Open GADTs and Declaration-site Variance: A Problem Statement

Paolo G. Giarrusso
Philipps University Marburg

ABSTRACT

Generalized algebraic data types (GADTs) allow embedding extensible typed ASTs and transformations on them. Such transformations on typed ASTs are useful for code optimization in deeply embedded DSLs, for instance when using Lightweight Modular Staging (LMS). However, in Scala it is hard to make transformations for typed ASTs type-safe. Therefore, AST transformations in LMS are often not fully typechecked, preventing bugs from being caught early and without extensive testing.

We show that writing type-safe transformations in such embeddings is in fact not just hard, but impossible without using unsafe casts or significantly restricting extensibility: Declaration-site variance opens GADTs representing typed ASTs not only to desirable extensions, but also to extensions that introduce exotic terms. We make the problem concrete on an embedding of $\lambda_{<:}$ through covariant GADTs. We discuss solution approaches, and sketch a Scala extension to address this problem without either introducing unsafe casts or restricting extensibility.

We believe a complete solution would significantly ease writing transformations by allowing type-checking to verify them, and thus would ease their development.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs—*Inheritance, polymorphism, patterns*

Keywords

Type-safety; Scala; soundness; DSL embedding; lambda-calculus; Lightweight Modular Staging

1. INTRODUCTION

Type-preserving AST transformations have multiple applications. In particular, implementations of deeply embedded domain-specific languages (EDSLs) typically optimize programs written in these DSLs automatically, to achieve high performance with little manual effort. Such optimizations are implemented as part of the EDSLs; in particular, many optimizations are rewrite rules, which can be expressed as type-preserving transformations.

Since these optimizations are part of an EDSL, that is of a library, they can easily be extended with domain-specific optimizations; hence writing optimizers becomes possible not only for compiler authors but also for authors of libraries, making this task accessible to a wider audience, and making support for it more important.

In Scala, type-preserving transformations are used for instance in SQuOpt [3] or Lightweight Modular Staging (LMS) [8]. Such systems define a GADT named `Exp[T]`, such that ASTs of type `Exp[T]` encode *well-typed* object-level terms of type `T`. Type-preserving transformations should then have type $\forall$ `T. Exp[T]` $\Rightarrow$ `Exp[T]`; the typechecker should report when such a transformation is not type-preserving.

Writing type-preserving transformations is currently hindered by several problems. On the one hand, typechecker bugs or limitations in type inference prevent the compiler from recognizing type-preserving transformations. On the other hand, and more fundamentally, writing such transformations is *impossible*, unless we significantly restrict extensibility or use unsafe casts, as we explain in this paper.

To clarify problems with type-preserving transformations, we focus on an even simpler application of GADTs: writing typed embeddings of λ-calculi with interpreters. We choose this task because similar embeddings arise naturally in deeply embedded DSLs [8, 3], are for our purposes comparable to the one by Emir et al. [2], and represent a rather lightweight and natural encoding. Moreover, the problem with we demonstrate extends to type-preserving transformations (as shown in Sec. 3.1). Finally, interpreters are a standard application of GADTs in the literature (e.g. [2, 4]).

Our contributions are the following:

- We analyze a widely used embedding of $\lambda_{<:}$ with covariant GADTs and show that it admits exotic terms that make interpreters not type-safe.
- We demonstrate disadvantages of apparent solutions.
- We minimally sketch an extension to Scala and argue why it addresses the problem accurately while preserving extensibility.
- We sketch the relation between the problem and declaration-site variance.

All code examples have been compiled with Scala 2.10.2 and are available at `http://www.informatik.uni-marburg.de/`

Scala '13, Montpellier, France

ACM 978-1-4503-2064-1 ...$15.00
http://dx.doi.org/10.1145/2489837.2489842.

~pgiarrusso/research/gadts/, together with more extensive examples.

