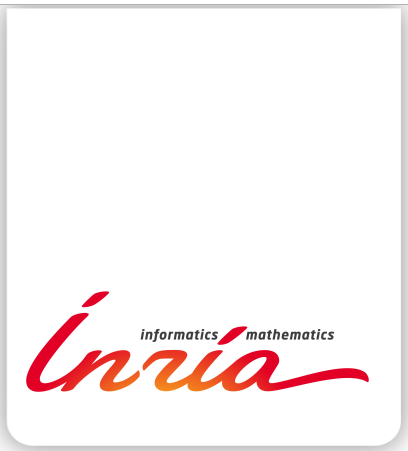
[MR09] Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009. URL: `http://gallium.inria.fr/~remy/modules/Montagu-Remy@popl09:fzip.pdf`.

[OL92] Martin Odersky and Konstantin Läufer. An extension of ML with first-class abstract types. In *ACM Workshop on ML and its Applications*, pages 78–91, June 1992. URL: `http://www.cs.luc.edu/laufer/papers/ml92.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.

Contents



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803