Implementing Path-Dependent GADT Reasoning for Scala 3

Yichen Xu
Beijing University of Posts and Telecommunications
China
linyxus@bupt.edu.cn

Aleksander Boruch-Gruszecki
École polytechnique fédérale de Lausanne
Switzerland
aleksander.boruch-gruszecki@epfl.ch

Lionel Parreaux
Hong Kong University of Science and Technology
China
parreaux@cse.ust.hk

Abstract

Generalized Algebraic Data Types (GADT) are a popular programming language feature allowing advanced type-level properties to be encoded in the data types of a program. While Scala does not have direct support for them, GADT definitions can be encoded through Scala class hierarchies. Moreover, the Scala 3 compiler recently augmented its pattern matching capabilities to reason about such class hierarchies, making GADT-based programming practical in Scala. However, the current implementation can only reason about *type parameters*, but Scala's type system also features *singleton types* and *abstract type members* (collectively known as *path-dependent types*), about which GADT-style reasoning is also useful and important. In this paper, we show how we extended the existing constraint-based GADT reasoning of the Scala 3 compiler to also consider path-dependent types, making Scala's support for GADT programming more complete and bringing Scala closer to its formal foundations.

CCS Concepts: • **Software and its engineering → Data types and structures**; **Classes and objects**.

Keywords: Generalized algebraic data types, Type members, Path-dependent types, Singleton types, Scala

ACM Reference Format:
Yichen Xu, Aleksander Boruch-Gruszecki, and Lionel Parreaux. 2021. Implementing Path-Dependent GADT Reasoning for Scala 3. In *Proceedings of the 12th ACM SIGPLAN International Scala Symposium (SCALA '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3486610.3486892

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SCALA '21, October 17, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9113-9/21/10…$15.00
https://doi.org/10.1145/3486610.3486892

1 Introduction

Scala supports both flexible object-oriented class hierarchies and class-based pattern matching on them. This combination of features naturally gives rise to tricky typing problems, because pattern matching can uncover new typing information at runtime from the shapes of deconstructed data. Consider the definitions of the Expr base class and its children (Listing 1), and how the eval function (Listing 1b) performs pattern matching on Expr instances. These definitions will be explained in detail in the next section, but for now it suffices to remark that something non-trivial happens when type checking eval: in the IntLit case, we return an Int while a T is expected by the function signature. The Scala compiler has to reason that T is compatible with Int in this specific pattern matching branch.

```scala
sealed abstract class Expr[+A]
case class IntLit(i: Int)                 extends Expr[Int]
case class Add(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
case class Pair[B, C](a: Expr[B], b: Expr[C])
                                           extends Expr[(B, C)]
```

(a) Scala Definition of the Expr GADT.

```scala
def eval[T](e: Expr[T]): T = e match
  case IntLit(i)   => i
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Pair(a, b)  => (eval(a), eval(b))
```

(b) The eval function pattern matching on Expr.

Listing 1. A Scala GADT Example.

This is essentially the problem of *GADT reasoning*, which also arises in functional languages with support for generalized algebraic data types (GADTs). However, it is not straightforward to apply existing GADT reasoning approaches to Scala, because of its support for advanced type system features based on subtyping. And while Dotty (the new Scala 3 compiler) already integrates proper GADT reasoning for type parameters, it lacks supports for other aspects of Scala's type system, most notably *singleton types* and *abstract type members*, collectively known as *path-dependent types*.

In this paper, we propose an implementation of GADT reasoning for path-dependent types in Dotty. We lay out the current status of GADTs in Dotty, present our implementation for path-dependent GADTs, briefly discuss its soundness, and suggest future work in the same vein. It turns out that

we can reuse much of the existing logic for type-parameter GADT reasoning in our implementation. The specific contribution of this paper are the following:

- We show how to implement relatively efficient path-dependent GADT reasoning in Dotty and describe Dotty's GADT reasoning framework.
- We explain the interplay between path-dependent types and GADT reasoning, practically and theoretically.

2 Background

Functional programming languages traditionally define the data structures programs operate on through Algebraic Data Types (ADTs), which are essentially named sums of product types — i.e., an ADT is a type constructor with a set of possible data constructors, where each data constructor contains its own fields. Generalized Algebraic Data Types (GADTs) [Xi et al. 2003] extend the notion of plain ADTs by allowing each data constructor to refine the type of the data type being defined, thereby encapsulating additional information about the types involved in the construction of this data. This extra type information is then retrieved during pattern matching, which requires special reasoning capabilities from the type checker.

Scala is an object-oriented programming language at heart, whereby data types are defined through classes, rather than (G)ADTs. Thankfully, Scala's expressive class definitions can be used to model GADTs.[1] Listing 1a shows how to define a typical GADT in Scala: an Expr data type representing typed expressions. Expr is parameterized by some type A and defined as one of three constructors: integer literals and addition expressions, which only make sense when A is Int, and pairs, which similarly restrict A to the case where it is a tuple of two other types B and C. Since the types B and C do not appear in the parent Expr type, we say that they are *existentially quantified* — if we know that an Expr[T] value is constructed with the Pair constructor, then there *exist* some types a and b such that $T = (a, b)$. The use of a plus '+' sign in front of A additionally specifies that Expr is *covariant* in this type parameter, meaning that Expr[A] is a *subtype* of Expr[B] if A is itself a subtype of B.

Modelling GADTs through object-oriented class hierarchies is not enough to make them as useful as GADTs in languages where they are supported natively. One also needs a pattern matching construct which can leverage the typing information uncovered while deconstructing the corresponding GADT values. Consider Listing 1b as a concrete example, which shows how to evaluate an expression of type Expr[T], for any T. In this example, GADT reasoning makes the compiler derive the constraint T :> Int in the IntLit branch (i.e., T is a *supertype* of Int) allowing the case body to type check.