2. EMBEDDING $\lambda_{\to}$ WITH GADTS

As a background for our discussion, in this section we review a typed embedding of the simply-typed λ-calculus $\lambda_{\to}$:

```
trait Lambda {
  trait Exp[T]
  case class Const[T](t: T) extends Exp[T]
  case class App[S, T](fun: Exp[S ⇒ T], arg: Exp[S]) extends
    Exp[T]
  case class Fun[S, T](body: Exp[S] ⇒ Exp[T]) extends Exp[S ⇒ T]
}
```

Fun and App represent respectively abstraction and application using higher-order abstract syntax (HOAS) [6]. In addition, to allow easily embedding some terms, we allow arbitrary constants of the metalanguage to be used as terms in the object language, thanks to the constructor Const.

We define a few small example terms using this language:

```
8  trait LamExamples extends Lambda {
9    val term1 = App(Fun((x: Exp[Int]) ⇒ x), Const(1))
10   //val termWrong = Fun((x: Exp[Int]) ⇒ App(x, Const(1)))
11   //termWrong gives a type error
12 }
```

Our encoding is typed, thus, as observed in line 10, we cannot represent ill-typed terms.

We now use pattern matching [2] to define a *type-safe* interpreter, which is *well-typed* with type Exp[T] ⇒ T:

```
13 trait Interp extends Lambda {
14   def eval[T](term: Exp[T]): T =
15     term match {
16       case Const(t) ⇒ t
17       case App(fun, arg) ⇒
18         eval(fun) apply eval(arg)
19       case f: Fun[s, t] ⇒
20         (x: s) ⇒ eval(f.body(Const(x)))
21     }
22 }
```

When matching a Const node on line 16, the compiler can infer that t has type T and that the branch returns a value of type T. Similarly, on line 17 the compiler infers that fun has type Exp[s ⇒ T] and arg has type Exp[s], where s is an unspecified type variable, so that the branch returns again a value of type T.

Note that in line 19, we need a *type pattern* to bind type variables s and t (with T = s ⇒ t), because we need s in the type annotation on the next line.

This language is in fact richer than the $\lambda_{\to}$, because we are freely reusing Scala types. However, here and in our applications our goal is *not* to model accurately a λ-calculus, but to implement a core for EDSLs with *low effort*, so reusing the host language by writing a *metainterpreter* is appropriate.

2.1 Extensibility

In Scala, (generalized) algebraic data types are naturally open to extension. Hence, we can easily extend this language, for instance to support summing numbers:

```
trait Nums extends Lambda with Interp {
  case class Plus(a: Exp[Int], b: Exp[Int]) extends Exp[Int]

  override def eval[T](term: Exp[T]): T =
    term match {
      case p @ Plus(a, b) ⇒
        eval(a) + eval(b)
      case _ ⇒
        super.eval(term)
    }
}
```

All the examples we have seen up to now can be implemented without GADTs. We could have made eval a method of type Exp [4]; then we would decompose the eval methods shown above into implementations of eval in each subclass of Exp.

However, using GADTs allows easily adding new operations without modifying existing code, as shown in the following fragment:

```
trait BetaReduce extends Lambda {
  def betaReduce[T](term: Exp[T]): Exp[T] =
    term match {
      case App(Fun(f), arg) ⇒ f(arg)
      case e ⇒ e
    }
}
```

This snippet implements method betaReduce, which beta-reduces its argument if it is a redex (but not if it just contains a redex). Notably, we needed no change to existing modules.

To sum up, we can easily add new subtypes of Exp[T] and new operations on this type. Hence, what we have shown is a partial solution to the expression problem with defaults [8, 5]. In this encoding we cannot check exhaustiveness, that is whether we handle all possible node types, unless the transformation can handle unknown nodes using a default case. This is a problem for an interpreter, but we can ignore this problem since we focus on AST transformations: For typical type-preserving transformations, checking exhaustiveness is superfluous, since returning the input argument unchanged, as in betaReduce, is a valid default. Moreover, safer solutions are significantly heavier-weight.

3. EMBEDDING $\lambda_{<:}$

As we have seen, using GADTs we can embed the $\lambda_{\to}$ in Scala with high extensibility, because we can easily and modularly extend both the language and the set of operations.

In this section, we try to extend the embedded language with subtyping, that is to embed $\lambda_{<:}$ [7], and keep the embedding extensible. We will see that this is impossible.

