InvarGenT – Manual

by Łukasz Stafiniak

Institute of Computer Science
University of Wrocław

Abstract

InvarGenT is a proof-of-concept system for invariant generation by full type inference with Guarded Algebraic Data Types and existential types encoded as automatically generated GADTs. This user manual discusses motivating examples, briefly presents the syntax of the InvarGenT language, and describes the parameters of the inference process that can be passed to the InvarGenT executable.

1 Introduction

Type systems are an established natural deduction-style means to reason about programs. Dependent types can represent arbitrarily complex properties as they use the same language for both types and programs, the type of value returned by a function can itself be a function of the argument. Generalized Algebraic Data Types bring some of that expressivity to type systems that deal with data-types. Type systems with GADTs introduce the ability to reason about return type by case analysis of the input value, while keeping the benefits of a simple semantics of types, for example deciding equality can be very simple. Existential types hide some information conveyed in a type, usually when that information cannot be reconstructed in the type system. A part of the type will often fail to be expressible in the simple language of types, when the dependence on the input to the program is complex. GADTs express existential types by using local type variables for the hidden parts of the type encapsulated in a GADT.

Our type system for GADTs differs from more pragmatic approaches in mainstream functional languages in that we do not require any type annotations on expressions, even on recursive functions. Our implementation: InvarGenT, see [?], also includes linear equations and inequalities over rational numbers in the language of types, with the possibility to introduce more domains in the future.

2 Tutorial

The concrete syntax of InvarGenT is similar to that of OCaml. However, it does not currently cover records, the module system, objects, and polymorphic variant types. It supports higher-order functions, algebraic data-types including built-in tuple types, and linear pattern matching. It supports conjunctive patterns using the `as` keyword, but it does not support disjunctive patterns. It does not currently support guarded patterns, i.e. no support for the `when` keyword of OCaml.

The sort of a type variable is identified by the first letter of the variable. `a`,`b`,`c`,`r`,`s`,`t`,`a1`,... are in the sort of terms called `type`, i.e. "types proper". `i`,`j`,`k`,`l`,`m`,`n`,`i1`,... are in the sort of linear arithmetics over rational numbers called `num`. Remaining letters are reserved for sorts that may be added in the future. Type constructors and value constructors have the same syntax: capitalized name followed by a tuple of arguments. They are introduced by `newtype` and `newcons` respectively. The `newtype` declaration might be misleading in that it only lists the sorts of the arguments of the type, the resulting sort is always `type`. Values assumed into the environment are introduced by `external`. There is a built-in type corresponding to declaration `newtype Num : num` and definitions of numeric constants `newcons 0 : Num 0 newcons 1 : Num 1`... The programmer can use `external` declarations to give the semantics of choice to the `Num` data-type.

In examples here we use Unicode characters. For ASCII equivalents, take a quick look at the tables in the following section. `equal` is a function comparing values provided representation of their types:

```
newtype Ty : type
newtype Int
newtype List : type
newcons Zero : Int
newcons Nil : ∀a. List a
newcons TInt : Ty Int
newcons TPair : ∀a, b. Ty a * Ty b ⟶ Ty (a, b)
newcons TList : ∀a. Ty a ⟶ Ty (List a)
newtype Boolean
newcons True : Boolean
newcons False : Boolean
external eq_int : Int → Int → Bool
external b_and : Bool → Bool → Bool
external b_not : Bool → Bool
external forall2 : ∀a, b. (a → b → Bool) → List a → List b → Bool

let rec equal1 = function
  | TInt, TInt -> fun x y -> eq_int x y
  | TPair (t1, t2), TPair (u1, u2) ->
    (fun (x1, x2) (y1, y2) ->
        b_and (equal (t1, u1) x1 y1)
              (equal (t2, u2) x2 y2))
  | TList t, TList u -> forall2 (equal (t, u))
  | _ -> fun _ _ -> False
```

InvarGenT returns an unexpected type: `equal1`: ∀`a,b.(a,b)`→`b`→`b`→`Bool`, one of four maximally general types of `equal1`. This illustrates that unrestricted type systems with GADTs lack principal typing property.