GADT constraints are inferred by Dotty based on the *cohabitation* of types. Formally, types A and B are said to "cohabit" if there exists a value x that can be given both types. For example, in the IntLit case of Listing 1b, we know that e is of type Expr[T] (based on its type signature) *and* of type IntLit (based on the matching pattern), implying that Expr[T] and IntLit cohabit. Scala's type system always ensures that all parent classes provide compatible type parameters to the shared base classes. In our case, this means that since IntLit inherits from Expr[Int], then we must have T :> Int for it to be compatible with Expr[T] (Expr is covariant, so this does not have to be T = Int). Hence, we can make this assumption while type checking the IntLit case. More formally, Parreaux et al. [2019] showed that the problem of GADT reasoning could be understood and justified through the lens of Scala's core *dependent object types* (DOT) calculus. This core calculus does not support type parameters, but a standard technique is to represent type parameters as path-dependent types, for which GADT reasoning can be explained.

Path-dependent types are an essential feature of Scala. A stable path p is a path made of "stable" values, such as function parameters and immutable object fields (for instance $p = param.field_1.field_2$). A path-dependent type is a type of the form p.type or $p.X$ — the former is a *singleton type* representing the type of the specific value of p, and the latter represents the specific type X that "lives" in p, where X is declared as a possibly abstract type member in one of the parent types of p. Since they depend on the runtime value of variables, path-dependent types can be viewed as a form of *dependent types*. Boruch-Gruszecki's original implementation of GADT reasoning for Scala 3 was limited to dealing with type parameters, leaving general path-dependent types out of the picture. This was unfortunate for two reasons: first, because path-dependent types provide a very general and powerful way of abstracting over types in Scala, and leaving them out makes Scala's support for GADTs patently incomplete; second, because as explained above our current understanding of GADT reasoning in Scala is *based* on the theory of path-dependent types, so not handling them in Scala 3 seems like a major missed opportunity.

It is possible to infer GADT-style constraints for type members and singleton types based on the typing information that arises from pattern matching, but since this functionality is currently missing from the compiler, many dependently-typed programming patterns that are provably sound cannot be type-checked as is. Listing 2 gives such an example, In this example, we refine the type of the Expr data type to also include the size information which is the number of

[1] In fact, Scala classes can model a *generalization* of GADTs, allowing the encoding of not only closed single-level data types, but also open and multi-level ones as well. Open data types are those for which it is possible to define new constructors later, in other parts of a program, and multi-level data types are those where the data constructors are organized as the leaves in a nested hierarchy of type constructors related by subtyping. Another quality-of-life improvement of Scala's approach to GADTs is that each data constructor has its own associated type constructor; for example, Pair[S, T] defined in Listing 1a is a type on its own, a subtype of Expr[(S, T)].

```
trait Expr[+X] { type Size <: Int }
case class IntLit(x: Int) extends Expr[Int] { type Size = 1 }
case class Add[N1 <: Int, N2 <: Int](
  e1: Expr[Int] sized N1,
  e2: Expr[Int] sized N2
) extends Expr[Int] { type Size = N1 + N2 + 1 }
```

(a) Definition of a sized variant of Expr.

```
def swap[X](tree: Expr[X]): Expr[X] sized tree.Size =
  tree match
    case IntLit(x) => IntLit(x)
    case Add(l, r) => Add(swap(r), swap(l))
```

(b) Definition of the swap function on sized Exprs.

```
type sized[X, N <: Int] = X & { type Size = N }
```

(c) Definition of the sized infix operator.

Listing 2. A piece of Scala code using both type parameter and type members GADT reasoning. It is sound but cannot type check in the current version of Dotty. Note that the example uses the compile time integer operation type '+' introduced by Scala 3 in package scala.compiletime.ops.int.

nodes in the expression tree, encoded through a Size *type member*.[2] For the IntLit case in Sublisting 2b, the compiler infers that X = Int via type parameter GADT reasoning, and should additionally know that tree.Size = 1 via type member reasoning, allowing IntLit(x) to be recognized as a value of type Expr[X] sized tree.Size. Without the support of type member GADT reasoning, one is currently required to painstakingly add many path-dependent type annotations to the program, which results in needless boilerplate and obscures the meaning of the program. On the theory side, Scala type parameters themselves are modeled in DOT through type members, and DOT allows GADT reasoning based on information implied by constraints over type members [Parreaux et al. 2019; Waśko 2020]. Thus, supporting GADT reasoning for type members will bring the compiler closer to its formal foundations.

3 Current Implementation of GADTs

In this section, we introduce the implementation of GADT constraint inference in the current compiler by answering two design questions: (1) how GADT constraints are stored in the compiler and (2) how GADT constraints are inferred. The two questions account for data structures and program logic of the implementation respectively. Note that the current GADT implementation only supports type parameters, and not path-dependent types. We present our implementation of path-dependent GADT reasoning in Section 4.

[2]This encoding is convenient as it does not "pollute" Expr with additional type parameters. Users who care about the sizes of manipulated expressions may use 'sized' refinements, as in 'Expr[A] sized N', which is a shorthand for the type refinement Expr[A] & {type Sized = N}, and users who are not interested in size information may simply use Expr[A], whereby the expression's size is existentially quantified.

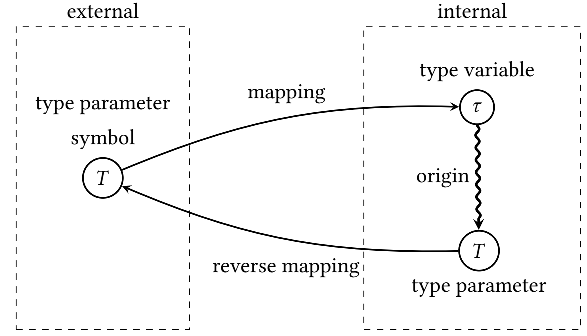

Figure 2. The mappings between constrained type parameters and their internal representations in GadtConstraint.

3.1 Storing Constraints for Type Parameters

In Dotty, GADT constraints are stored in the class GadtConstraint. Specifically, the class contains (1) the mapping between type parameters and internal representations, and (2) derived GADT constraints over the internal representations in an OrderingConstraint.

