Theoretical Computer Science 741 (2018) 3–18

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Transposing G to C♯: Expressivity of generalized algebraic data types in an object-oriented language

Andrew J. Kennedy [1], Claudio V. Russo *

Microsoft Research Ltd, Cambridge, UK

ARTICLE INFO

Article history:
Received 17 July 2017
Received in revised form 15 January 2018
Accepted 26 February 2018
Available online 2 April 2018

Keywords:
Generalized algebraic data types
Generics
Polymorphism
System F
C#

ABSTRACT

Generalized algebraic datatypes (GADTs) are a hot topic in the functional programming community. Previously we showed that object-oriented languages such as C♯ and Java can express GADT declarations using Generics, but only some GADT programs. The addition of equational constraints on type parameters recovers expressivity. We now study this expressivity gap in more depth by extending an earlier translation from System F to C♯ to handle GADTs. Our efforts reveal some surprising limitations of Generics and provide further justification for equational constraints.

Dear Don, if papers were printed on vinyl, this would be the B-side of [4]. Underappreciated by critics, it remains a personal favorite. So what's the connection? You introduced me to the wonderful world of Type Theory and Standard ML which led to my first proper job working with Andrew Kennedy on an SML compiler. You encouraged me to look at System F which we used to crack SML Modules and which reappears in this paper to shed some light on object-oriented Generics. It may not be what you wished for but Happy Birthday! — Claudio

1. Introduction

Functional programming languages such as Haskell and ML have long supported user-defined *datatypes*. A datatype declaration simultaneously defines a named type, parameterized by other types, and the means of constructing values of that type. For example, here is a Haskell datatype of binary trees parameterized on the type d of data and type k of keys stored in the nodes:

data $Tree\ k\ d\ =\ Leaf\ \mid\ Node\ k\ d\ (Tree\ k\ d)\ (Tree\ k\ d)$

This defines two polymorphic *value constructors* $Leaf$ and $Node$ with types:

$Leaf\ ::\ Tree\ k\ d, \quad Node\ ::\ k\ \to\ d\ \to\ Tree\ k\ d\ \to\ Tree\ k\ d\ \to\ Tree\ k\ d.$

Notice how both term constructors have the fully generic result type $Treekd$; there is no specialization of the type parameters to $Tree$. Conversely, any value of type $Tree\tau\sigma$, for some concrete τ and σ, can either be a leaf or a node —

* Corresponding author.
E-mail addresses: akenn@fb.com (A.J. Kennedy), crusso@microsoft.com (C.V. Russo).
[1] Present address: Facebook UK Ltd., London, UK.

https://doi.org/10.1016/j.tcs.2018.02.030
0304-3975/

the static type does not reveal which. Observe that all recursive uses of the datatype within its definition are *Treekd*: this makes *Tree* a *regular* datatype.

The restrictions on *parameterized algebraic datatypes* (PADTs) can be relaxed in three ways, yielding *generalized algebraic datatypes* (GADTs):

1. The restriction that constructors all return 'generic' instances of the datatype can be removed. This feature defines GADTs.
2. The regularity restriction can be removed, permitting datatypes to be used at different instantiations within their own definition. Writing useful functions over such types requires *polymorphic recursion*: the ability to use a polymorphic function at different types within its own definition. C♯, Java and Haskell allow this, ML does not.
3. A constructor can be allowed to mention additional type variables that may appear in its argument types but do not appear in its result type. These type arguments are hidden by the type of the constructed term and thus existentially quantified.

Most useful examples of GADTs make use of all three abilities. Consider the following type *Expt* representing abstract syntax for expressions of type *t*, written in Haskell with GADTs [10,11]:

```
data Exp t where
  Lit    :: Int → Exp Int
  Plus   :: Exp Int → Exp Int → Exp Int
  Equals :: Exp Int → Exp Int → Exp Bool
  Cond   :: Exp Bool → Exp a → Exp a → Exp a
  Tuple  :: Exp a → Exp b → Exp (a, b)
  Fst    :: Exp (a, b) → Exp a
  ...
```

All constructors except for *Cond* make use of feature (1), as their result types refine the type arguments of *Exp*: for example, *Lit* has result type *ExpInt*. All constructors except for *Lit* make use of feature (2), using the datatype at different instantiations in arguments to the constructor. Finally, *Fst* uses a hidden type *b*, thus making use of feature (3).

Why is this interesting? Consider this evaluator for expressions, defined by case analysis on values of type *Expt* (note that '—' begins a Haskell comment):

```
eval :: Exp t → t
eval e = case e of  Lit i → i                  -- t = Int
                    Plus e1 e2 → eval e1 + eval e2     -- t = Int
                    Equals e1 e2 → eval e1 == eval e2  -- t = Bool
                    Cond e1 e2 e3 →        -- t = a
                      if eval e1 then eval e2 else eval e3
                    Tuple e1 e2 → (eval e1, eval e2)   -- t = (a, b)
                    Fst e → fst (eval e)   -- t = a
                    ...
```

The fascinating thing about *eval* is that the compiler doesn't reject it. Observe closely: some branches of the *case* expression return computations of different types. The *Lit* branch returns an integer, the *Equals* branch returns a boolean, the *Tuple* branch returns a pair. In the ML type system, all the continuations of a case expression are required to have the same type and one would expect *eval* to be rejected as type-incorrect. In GADT Haskell, this requirement is subtly relaxed: each branch must, instead, merely have an *appropriate* type, given the type of its pattern and the type of the scrutinee.

Although probably unintentional, both C♯ and Java Generics already support GADTs. Consider the C♯ code in Fig. 1. This is a straightforward encoding of the GADT Haskell datatype *Expt*. Abstract syntax trees are represented using an abstract class of expressions, with a concrete subclass for each node type. The interpreter is implemented by an abstract `Eval` method in the expression class, overridden for each node type. Indeed, this is a subtle variant of the *Interpreter* design pattern. Observe how the type parameter of `Exp` is refined in subclasses; moreover, this refinement is reflected in the signature and code of the overridden `Eval` methods. For example, `Plus.Eval` has result type `int` and requires no runtime casts in its calls to `e1.Eval()` and `e2.Eval()`. Not only is this a clever use of static typing, it is also more efficient than a dynamically-typed version, particularly in an implementation that performs code specialization to avoid boxing [5].

Just like our Haskell datatype, these C♯ classes make use of all three features that characterize GADTs. Feature (1) is expressed by defining a subclass of a generic type that does not just propagate its type parameters through to the superclass. (`Plus` is a non-generic class that extends the particular instantiation `Exp<int>`.) Feature (2) corresponds to the existence of fields in the subclass whose types are unrelated instantiations of the generic type of the superclass. (`Tuple<A,B>` has a field of type `Exp<A>` but superclass `Exp<Pair<A,B>>`.) Feature (3) corresponds to the declaration of type parameters on the subclass that are not referenced in the superclass. (`Fst<A,B>`'s superclass `Exp<A>` hides `B`.)