To extend our embedding to $\lambda_{<:}$, we need to implement the rule of *subsumption*: if t: S and S <: T, then t: T. In our embedding, if t: Exp[S] and S <: T, we need to ensure t: Exp[T]. To this end, we can make Exp covariant, by replacing its declaration **trait** Exp[T] with **trait** Exp[+T]: then S <: T implies Exp[S] <: Exp[T]. In other words, we encode subtyping in the object language by reusing subtyping in the metalanguage. Beyond other advantages, this preserves the extensibility of our embedding.

However, if we make Exp covariant, extensions become able to introduce nonsensical or *exotic* terms, which have type Exp[T] for some T but don't correspond to object-language terms. Our interpreter remains (dynamically) type-correct on non-exotic terms, but fails to evaluate exotic terms with a dynamic type error, as we now demonstrate. For simplicity, we first reduce the language to the Const node:

```
trait Consts {
  trait Exp[+T]
  case class Const[T](t: T) extends Exp[T]
  def eval[T](term: Exp[T]): T =
    term match {
      case Const(t) ⇒ t
    }
}
```

This interpreter is correct on instances of `Const` itself, but not on instances of its subclasses, such as `unsoundTerm1` in the following snippet:

```
49 object Unsound extends Consts {
50   class UnsoundConst(t: String) extends Const[Any](t) with
51     Exp[Boolean]
52   val unsoundTerm1 = new UnsoundConst("")
53   val unsound1 = eval(unsoundTerm1)
54 }
```

Executing line 53 produces a `ClassCastException` without an explicit type cast, that is, a (dynamic) type error.

The problem arises because `UnsoundConst` is a subtype of both `Const[Any]` (and thus `Exp[Any]`) and `Exp[Boolean]`. This is legal because `Exp` is covariant and `Exp[U]` is a subtype of `Exp[Any]` for any `U`, so we `UnsoundConst` can be declared a subtype `Exp[U]` for any type `U` (in the example we have `U = Boolean`, but only `U != String` is required): it *refines the generic instantiation* of `Exp` (in `Const`) to `Exp[U]`. So, `UnsoundConst` is a subtype of `Exp[Boolean]`, but its instances contain, instead of a `Boolean`, a `String`. `unsoundTerm1` has type `UnsoundConst <: Exp[Boolean]`, so from `eval(unsoundTerm1)` type inference will produce `eval[Boolean](unsoundTerm1)`, having static type `Boolean`. In `eval` (line 46), `t` will be bound to `""` (of type `String`) and returned, whereas a value of type `T = Boolean` is expected, leading to a type error.

To prevent this type error, a sound typechecker should reject line 46 of our program. That line is accepted because the typechecker deduces `t: T`, but in our example `t = ""` is not an instance of `T = Boolean`. In general, when the pattern `Const(t)` matches an instance of `Exp[T]`, we can only deduce that it matches an instance of `Const[U](t) with Exp[T]` with `U >: T` and `t: U`; hence, we can't infer `t: T`, and writing a type-safe interpreter is *impossible*. We did not need to embed the full $\lambda_{<:}$ to show this impossibility. We simply need polymorphic AST nodes like `Const` and subtyping.

`UnsoundConst` is a valid Scala definition, and we believe this is reasonable in general—but it does not encode a valid object-language term and cannot be handled by our interpreter; while our interpreter correctly interprets $\lambda_{<:}$, it cannot be given type `Exp[T] ⇒ T`. Hence we believe that programmers should be able to choose whether refining generic instantiations (and so defining `UnsoundConst`) is allowed; our solution allows this choice, as sketched in Sec. 4.3.

Scalac's typechecker currently considers our interpreter (statically) *well-typed*, demonstrating an unsoundness problem (due to bug `https://issues.scala-lang.org/browse/SI-6944`). Many variations of this error are instead prevented statically; it is enough to reintroduce the branch handling `Fun` (lines 19–20) in the interpreter to observe a static type error (as demonstrated in the companion source code). Moreover, our focus is on the expressivity limitation: whether our interpreter is *well-typed* or not, it can trigger dynamic type errors—it is not *type-correct*.