The basic idea of GadtConstraint is to map the type parameters to the internal representations OrderingConstraint operates on, and rely on OrderingConstraint to store and manage the constraints. OrderingConstraint is an immutable data structure that can record and manage constraints for type parameters, but it is designed for type inference and cannot be directly used for GADT constraints. Therefore, we implement GadtConstraint to adapt the functionalities provided by OrderingConstraint. GadtConstraint also utilizes the additional functionalities provided by ConstraintHandling, which keeps an instance of OrderingConstraint in a mutable field. For brevity, we will not present ConstraintHandling – it merely defines "surface" methods which require mutating the current constraint. In the following part, we first describe the structure of mappings used in GadtConstraint, which serve as *bridges* between the GADT-constrained external type parameters and the internal representations for constraint handling. Then, we explain the logic for the addition and retrieval of GADT bounds utilizing stored mappings.

3.1.1 Mappings.

The internal representations of the external constrained type parameters are the internal type parameters in the OrderingConstraint and its associated type variables. Type variables are a data structure for constraint inference, and they can be viewed as mutable trackers of type parameters in the OrderingConstraint. The GadtConstraint class will store the two-way mappings between the external type parameters and their internal representations. As illustrated in Figure 2, there will be two data members in the GadtConstraint to store the mappings: mapping and

```
sealed abstract class GadtConstraint:
  def addBound(sym: Symbol, bound: Type, isUpper: Boolean)(using
      Context): Boolean

  def bounds(sym: Symbol)(using Context): TypeBounds

  def isLess(sym1: Symbol, sym2: Symbol)(using Context): Boolean

  def addToConstraint(syms: List[Symbol])(using Context):
     Boolean
```

Listing 3. GadtConstraint interfaces for adding and retrieving constraints, and registering type parameters.

reverseMapping. The signature for mapping is SimpleIdentityMap[Symbol, TypeVar][3] that maps the symbol of the constrained type parameter to the internal type variable; reverseMapping is of type SimpleIdentityMap[TypeParamRef, Symbol] that maps the reference to an internal type parameters back to the external symbols.

3.1.2 Adding and Retrieving GADT Bounds. As listed in Listing 3, GadtConstraint provides interfaces for adding and retrieving GADT bounds. For adding bounds, the addBound function will first try to transform the type parameter referenced by the symbol or in the bounds to their internal representations, and then call the constraint handling logic. For retrieving bounds, bounds will return all GADT-inferred bounds for the type parameter, but the constraints of the ordering between other GADT-constrained type parameters will not be included, since including this information will require subtype checking, and may thus result in infinite loops interacting with TypeComparer. In cases where this information is necessary, isLess is used to retrieve the ordering constraints between two constrained type parameters. Both functions will first try to transform the external symbols into the internal representations, and then query bounds for them.

A key implementation concern here is that we must ensure to transform all constrained type parameters into their internal representations when adding and querying constraints for them, and transform the internal type variables back to the external types to prevent the leaking of internal representations.

3.1.3 Type Parameter Registration. Before we can actually record and retrieve constraints for type parameters, we explicitly tell the GadtConstraint which type parameters are constrainable by *registering* them. The addToConstraint method can register type parameters into the constrainer. Specifically, the method will create internal representations for the type parameters to be registered, and update the mappings to record the relationship between external types and internal variables. Additionally, it will also properly handle the inter-dependency between registered type parameters.

[3]SimpleIdentityMap is a linear map based on instance identity (using the eq method in Scala).

```
trait Container[+T]
case class IntList() extends Container[List[Int]]

def foo[T](e: Container[List[T]]): T = e match
  case IntList() => 0
```

Listing 4. Scala code illustrating GADT constraint inference.

To see a concrete example, consider we are registering a type parameter A with bounds $A <: T$, where T is a type parameter that is being registered together with A, or already being in the GADT constraint handler. Then the T should be substituted to its internal representation when we are storing bounds for A internally.

The type parameter registration is triggered in Typer. When we are typing a method or class definition with a type parameter list, we will register the type parameter introduced by them.

3.2 Inferring GADT Constraints

With the utilities for storing and handling GADT constraints for type parameters, we are now concerned with the way to actually inference the constraints. In Dotty, the class named PatternTypeConstrainer implements the GADT constraints inference, relying on the subtyping comparison logic implemented in TypeComparer. The inference logic is called when we are typing the pattern match, with the scrutinee and pattern types supplied in the arguments. The inference is done in two main steps: (1) firstly, since the GADT reasoning is actually based on the cohabitation of the scrutinee and pattern type, we will find out the subtyping relations necessary for the cohabitation condition to hold; (2) then, we call TypeComparer on these subtyping relations to get the constraints.

Listing 4 gives a concrete example for GADT constraint inference. Firstly, to satisfy the cohabitation condition between Container[List[T]] and IntList, the relation List[Int] <: List[T] must hold, since IntList is a case class [Odersky et al. 2016] that is impossible to extend and the type parameter is covariant [Castagna 1995]. Then we call TypeComparer on this subtyping relation, and will end up with the constraint T :> Int being inferred and recorded.

4 Implementing Path-Dependent GADT

In this section, we present our path-dependent GADT reasoning implementation. We will start with an overview of path-dependent GADT's use cases, followed by an explanation of how we enable the recording and handling of GADT constraints for type members. Then, we present a primitive implementation for recording equalities between singleton types based on pattern match. Finally, we discuss the relation with theory, the soundness of the implementation and prospect future work.

```
trait Tag { type T >: Int }
def f(e: { type T }): e.T = e match
  case e1: Tag => 0
```

(a) Deriving bound between type members.

```
trait Tag { type T }
def f(p: Tag, m: Expr[p.T]): p.T = m match
  case IntLit(i) => i
```

(b) Constraining path-dependent types.

```
trait IntTag { type T >: IntExpr }
def f[X](e: { type T <: Expr[X] }): X = e match
  case e1: IntTag => 0
```

(c) Type members as subtyping proofs.

Listing 5. Scala code illustrating the use cases of path-dependent GADT reasoning.

