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.

The InvarGenT 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. The implementation 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 currently does not support disjunctive patterns. It currently has limited support for guarded patterns: after `when`, only inequality `<=` between values of the `Num` type are allowed.

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. Value constructors (like in OCaml) and type constructors (unlike in OCaml) have the same syntax: capitalized name followed by a tuple of arguments. They are introduced by `datatype` and `datacons` respectively. The `datatype` 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 `datatype 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.

When solving negative constraints, arising from `assert false` clauses, we assume that the intended domain of the sort `num` is integers. This is a workaround to the lack of strict inequality in the sort `num`. We do not make the whole sort `num` an integer domain because it would complicate the algorithms.

In examples here we use Unicode characters. For ASCII equivalents, take a quick look at the tables in the following section.

We start simple, with a function that can compute a value from a representation of an expression – a ready to use value whether it be `Int` or `Bool`. Prior to the introduction of GADT types, we could only implement a function `eval : ∀a. Term a → Value` where, using OCaml syntax, `type value = Int of int | Bool of bool`.

```
datatype Term : type
external let plus : Int → Int → Int = "(+)"
external let is_zero : Int → Bool = "(=) 0"
datacons Lit : Int ⟶ Term Int
datacons Plus : Term Int * Term Int ⟶ Term Int
datacons IsZero : Term Int ⟶ Term Bool
datacons If : ∀a. Term Bool * Term a * Term a ⟶ Term a

let rec eval = function
  | Lit i -> i
  | IsZero x -> is_zero (eval x)
  | Plus (x, y) -> plus (eval x) (eval y)
  | If (b, t, e) -> (match eval b with True -> eval t | False -> eval e)
```

Let us look at the corresponding generated, also called *exported*, OCaml source code:

```
type _ term =
  | Lit : int -> int term
  | Plus : int term * int term -> int term
  | IsZero : int term -> bool term
  | If : (*∀'a.*)bool term * 'a term * 'a term -> 'a term

let plus : (int -> int -> int) = (+)
let is_zero : (int -> bool) = (=) 0
let rec eval : type a . (a term -> a) =
  (function Lit i -> i | IsZero x -> is_zero (eval x)
    | Plus (x, y) -> plus (eval x) (eval y)
    | If (b, t, e) -> (if eval b then eval t else eval e))
```

The `Int`, `Num` and `Bool` types are built-in. `Int` and `Bool` follow the general scheme of exporting a datatype constructor with the same name, only lower-case. However, numerals `0`, `1`, ... are always type-checked as `Num 0`, `Num 1`... `Num` can also be exported as a type other than `int`, and then numerals are exported via an injection function (ending with) `of_int`.

The syntax `external let` allows us to name an OCaml library function or give an OCaml definition which we opt-out from translating to InvarGenT. Such a definition will be verified against the rest of the program when InvarGenT calls `ocamlc -c` (or Haskell in the future) to verify the exported code. Another variant of `external` (omitting the `let` keyword) exports a value using `external` in OCaml code, which is OCaml source declaration of the foreign function interface of OCaml. When we are not interested in linking and running the exported code, we can follow the convention of reusing the name in the FFI definition: `external f : ... = "f"`.

The type inferred is `eval : ∀a. Term a→a`. GADTs make it possible to reveal that `IsZero x` is a `Term Bool` and therefore the result of `eval` should in its case be `Bool`, `Plus (x, y)` is a `Term Num` and the result of `eval` should in its case be `Num`, etc. InvarGenT does not provide an `if`...`then`...`else`... syntax to stress that the branching is relevant to generating postconditions, but it does export `match`/`ematch` ... `with True ->` ... `| False ->` ... using `if` expressions.

`equal` is a function comparing values provided representation of their types:

```
datatype Ty : type
datatype Int
datatype List : type
datacons Zero : Int
datacons Nil : ∀a. List a
datacons TInt : Ty Int
datacons TPair : ∀a, b. Ty a * Ty b ⟶ Ty (a, b)
datacons TList : ∀a. Ty a ⟶ Ty (List a)
datatype Boolean
datacons True : Boolean
datacons 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 equal = 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: `equal`:∀a,b.`(Ty a, Ty b)`→a→a→`Bool`, one of four maximally general types of `equal` as defined above. The other maximally general "wrong" types are ∀a,b.`(Ty a, Ty b)`→b→b→`Bool` and ∀a,b.`(Ty a, Ty b)`→b→a→`Bool`. 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 three means to enrich the generated constraints: `assert false` syntax for providing negative constraints, `assert type e1 = e2; ...` and `assert num e1 <= e2; ...` for positive 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 `assert false` and `test`. 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.`(Ty a, Ty b)`→a→b→`Bool` when either the two `assert false` clauses or the `test` clause is added.

Now we demonstrate numerical invariants:

```
datatype Binary : num
datatype Carry : num
datacons Zero : Binary 0
datacons PZero : ∀n[0≤n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n[0≤n]. Binary n ⟶ Binary(2 n + 1)
datacons CZero : Carry 0
datacons 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,k,n.Carry i`→`Binary k`→`Binary n`→`Binary (n + k + i)`.