3.1 Type-preserving transformations

We have shown that an interpreter for (a subset of) $\lambda_{<:}$ is not type-correct on exotic terms. As we mentioned, similar issues affect type-preserving transformations, as in the following minimal example:

```
55 def rebuild[T](term: Exp[T]): Exp[T] =
56   term match {
57     case Const(t) ⇒ Const(t)
58   }
```

The function `rebuild` takes apart a term in the language with only `Const` nodes and reconstructs a new but equal term. This typed transformation is clearly trivial but, similarly to `eval`, it is not type-safe if `term` is an instance of `UnsoundConst`.

The problem extends to useful transformations: We often faced such errors in our work on SQuOpt, for instance when trying to write a type-preserving code transformation implementing map fusion, as shown in the companion source.

Moreover, we conjecture the lack of a type-safe `rebuild` affects type-preserving transformations of covariant GADTs also in applications unrelated to language embeddings.

4. SOLUTION APPROACHES

In this section we survey possible solution approaches.

4.1 Ignoring the problem and using casts

Since we do not mean to write classes like `UnsoundConst`, we might just ignore the problem and use typecasts wherever type errors are detected, as in both LMS and SQuOpt.

However, in this way we give up some benefits of typechecking: one cannot ensure that transformations are type-preserving *statically*.

4.2 Reifying upcasts

Instead of making `Exp` covariant and cause the issues we discussed, we can encode subsumption as an explicit operation in ASTs (line 60). This approach does produce an adequate encoding, but it is rather inconvenient to use. An implicit conversion can reduce the need to apply this conversion explicitly (line 61):

```
59 trait LambdaUpcast extends Lambda {
60   case class Upcast[U, T <: U](e: Exp[T]) extends Exp[U]
61   implicit def upcast[U, T <: U](e: Exp[T]): Exp[U] = Upcast(e)
62 }
```

However, Scala type inference is too fragile and unpredictable for handling such implicit conversions (as shown in the companion code) or a robust variant.

Moreover, with this solution the input to transformations can contain `Upcast` at arbitrary locations, and we need to duplicate transformation code to transform such trees correctly. For instance, `betaReduce` becomes now:

```
63 trait BetaReduceSub extends LambdaUpcast {
64   def betaReduce[T](term: Exp[T]): Exp[T] =
65     term match {
66       case App(Fun(f), arg) ⇒ f(arg)
67       case App(Upcast(Fun(f)), arg) ⇒ f(arg)
68       case e ⇒ e
69     }
70 }
```

4.3 Restricting refinement

In our example, nodes like `UnsoundConst` should be forbidden. All extensions to AST classes such as `Const` can be prevented, by making them **`final`**. Since no class like `UnsoundConst` can be defined, our basic interpreter becomes type-safe and well-typed again:

```
71 trait LambdaInterpFinal {
72   trait Exp[+T]
73 
74   final case class Const[T](t: T) extends Exp[T]
75   final case class App[S, T](fun: Exp[S ⇒ T], arg: Exp[S])
76     extends Exp[T]
77   final case class Fun[S, T](body: Exp[S] ⇒ Exp[T])
78     extends Exp[S ⇒ T]
```

```
79
80   def eval[T](term: Exp[T]): T =
81     term match {
82       case Const(t) ⇒ t
83       case App(fun, arg) ⇒
84         eval(fun) apply eval(arg)
85       case f: Fun[s, t] ⇒
86         (x: s) ⇒ eval(f.body(Const(x)))
87     }
88 }
```

This interpreter for $\lambda_{<:}$ is almost identical to the one for $\lambda_{\rightarrow}$, except for the covariance annotation to `Exp` and for marking the case classes as **`final`**.

However, this solution prevents legitimate inheritance from AST classes, which at least was often useful when we implemented SQuOpt. Moreover, the restriction is too draconian. Compare the two definitions:

```
89 trait Compared extends Consts {
90   class UnsoundConst(t: String) extends Const[Any](t) with
91     Exp[Boolean]
92   class SoundConst(t: String) extends Const[Any](t)
93 }
```

`SoundConst` does not cause any problem, and can also be defined when `Exp` is invariant. `UnsoundConst` causes problems because it is a subtype of `Const[Any]`, thus of `Exp[Any]`; at the same time, it refines the generic instantiation of `Exp` to `Exp[Boolean]`. In our scenario, such a refinement should be forbidden. To forbid that, we propose to allow writing:

```
94 case class Const[T](t: T) extends Exp[=T]
```