4.1 Use Cases

GADT constraints of path-dependent types are useful in many real-world scenarios, and the lack of them will prevent a certain number of sound Scala codes from compiling. Here we list a series of key use cases of path-dependent GADT reasoning.

Case 1. Deriving bounds between type members. Based on the pattern match, GADT constraints can be inferred for type members from the cohabitation of the scrutinee and pattern type. Consider the example given in Listing 5a. Since in the case body, we actually have e <: Tag & $\{T\}$, which implies e <: $\{T$:> Int$\}$, we can derive the GADT bound $e.T$:> Int for the path-dependent type $e.T$.

Case 2. Constraining path-dependent types. The path-dependent types addressed from extensible classes can be treated as *constrainable*. In other words, we can inference GADT constraints for them just like what we do for type parameters. Listing 5b gives such an example. The path-dependent type $p.T$ is constrainable, and we can derive the GADT constraint $p.T$:> Int for it.

Case 3. Type members as subtyping proofs. Additionally, the type members of an inhabited type can serve as subtyping proofs [Amin et al. 2016, 2014]. Therefore, we can derive more GADT constraints based on the subtyping relation. Inside the case body of Listing 5c, we know that the structural type IntTag & $\{T$ <: Expr[X]$\}$, which can be simplified as $\{T$:> IntExpr <: Expr[X]$\}$, is inhabited. Therefore, the type member T in the structural type can serve as proof for IntExpr <: Expr[X], leading to the GADT constraint X :> Int.

A real-world example. To better see the real-world benefits brought by path-dependent GADT reasoning, we provide an example in Listing 6. It implements a sized homogeneous vector Vec, and defines a constructor that turns its arguments *of arbitrary number* into a Vec instance. Listing 6d shows its usage: vec1 is constructed naively while vec2 is defined using the constructor. To implement such a constructor, we utilize the Tuple class and defines a type class TupOf to encode the shape and element type of a homogeneous (i.e., uniformly-typed) tuple. An instance p of type TupOf[T, A] serves as proof that the tuple type T has a size of p.Size and all its elements have type A. Thanks to Scala's implicits system [Odersky et al. 2016] (whose syntax was recast in terms of the 'given' and 'using' keywords in Scala 3), we can automatically derive type class instances for tuples. Then, Vec's apply constructor pattern matches on the automatically derived TupOf instance to extract information about the tuple's shape and element type, and utilize this information to convert the tuple into a Vec. For instance, Vec(0, 1, 2), which is syntax sugar for Vec(0 *: 1 *: 2 *: EmptyTuple), results in a vector typed as Vec[Int, 3]. GADT constraints of path-dependent types are crucial for the compilation of the example. Specifially, in the TupleOf.Empty case, we can derive the GADT constraint that p.Size = TupOf.Empty.Size = 0. This allows us to supply a VecNil : Vec[0, Nothing] for return type Vec[p.Size, A]. This example illustrates how path-dependent GADT reasoning can benefit type reasoning in real world scenarios, showing that our work improves the experience of dependently-typed programming in Scala.

```
abstract class TupOf[T, +A]:
  type Size <: Int
```

(a) Definition of the TupOf type class.

```
object TupOf:
  given Empty: TupOf[EmptyTuple, Nothing] with
    type Size = 0
  final given Cons[A, T <: Tuple, N <: Int]
    (using p: T TupOf A sized N): TupOf[A *: T, A] with
      val p0: T TupOf A sized N = p
      type Size = S[N]
```

(b) Implicit derivation for the TupOf type class.

```
enum Vec[N <: Int, +A]:
  case VecNil                             extends Vec[0, Nothing]
  case VecCons[N0 <: Int, A](head: A, tail: Vec[N0, A])
                                          extends Vec[S[N0], A]

object Vec:
  def apply[A, T <: Tuple]
    (xs: T)(using p: T TupOf A): Vec[p.Size, A] = p match
      case _: TupOf.Empty.type => VecNil
      case p1: TupOf.Cons[a, t, n] =>
        VecCons(xs.head, apply(xs.tail)(using p1.p0))
```

(c) Definition of the Vec class.

```
val vec1 = VecCons(1, VecCons(2, VecCons(3, VecNil)))
val vec2 = Vec(1, 2, 3)
```

(d) Usage example of the Vec class.

Listing 6. A real-world code example involving type classes and dependent programming. It relies on path-dependent GADT reasoning to pass type checking. The definition in Listing 6c uses the enum[4] syntax, and the example operates on the Tuple[5] class, both introduced in Scala 3.

[4] https://dotty.epfl.ch/docs/reference/enums/enums.html
[5] https://www.scala-lang.org/2021/02/26/tuples-bring-generic-programming-to-scala-3.html

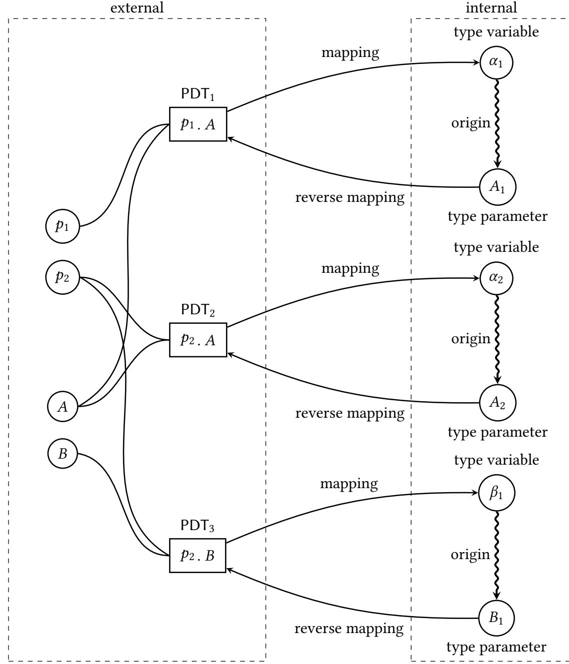

Figure 3. The mappings between constrained path-dependent types and their internal representations in Gadt-Constraint.