InvarGenT commits to a type of a toplevel definition before proceeding to the next one, so sometimes we need to provide more information in the program. Besides type annotations, there are two means to enrich the generated constraints: `assert false` syntax for providing negative constraints, and `test` syntax for including constraints of use cases with constraint of a toplevel definition. To ensure only one maximally general type for `equal`, we use both. We add the lines:

```
  | TInt, TList l -> (function Nil -> assert false)
  | TList l, TInt -> (fun _ -> function Nil -> assert false)
test b_not (equal (TInt, TList TInt) Zero Nil)
```

Actually, InvarGenT returns the expected type `equal`: ∀`a,b.(a,b)`→`a`→`b`→`Bool` when either the two `assert false` clauses or the `test` clause is added. When using `test`, the program should declare the type `Boolean` and constant `True`. In a future version, this will be replaced by a built-in type `Bool` with constants `True` and `False`, exported into OCaml as type `bool` with constants `true` and `false`.

Now we demonstrate numerical invariants:

```
newtype Binary : num
newtype Carry : num
newcons Zero : Binary 0
newcons PZero : ∀n[0≤n]. Binary(n) ⟶ Binary(n+n)
newcons POne : ∀n[0≤n]. Binary(n) ⟶ Binary(n+n+1)
newcons CZero : Carry 0
newcons COne : Carry 1

let rec plus =
  function CZero ->
    (function Zero -> (fun b -> b)
```

```
      | PZero a1 as a ->
        (function Zero -> a
          | PZero b1 -> PZero (plus CZero a1 b1)
          | POne b1 -> POne (plus CZero a1 b1))
      | POne a1 as a ->
        (function Zero -> a
          | PZero b1 -> POne (plus CZero a1 b1)
          | POne b1 -> PZero (plus COne a1 b1)))
    | COne ->
    (function Zero ->
        (function Zero -> POne(Zero)
          | PZero b1 -> POne b1
          | POne b1 -> PZero (plus COne Zero b1))
      | PZero a1 as a ->
        (function Zero -> POne a1
          | PZero b1 -> POne (plus CZero a1 b1)
          | POne b1 -> PZero (plus COne a1 b1))
      | POne a1 as a ->
        (function Zero -> PZero (plus COne a1 Zero)
          | PZero b1 -> PZero (plus COne a1 b1)
          | POne b1 -> POne (plus COne a1 b1)))
```

We get `plus`:∀`i,j,k.Carry i`→`Binary j`→`Binary k`→`Binary (i+j+k)`.

We can introduce existential types directly in type declarations. To have an existential type inferred, we have to use `efunction` or `ematch` expressions, which differ from `function` and `match` only in that the (return) type is an existential type. To use a value of an existential type, we have to bind it with a `let..in` expression. Otherwise, the existential type will not be unpacked. An existential type will be automatically unpacked before being "repackaged" as another existential type.

```
newtype Room
newtype Yard
newtype Village
newtype Castle : type
newtype Place : type
newcons Room : Room ⟶ Castle Room
newcons Yard : Yard ⟶ Castle Yard
newcons CastleRoom : Room ⟶ Place Room
newcons CastleYard : Yard ⟶ Place Yard
newcons Village : Village ⟶ Place Village

external wander : ∀a. Place a → ∃b. Place b

let rec find_castle = efunction
  | CastleRoom x -> Room x
  | CastleYard x -> Yard x
  | Village _ as x ->
    let y = wander x in
    find_castle y
```

We get `find_castle`:∀`a. Place a`→ ∃`b. Castle b`.

A more practical existential type example:

```
newtype Bool
newcons True : Bool
newcons False : Bool
newtype List : type * num
```

```
newcons LNil : ∀a. List(a, 0)
newcons LCons : ∀n,a[0≤n]. a * List(a, n) ⟶ List(a, n+1)

let rec filter = fun f ->
  efunction LNil -> LNil
    | LCons (x, xs) ->
      ematch f x with
        | True ->
          let ys = filter f xs in
          LCons (x, ys)
        | False ->
          filter f xs
```

We get `filter`:∀a,i.(a→Bool)→List (a, i)→ ∃j[j≤i].List (a, j). Note that we need to use both `efunction` and `ematch` above, since every use of `function` or `match` will force the types of its branches to be equal. In particular, for lists with length the resulting length would have to be the same in each branch. If the constraint cannot be met, as for `filter` with either `function` or `match`, the code will not type-check.