Note that the only difference is the `=` in `Exp[=T]`. The type `Const[T] with Exp[U]` where `U <: T` and `U != T` would remain valid, but templates (classes, traits and objects) extending it (like `UnsoundConst`) would be forbidden. In other words, writing `Exp[=T]` forbids refining the instantiation of `Exp` in classes extending `Const`. The Scala specification (Sec. 5.1) already forbids creating such a template if `Exp` is not covariant, and we propose to extend this check.

Of course, this is just a sketch of the extension; its power-to-weight ratio in particular is yet unclear, and a formalization and proof of soundness (especially for typechecking pattern matching) are left as future work.

4.3.1 Declaration-site variance

We now explain why problematic classes like `UnsoundConst` can only be defined with declaration-site variance. With use-site variance one can encode covariant types, for instance `Exp` and `Const`, but not `UnsoundConst`.

A type `Base`, declared with declaration-site variance as **`trait`** `Base[+T]`, is conceptually equivalent to type `BaseVar`, declared with use-site variance as:

```
95 trait Base[T]; type BaseVar[T] = Base[_ <: T]
```

However, consider now:

```
96 class Derived[T] extends Base[T]
```

This fragment is valid with both use-site variance and declaration-site variance, but means different things. With use-site variance, it forbids writing `DerivedAgain` as in line 97; with declaration-site variance, it instead allows writing `DerivedAgain`:

```
97 class DerivedAgain[T] extends Derived[Any] with Base[Boolean]
```

With use-site variance and our extension, we can replace line 96 with **`class`** `Derived[T]` **`extends`** `Base[=T]` to forbid, if appropriate, writing `DerivedAgain` (or, in our original example, `UnsoundConst`). Hence, our extension allows again expressing a restriction that use-site variance already allowed.

5. RELATED WORK

Kennedy and Russo [4] first considered GADTs in object-oriented programming. They survey GADTs in functional languages (and related work in the area) and how to encode GADTs in OO languages, highlight cases where extra casts are needed and propose a language extension for C$^\sharp$, which allows implementing methods on parameterized classes (such as `Exp[T]`) which can only be called when `T` satisfies some constraints. Emir et al. [1] formalizes covariance in C$^\sharp$. Their system forbids refining generic instantiations, which restricts expressivity but would forbid `UnsoundConst`. We propose to control refinement of generic instantiations, not forbid it. Emir et al. [2] discusses GADTs in Scala and embeds $\lambda_{\rightarrow}$, but his encoding doesn't extend safely to $\lambda_{<:}$. Also their formalization forbids refining generic instantiations.

6. CONCLUSIONS

We have demonstrated that for sufficiently expressive EDSLs with subtyping, writing type-preserving AST transformations in Scala is impossible without reducing extensibility of the embedding or using type-unsafe casts, due to the interaction of GADTs and declaration-site variance. We have also analyzed a few alternatives and their limitations, and showed a minimal sketch of a language extension for Scala which we believe would solve the problem. We believe completing such a language extension would be a first step toward type-checked and thus safer type-preserving transformations.

Acknowledgments We thank Tillmann Rendel, Sebastian Erdweg, Eugene Burmako and the anonymous reviewers for their helpful comments. This work is supported by the European Research Council, grant #203099 "ScalPL".

References

[1] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C$^\sharp$ generics. In *ECOOP*, pages 279–303. Springer-Verlag, 2006.

[2] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP*, pages 273–298. Springer-Verlag, 2007.

[3] P. G. Giarrusso, K. Ostermann, M. Eichberg, R. Mitschke, T. Rendel, and C. Kästner. Reify your collection queries for modularity and speed! In *AOSD*, pages 1–12. ACM, 2013.

[4] A. Kennedy and C. V. Russo. Generalized algebraic data types and object-oriented programming. In *OOPSLA*, pages 21–40. ACM, 2005.

[5] M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *FOOL*, 2005.

[6] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI*, pages 199–208. ACM, 1988.

[7] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[8] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010.