4.2 Extending GADT to Path-Dependent Types

4.2.1 Storing and Managing GADT Constraints. We implement the storage of path-dependent GADT constraints based on the original GadtConstraint, which only supports type parameters previously. Recap that in the original Gadt-Constraint, we maintain mappings between the symbols of type parameters and their internal representations. When it comes to path-dependent types, only recording the symbol of the type member will not be enough, since a path-dependent type is consitituted by not only its type member symbol, but also the prefix. Therefore, we will map the prefix together with the path-dependent type's type member symbol to the internal data structures.

Specifically, the path-dependent types are represented as a TypeRef(prefix: Type, designator: Designator). We use a mapping of type SimpleIdentityMap[SingletonType, SimpleIdentityMap[Symbol, TypeVar]] to map the path dependent types to their type variables. Here we use the Singleton-Type to represent the prefix, since in order to form a valid path-dependent type, the prefix must be a singleton. On the reverse side, we use a mapping of SimpleIdentityMap[Type-ParamRef, TypeRef] to map the internal type parameters references back to external path-dependent types.

With the these mappings, we can implement the methods for constraint recording and retrieval similarly for path-dependent types.

4.2.2 Constraint Inference. Path-dependent GADT reasoning can be classified into two main cases. Firstly, we would like to inference GADT constraints based on the type members of the scrutinee and the pattern, as in Listing 5a and 5c. Since the pattern and scrutinee will inhabit the same intersection type in the case body, we assume that the shared type members of the pattern and scrutinee are the same type as each other and then derive constraints based on the subtyping relations. Specifically, in Listing 5a, for $e : \{T\}$ and $e_1 : \{T <: \mathsf{Int}\}$, we extract constraints from the subtyping relations $e.T <: e_1.T$ and $e_1.T <: e.T$, getting $e.T <: \mathsf{Int}$ as a GADT bound.

Secondly, we would like to constrain path-dependent types where they can be GADT constrained while no pattern matching is performed on the path, as in Listing 5b. This can be achieved with existing GADT reasoning logic for type parameters, as long as the path-dependent types have internal representations created and are recognized by the GadtConstraint as constrainable, We achieve this by implementing the *just-in-time* type registration scheme in GadtConstraint, which will be explained in detail in the following section.

4.3 Registering Path-Dependent Types

Unlike type parameters, whose registration can be done when its introducing definition is being typed, the registration of path-dependent types can happen in more cases and can involve subtle issues. To handle path-dependent type registration, we implement two schemes: (1) registering when pattern matching and (2) just-in-time registration.

Registering when pattern matching. When doing pattern matching, we will scan the type member list of the scrutinee and pattern, and create internal representations for induced path-dependent types. For example, when constraining the pattern match in Listing 5a, we will register $e.T$ and $e_1.T$. Then we trigger the constraint inference logic based on the subtyping relations between path-dependent types we have just registered.

Just-in-time registration. While $p.T$ in Listing 5b will not be internalized by the previous scheme (since no pattern matching is performed on its path p), it is possible to record GADT constraints for them. We propose the just-in-time (JIT) scheme to register path-dependent types in such cases. To help understanding the JIT scheme better, we first describe the way to infer GADT bounds in TypeComparer. Internally, the comparer will examine deeply into the subtyping relation to verify it and record GADT bounds when possible. Before recording GADT bounds, it will first check whether GADT inferencing is enabled and the constrained type is registered in handler. Before our work, when Gadt-Constraint is asked whether an unseen path-dependent type is constrainable, it will return false to prevent the bounds from being recorded. With the JIT scheme enabled, we will check whether the path-dependent type can be constrained, create internal representations for it *just-in-time*, and return

```
trait Tag { type T }
trait T1
trait T2 extends T1
def f(p: Tag, e: { type T = p.T }): p.T = e match
  case _: ({ type T <: T1 } | { type T <: T2 }) => new T2 {}
```

Listing 7. Scala code snippet illustrating the issue of union types.

```
trait Tag { type S >: Int; type T = S }
def f(e: { type T }): e.T = e match
  case _: Tag => 0
```

Listing 8. Scala code snippet illustrating the issue of unbound patterns.

```
a match
  case b: X1 =>
    a match
      case c: X2 =>
        x match
          case y: X3 =>
            // Infer that a.type == b.type == c.type
            //   and x.type == y.type
            val t0: b.type = c
```

Listing 9. Inferring GADT constraints on singleton types.

true if possible. For example, in Listing 5b, when the comparer asks whether $p.T$ is registered, GadtConstraint will create type variables for $p.T$ and allow the GADT bounds to be recorded for it. While the scheme allows the recording of path-dependent GADT constraints in more cases, it brings a subtle issue when pattern-matching on union types.

Issue of the JIT scheme with union types. Listing 7 illustrates the issue of pattern-matching on union types, in which the scrutinee $e : T$ is matched against a union type $\{T <: T_1\} \mid \{T <: T_2\}$. To extract GADT constraints from the pattern matching, the compiler will branch into two components in the union type, derive GADT bounds in each of the directions, and comparer the constraints in each way to find the more general one. However, with the JIT scheme enabled, the type $p.T$ will be registered *after* branching into each direction, producing two different internal representations for the same $p.T$ type in the two derived constraints. The current compiler implementation cannot properly handle the case where two different internal type variables track the same external type, thus preventing Listing 7 from compiling. Therefore, we rewrite the constraint comparison logic to properly handle the case brought by the JIT scheme, by discovering these sibling type variables referring to the same type through inspecting the mappings.

Unbound patterns. Apart from the previous issue originating from the JIT scheme, we encountered another issue involving *unbound* patterns. when we implement our approach. Considering a concrete example, in the pattern matching of Listing 8, we will end up unifying $e.T$ to the type member S in the pattern. Ideally, when comparing Int to $e.T$ in the body, we will first retrieve the GADT bound that

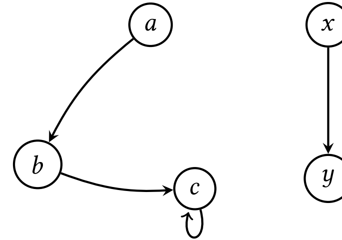

Figure 4. Visualization of the disjoint set data structure for recording singleton equalities in Listing 9.