A more complex example that computes bitwise *or* – `ub` stands for "upper bound":

```
newtype Binary : num
newcons Zero : Binary 0
newcons PZero : ∀n [0≤n]. Binary(n) ⟶ Binary(n+n)
newcons POne : ∀n [0≤n]. Binary(n) ⟶ Binary(n+n+1)

let rec ub = efunction
  | Zero ->
      (efunction Zero -> Zero
        | PZero b1 as b -> b
        | POne b1 as b -> b)
  | PZero a1 as a ->
      (efunction Zero -> a
        | PZero b1 ->
          let r = ub a1 b1 in
          PZero r
        | POne b1 ->
          let r = ub a1 b1 in
          POne r)
  | POne a1 as a ->
      (efunction Zero -> a
        | PZero b1 ->
          let r = ub a1 b1 in
          POne r
        | POne b1 ->
          let r = ub a1 b1 in
          POne r)
```

Type: `ub`:∀k,n.Binary k→Binary n→∃:i[n≤i∧k≤i∧i≤k+n∧0≤n∧0≤k].Binary i.

Why cannot we shorten the above code by converting the initial cases to `Zero -> (efunction b -> b)`? Without pattern matching, we do not make the contribution of `Binary n` available. Knowing `n`=`i` and not knowing 0≤`n`, for the case `k`=0, we get: `ub`:∀k,n.Binary k→Binary n→ ∃:i[n≤i∧i≤k+n∧0≤k].Binary i. `n`≤`i` follows from `n`=`i`, `i`≤`k+n` follows from `n`=`i` and 0≤`k`, but `k`≤`i` cannot be inferred from `k`=0 and `n`=`i` without knowing that 0≤`n`.

In fact, `let...in` expressions are syntactic sugar for pattern matching with a single branch.

Besides displaying types of toplevel definitions, InvarGenT can also export an OCaml source file with all the required GADT definitions and type annotations.

3 Syntax

Below we present, using examples, the syntax of InvarGenT: the mathematical notation, the concrete syntax in ASCII and the concrete syntax using Unicode.

type variable: types	$\alpha, \beta, \gamma, \tau$	`a,b,c,r,s,t,a1,...`	
type variable: nums	k, m, n	`i,j,k,l,m,n,i1,...`	
type constructor	List	`List`	
number (type)	7	`7`	
numeral (expr.)	7	`7`	
numerical sum (type)	$m+n$	`m+n`	
existential type	$\exists k, n[k \leqslant n].\tau$	`ex k n [k<=n].t`	`∃k,n[k≤n].t`
type sort	s_{ty}	`type`	
number sort	s_R	`num`	
function type	$\tau_1 \to \tau_2$	`t1 -> t2`	`t1 → t2`
equation	$a \doteq b$	`a = b`	
inequation	$k \leqslant n$	`k <= n`	`k ≤ n`
conjunction	$\varphi_1 \wedge \varphi_2$	`a=b && b=a`	`a=b ∧ b=a`

Toplevel expressions (corresponding to structure items in OCaml) introduce types, type and value constructors, global variables with given type (external names) or inferred type (definitions).

type constructor	`newtype List : type * num`
value constructor	`newcons Cons : all n a. a * List(a,n) --> List(a,n+1)`
	`newcons Cons : ∀n,a. a * List(a,n) ⟶ List(a,n+1)`
declaration	`external filter : ∀n,a. List(a,n)→∃k[k<=n].List(a,k)`
rec. definition	`let rec f =...`
non-rec. definition	`let a, b =...`

Like in OCaml, types of arguments in declarations of constructors are separated by asterisks. However, the type constructor for tuples is represented by commas, like in Haskell but unlike in OCaml.

For simplicity of theory and implementation, mutual non-nested recursion and or-patterns are not provided. For mutual recursion, nest one recursive definition inside another.

4 Solver Parameters and CLI