$e.T$ is equal to the pattern's type member S in the pattern, and then find out that S is a supertype of Int. However, the pattern is unbound. So it is impossible to refer to its type member S. The naive implementation will leak the internal representation of S, which will confuse the comparer, and prevent the code from compiling. We take the solution to creating a skolem serving as a *placeholder* of the unbound pattern, and map all pattern's type members onto it. This enables us to reference type members of a unbound pattern.

4.4 Recording Equality between Singletons

Apart from deriving constraints for the type members of the scrutinee and pattern, it is also safe to assume that the scrutinee and the pattern are referring to the same value in the case body, implying that their singleton types are equal. Listing 9 gives an example of GADT constraint inference over singleton types. In this snippet, a is matched against b, then c, implying that the singleton types from a, b and c are all equal. In other words, the three symbols refer to the same value in the case body. Similar reasoning applies for x and y.

We utilize the disjoint set data structure to record the equalities in a simple yet effective manner. The constraints between singleton types are all equalities, resulting a series of equivalent classes of singletons, which can be efficiently stored and queried using the disjoint union data structure. Figure 4 visualizes the state of the disjoint set within the innermost match case. This allows the type comparer to recognize that $c <: b.type$.

5 Experimental Evaluation

In this section, we empirically evaluate our implementation in two aspects: conservativeness and performance.

To verify that our implementation is conservative, in the sense that it neither breaks existing compiling Scala code nor unexpectedly makes unsound code compile, we run all compilation unit tests on the modified compiler. The test suite includes both positive and negative tests and covers a broad range of functionalities and use cases of the compiler. Our branch is able to pass all of the 8345 unit tests, suggesting that our implementation is conservative.

[6]https://github.com/lampepfl/bench

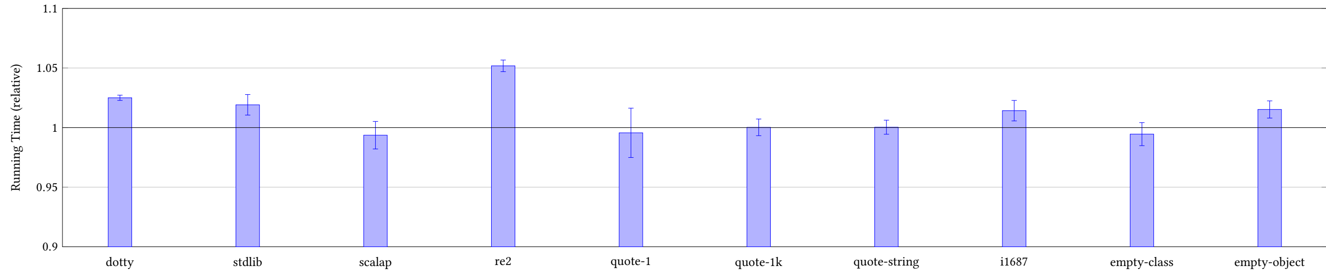

Figure 5. Performance impact of our implementation. It shows the relative running time of 10 tests in the Dotty benchmark[6] on our branch compared to the master.

To investigate the performance impact of our modification, we run the test suites in the Dotty benchmark and inspect the changes in the running times. We repeat the benchmarking for 10 times and average the results. We report the relative running time and its standard deviation before and after our modification in Figure 5. From the figure, we observe that our modification seems to slightly slow down the compiler in certain cases. It brings a rise of running time by 5% in the re test, while in most tests the increase is about 1% ∼ 2%. The drop of performance can be attributed to the additional GADT reasoning logic we implemented for type members and singleton types. The added logic will be triggered each time the compiler is typing a pattern match case, which can contribute negatively to the overall performance. However, it should be noted that we have not yet expended any efforts into optimizing our implementation, since we have been focusing on correctness first. So there is still much room for performance improvements to be explored, possibly making the overhead of path-dependent GADT reasoning close to negligible in the future.

6 Discussion

In previous sections, we lay out the framework of GADT reasoning framework in Dotty, and present our extension to the framework to enable GADT reasoning for path-dependent types and singleton types. Apart from the additional functionality it brings to the compiler that will benefit real-world use cases, our work also brings Dotty closer to the dependent object type system, the theoretical foundations of the compiler. In this section, we wrap up our presentation by informally discussing the soundness of our approach to path-dependent GADT constraint inference, and noting the future work in this direction.

6.1 Relation with Theory

Apart from benefiting real-world usages, our implementation also brings the Dotty compiler closer to its formal foundations.

The essence of GADT reasoning in Scala is to derive constraints from cohabitation. On the formal side, DOT [Amin et al. 2016] already takes into account type information following from cohabitation – that is, from values of types of the form $T \ \& \ U$ which are in scope. More specifically, DOT can leverage information present in the bounds of any values bound in the current typing context. This reasoning crucially uses the DOT type member typing rules, which states that path-dependent types are bounded by the bounds of the corresponding type member, and on subtyping transitivity rule. For instance, given a value x of type $x : \{A :> S <: T\}$ in scope, then the relationships $S <: x.A$ and $x.A <: T$ hold, and by transitivity so does $S <: T$.

Ideally, on the practical side, the compiler should also make use of this information found in the bounds of the variables in scope. However, in the general case, because DOT supports complex recursive type specifications (just like Scala), these rules make type checking in DOT undecidable, and an algorithmic version of DOT thus *has to* somehow restrict them [Hu and Lhoták 2019; Nieto 2017]. The current approach taken by the Scala 3 compiler until now was to simply completely disregard any information that could be obtained from the bounds of the variables in scope (making the subtyping relationship of Scala 3 not transitive). In order to leverage such bounds information, users would have to use explicit type annotations, essentially proving transitivity to the compiler every time — for instance, given x: $\{A :> S <: T\}$ in scope, if one wanted to upcast a value e of type F[S] to type F[T], assuming some covariant type constructor F, one would have to write 'e: F[x.A]'.

What we propose and implement in this paper is a compromise, a sweet spot between full leverage of bounds information (DOT, undecidable) and no leverage of bounds information at all (current Scala 3, too restrictive). Indeed, most cases that are actually useful in the context of GADT-style programming are simple and do not involve recursive types, so we can leverage their bounds without getting into decidability problems. We also choose to only consider bounds information arising specifically from values that are being pattern-matched because this is typically where new typing information comes into play in user programs. Trying to leverage all information present in all variables in scope at

```
(e1: { type T :> A1 <: B1 }) match
  case e2: { type T :> A2 <: B2 } => // ...
```

Listing 10. A code example for discussing soundness of path-dependent GADT constraint inference.

any time would probably impose too much of a performance penalty on the compiler, for relatively little gain. We reserve investigating this possible extension as future work.

In a nutshell, our work helps the compiler infer and make use of additional subtyping relationships that demonstrably hold in the formal system, thus bringing the implementation closer to the theory.

6.2 Informal Discussion of Soundness

We informally discuss the soundness of the approach adopted in our path-dependent GADT implementation. The derived constraints are sound if they *necessarily* follow from the cohabitation condition. Listing 10 presents an example for reasoning pattern matching involving type members. Note that the example can be generalised to cases with multiple type members easily, while we stick to the single type member case for the ease of presentation. In our algorithm, when reasoning about the case, we will extract GADT bounds from the subtyping relations $e_1.T <: e_2.T$ and $e_2.T <: e_1.T$. More specifically, this will end up performing the following operations:

1. Add orderings between type members: $e_1.T <: e_2.T$ and $e_2.T <: e_1.T$.
2. Further constrain type members with propagted bounds: $e_1.T <: B_2$, $e_1.T :> A_2$, $e_2.T <: B_1$ and $e_2.T :> A_1$.
3. Derive GADT constraints based on the subtyping relations: $A_1 <: B_2$ and $A_2 <: B_1$.

Then, we informally verify that all operations will produce constraints necessary from the cohabitation $e_2 : \{T :> A_2 <: B_2\} \& e_1.type$. Firstly, e_2 being of type $e_1.type$ implies that e_1 and e_2 are two symbols denoting the same runtime instance. Therefore, $e_1.T$ and $e_2.T$ will refer to the same path-dependent type, and the constraints from the first and the second operation are all necessary from the cohabitation.

Next, we show the necessity of constraints derived by the third operation. In the body of the pattern matching, there exists a variable $e = e_1 = e_2$ of type $\{T :> A_1 <: B_1\} \& \{T :> A_2 <: B_2\} = \{T :> A_1 \mid A_2 <: B_1 \& B_2\}$. As stated before, the existence of an instance of a type can serve as subtyping proof between the lower and upper bounds of its type members. Therefore, from the existence of variable e , we know that $A_1 \mid A_2 <: B_1 \& B_2$, which implies $A_2 <: B_1$ and $A_1 <: B_2$. We leave the formal soundness proof as future work.

[7] https://github.com/lampepfl/dotty/issues/13074

```
class Module[*[_, _]]:

  sealed trait Box[A]

  case class Box2x2[A, B, C, D](value: (A * B) * (C * D))
     extends Box[(A * B) * (C * D)]

  def unbox2[X, Y](box: Box[X * Y]): (X * Y) =
    box match
      case Box2x2(v) => v
```

Listing 11. A code snippet showing the necessity of storing GADT constraints for concrete types. It is taken from issue #13074[7] in the Dotty repository.

6.3 Future Work

Here, we further note the future work that can be done in this direction.

Unified GADT reasoning. Currently, GADT reasoning for type parameters and type members have their separate data structure and logic. However, type parameter GADT can be modeled with type members [Parreaux et al. 2019; Waśko 2020], suggesting the possibility of unifying the handling logic for them. This will not only simplify the codebase, but also improve the consistency between the compiler and the theory.

Recording GADT constraint for concrete types. Sometimes we fail to derive GADT constraints due to lack of constructor injectivity. Listing 11 gives such an example, where we can not assume the injectivity of the abstract constructor *, so we will not break the constraint $A * B * C * D = X * Y$ down to further derive bounds for the type parameters, and thus reject the code. However, the code itself is sound since it only relies on the constraint $A * B * C * D = X * Y$. The key idea of JIT registration scheme in our implementation, where we internalize and record bounds for path-dependent types *when we found it possible*, suggests the possibility of recording and utilizing such constraints between concrete types to facilitate type checking.

Constraint inference for a wider range of cohabitation. As stated before, DOT can leverage constraints implied from the bounds of *all* variables in the typing environment, while the GADT reasoning in Dotty currently only exploit bounds of the scrutinee in pattern matching. This suggests the possibility of deriving GADT bounds whenever a new instance of cohabitated types comes into scope. Although the user can always pattern match on the instance to trigger GADT reasoning manually, this allows the *automatic* discovery of rich bounds information, that can benefit type reasoning, reduce boilerplates and improve user experience.

7 Related Work

In this section, we briefly introduce the related work on background topics. We will start with the definition and informal

discussion of general GADTs, followed by the implementation of GADTs in Scala, and finally introduce the work on Scala's type members.

Generalized Algebraic Data Types. Generalized Algebraic Data Types [Xi et al. 2003] is an extension to parameterized ADTs that enables the data constructors to specialize type parameters of the classes, and makes it possible to discover constraints over type parameters when performing pattern matching. These constraints allow the compilation of code snippets that will not compile before without GADT reasoning. Additionally, GADTs enable *existential quantification.* To see an example of this, if we have a TupleTag of type Tag t defined in Listing 12, we know that there *exists* two types a and b, such that $t = (a, b)$. As a widely-adopted language feature, GADT finds its support in most functional languages. OCaml introduces the concept of upward and downward-closure along with a calculus for variance signs to support the checking of GADT definitions and the inference of GADT constraints in pattern matching. However, the GADT constraints inferenced in OCaml are restricted to equality constraints, which limits the full power of GADT reasoning with subtyping [Scherer and Rémy 2013]. Haskell, on the other side, supports first-class phantom type for GADT reasoning. In Haskell, algebraic data types can be created with phantom types and equality constraints, and the constraints can be extracted when doing pattern matching to facilitate typechecking [Cheney and Hinze 2003; Jones et al. 2006]. However, different from Scala, GADT support in OCaml and Haskell are all limited to closed GADT, and neither of them support GADT reasoning for types with a open and complex hierarchy.

```
trait Tag[T]
case class TupleTag[A, B]() extends Tag[(A, B)]
```

Listing 12. Definition of Tag using GADT. Existential quantification is involved in the TupleTag constructor.

Implementing GADTs in Scala. The task of implementing GADT reasoning for Scala is complicated by Scala's support for advanced type system features, including subtyping, variance [Castagna 1995], and refinements [Cook et al. 1989], which fundamentally differ from existing languages with support for GADTs. Boruch-Gruszecki [2017] proposed a GADT-aware algorithm for verifying the exhaustiveness of GADT pattern matching and later added support for GADT reasoning to the Scala 3 compiler, which enabled many usual GADT patterns to type check in Scala 3. Parreaux et al. [2019] showed that looking at the problem through the lens of Scala's formal foundation, the Dependent Object Type (DOT) calculus [Amin et al. 2016] (and its pDOT extension [Rapoport and Lhoták 2019]), provided a way of deriving sound GADT reasoning principles for Scala, which Waśko [2020] later elaborated. The core of this idea is to use type members to represent the existential types that arise from GADTs and to use bounds on these type members and path-dependent types to represent and leverage the additional information attached to them. Listing 13 illustrates a pitfall brought by the variance of class type parameters [Giarrusso 2013; Parreaux et al. 2019]. Counterintuitively, the inequality $A <: B$ does not hold in the body because the class type parameters are defined to have variance and can be further refined. For instance, it is possible to set A = Any and B = Nothing and create new Identity[X] & Func[Nothing, Any] to satisfy the match case. If we wrongly assume that $A <: B$ in this example, we will have Any <: Nothing and end up with a soundness hole.

```
trait Func[-A, +B]
class Identity[X] extends Func[X, X]

def foo[A, B](func: Func[A, B]) = func match
  case _: Identity[c] =>
    val b: B = ??? : A // error! We cannot assume A <: B here
```

Listing 13. Code illustrating a pitfall in Scala GADT reasoning due to type parameter variance.

Type members. As an essential language feature of Scala type members have been studied and formalized by previous work [Amin et al. 2014; Odersky et al. 2016]. Type members are type fields in Scala objects. They can be abstract in the base classes and be instantiated or refined in derived subclasses. Importantly, all type members of a class must be instantiated when creating an instance of that class [Amin et al. 2016]. Therefore, the type member of an class instance can be viewed as subtyping proof between its lower and upper bounds. Previous work studies the properties of path-dependent types [Amin et al. 2014] and Rapoport and Lhoták formalizes them in a formal system, pDOT. We may infer GADT constraints for path-dependent types just like what we do for type parameters. As a missing part of Dotty's GADT reasoning, path-dependent GADT constraint inference can produce richer constraints and bring the compiler closer to its formalism.

8 Conclusion

As a missing piece of the puzzle, path-dependent and singleton GADT reasoning is not implemented in the current Dotty compiler but will benefit many real world use cases and shorten the gap between the compiler implementation and the formal foundations. To this end, we propose to implement GADT reasoning for path-dependent and singleton types in Dotty. This paper hopes to present the GADT implementation in Dotty, and get those who are interested in the technical details of Dotty internals familiar with related data structures and program logic, to facilitate future development.

References

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*.

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-Dependent Types. *SIGPLAN Not.* 49, 10 (Oct. 2014), 233–249. https://doi.org/10.1145/2714064.2660216

Aleksander Boruch-Gruszecki. 2017. *Verifying the totality of pattern matching in Scala.* Master's thesis. Wrocław University of Science and Technology.

Giuseppe Castagna. 1995. Covariance and Contravariance: Conflict without a Cause. *ACM Trans. Program. Lang. Syst.* 17, 3 (May 1995), 431–447. https://doi.org/10.1145/203095.203096

J. Cheney and R. Hinze. 2003. First-Class Phantom Types.

William R. Cook, Walter Hill, and Peter S. Canning. 1989. Inheritance is Not Subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 125–135. https://doi.org/10.1145/96709.96721

Paolo G. Giarrusso. 2013. Open GADTs and Declaration-Site Variance: A Problem Statement. In *Proceedings of the 4th Workshop on Scala* (Montpellier, France) *(SCALA '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/2489837.2489842

Jason Z. S. Hu and Ondřej Lhoták. 2019. Undecidability of D<: And Its Decidable Fragments. *Proc. ACM Program. Lang.* 4, POPL, Article 9 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371077

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia L. Lawall (Eds.). ACM, 50–61. https://doi.org/10.1145/1159803.1159811

Abel Nieto. 2017. Towards Algorithmic Typing for DOT (Short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 2–7. https://doi.org/10.1145/3136000.3136003

Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala: Updated for Scala 2.12* (3rd ed.). Artima Incorporation, Sunnyvale, CA, USA.

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019. Towards Improved GADT Reasoning in Scala. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala* (London, United Kingdom) *(Scala '19)*. Association for Computing Machinery, New York, NY, USA, 12–16. https://doi.org/10.1145/3337932.3338813

Marianna Rapoport and Ondřej Lhoták. 2019. A Path to DOT: Formalizing Fully Path-Dependent Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 145 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360571

Gabriel Scherer and Didier Rémy. 2013. GADTs Meet Subtyping. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 554–573. https://doi.org/10.1007/978-3-642-37036-6_30

Radosław Waśko. 2020. Formal foundations for GADTs in Scala.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150