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.

```
datatype Room
datatype Yard
datatype Village
datatype Castle : type
datatype Place : type
datacons Room : Room ⟶ Castle Room
datacons Yard : Yard ⟶ Castle Yard
datacons CastleRoom : Room ⟶ Place Room
datacons CastleYard : Yard ⟶ Place Yard
datacons 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:

```
datatype Bool
datacons True : Bool
datacons False : Bool
```

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
datacons 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`: ∀n, a.(a→Bool)→List (a, n)→ ∃k[0≤n ∧ 0≤k ∧ k≤n].List (a, k). 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":

```
datatype Binary : num
datacons Zero : Binary 0
datacons PZero : ∀n [0≤n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n [0≤n]. Binary n ⟶ Binary(2 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)
```

`ub`:∀k,n.Binary k→Binary n→∃:i[0≤n ∧ 0≤k ∧ n≤i ∧ k≤i ∧ i≤n+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[0≤k∧n≤i∧i≤n+k].Binary i. `n`≤`i` follows from `n`=`i`, `i`≤`n`+`k` follows from `n`=`i` and 0≤`k`, but `k`≤`i` cannot be inferred from `k`=0 and `n`=`i` without knowing that 0≤`n`.

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 var. with coef.	$\frac{1}{3}n$	`1/3 n`	
type constructor	List	`List`	
number (type)	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`

For the syntax of expressions, we discourage non-ASCII symbols. Below e, e_i stand for any expression, p, p_i stand for any pattern, x stands for any lower-case identifier and K for an upper-case identifier.

named value	x	`x` –lower-case identifier
numeral (expr.)	7	`7`
constructor	K	`K` –upper-case identifier
application	$e_1\, e_2$	`e1 e2`
non-br. function	$\lambda(p_1.\lambda(p_2.e))$	`fun (p1,p2) p3 -> e`
branching function	$\lambda(p_1.e_1 ... p_n.e_n)$	`function p1->e1 | ... | pn->en`
pattern match	$\lambda(p_1.e_1 ... p_n.e_n)\, e$	`match e with p1->e1 | ... | pn->en`
postcond. function	$\lambda[K](p_1.e_1 ... p_n.e_n)$	`efunction p1->e1 | ...`
postcond. match	$\lambda[K](p_1.e_1 ... p_n.e_n)\, e$	`ematch e with p1->e1 | ...`
rec. definition	$\mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2$	`let rec x = e1 in e2`
definition	$\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2$	`let p1,p2 = e1 in e2`
asserting dead br.	$\boldsymbol{F}$	`assert false`
assert equal types	$\mathbf{assert\ type}\, \tau_{e_1} \doteq \tau_{e_2}; e_3$	`assert type e1 = e2; e3`
assert inequality	$\mathbf{assert\ num}\, e_1 \leqslant e_2; e_3$	`assert num e1 <= e2; e3`

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	`datatype List : type * num`
value constructor	`datacons Cons : all n a. a * List(a,n) --> List(a,n+1)`
	`datacons Cons : ∀n,a. a * List(a,n) ⟶ List(a,n+1)`
declaration	`external foo : ∀n,a. List(a,n)→∃k[k<=n].List(a,k)="c_foo"`
	`external filter : ∀n,a. List(a,n)→∃k[k<=n].List(a,k)`
let-declaration	`external let mult : ∀n,m. Num n→Num m→∃k.Num k = "( * )"`
rec. definition	`let rec f =...`
non-rec. definition	`let a, b =...`
definition with test	`let rec f =... test e1; ...; en`
	`let p1,p2 =... test e1; ...; en`

Tests list expressions of type `Bool` that at runtime have to evaluate to `True`. Type inference is affected by the constraints generated to typecheck the expressions.

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.

At any place between lexemes, regular comments encapsulated in `(*...*)` can occur. They are ignored during lexing. In front of all toplevel definitions and declarations, e.g. before a `datatype`, `datacons`, `external`, `let rec` or `let`, and in front of `let rec .. in` and `let .. in` nodes in expressions, documentation comments `(**...*)` can be put. Documentation comments at other places are syntax errors. Documentation comments are preserved both in generated interface files and in exported source code files.

4 Solver Parameters and CLI

The default settings of InvarGenT parameters should be sufficient for most cases. For example, after downloading InvarGenT source code and changing current directory to `invargent`, we can enter, assuming a Unix-like shell:

```
$ make main
$ ./invargent examples/binary_upper_bound.gadt
```

To get the inferred types printed on standard output, use the `-inform` option:

```
$ ./invargent -inform examples/binomial_heap_nonrec.gadt
```

In some situations, hopefully unlikely for simple programs, the default parameters of the solver algorithms do not suffice. Consider this example, where we use `-full_annot` to generate type annotations on `function` and `let..in` nodes in the `.ml` file, in addition to annotations on `let rec` nodes:

```
$ ./invargent -inform -full_annot examples/equal_assert.gadt
File "examples/equal_assert.gadt", line 20, characters 5-103:
No answer in type: term abduction failed

Perhaps increase the -term_abduction_timeout parameter.
Perhaps increase the -term_abduction_fail parameter.
```

The `Perhaps increase` suggestions are generated only when the corresponding limit has actually been exceeded. Remember however that the limits will often be exceeded for erroneus programs which should not type-check. Here the default number of steps till term abduction timeout, which is just `700` to speed up failing for actually erroneous programs, is too low. The complete output with timeout increased:

```
$ ./invargent -inform -full_annot -term_abduction_timeout 4000 \
examples/equal_assert.gadt
val equal : ∀a, b. (Ty a, Ty b) → a → b → Bool
InvarGenT: Generated file examples/equal_assert.gadti
InvarGenT: Generated file examples/equal_assert.ml
InvarGenT: Command "ocamlc -c examples/equal_assert.ml" exited with code 0
```

To understand the intent of the solver parameters, we need a rough "birds-eye view" understanding of how InvarGenT works. The invariants and postconditions that we solve for are logical formulas and can be ordered by strength. Least Upper Bounds (LUBs) and Greatest Lower Bounds (GLBs) computations are traditional tools used for solving recursive equations over an ordered structure. In case of implicational constraints that are generated for type inference with GADTs, constraint abduction is a form of LUB computation. *Disjunction elimination* is our term for computing the GLB wrt. strength for formulas that are conjunctions of atoms. We want the invariants of recursive definitions – i.e. the types of recursive functions and formulas constraining their type variables – to be as weak as possible, to make the use of the corresponding definitions as easy as possible. The weaker the invariant, the more general the type of definition. Therefore the use of LUB, constraint abduction. For postconditions – i.e. the existential types of results computed by `efunction` expressions and formulas constraining their type variables – we want the strongest possible solutions, because stronger postcondition provides more information at use sites of a definition. Therefore we use GLB, disjunction elimination, but only if existential types have been introduced by `efunction` or `ematch`.

Below we discuss all of the InvarGenT options.

`-inform.` Print type schemes of toplevel definitions as they are inferred.

`-time.` Print the time it took to infer type schemes of toplevel definitions.

`-no_sig.` Do not generate the `.gadti` file.

`-no_ml.` Do not generate the `.ml` file.

`-no_verif.` Do not call `ocamlc -c` on the generated `.ml` file.

`-num_is.` The exported type for which `Num` is an alias (default `int`). If `-num_is bar` for `bar` different than `int`, numerals are exported as integers passed to a `bar_of_int` function. The variant `-num_is_mod` exports numerals by passing to a `Bar.of_int` function.

`-full_annot.` Annotate the `function` and `let..in` nodes in generated OCaml code. This increases the burden on inference a bit because the variables associated with the nodes cannot be eliminated from the constraint during initial simplification.

`-keep_assert_false.` Keep `assert false` clauses in exported code. When faced with multiple maximally general types of a function, we sometimes want to prevent some interpretations by asserting that a combination of arguments is not possible. These arguments will not be compatible with the type inferred, causing exported code to fail to typecheck. Sometimes we indicate unreachable cases just for documentation. If the type is tight this will cause exported code to fail to typecheck too. This option keeps pattern matching branches with `assert false` in their bodies in exported code nevertheless.

`-no_dead_code.` Reject all programs with dead code (may misclassify programs using *min* or *max* atoms). Unreachable pattern matching branches lead to unsatisfiable premises of the type inference constraint, which we detect. However, sometimes multiple implications in the simplified form of the constraint can correspond to the same path through the program, in particular when solving constraints with *min* and *max* clauses. Dead code due to datatype mismatch, i.e. patterns unreachable without resort to numerical constraints, is detected even without using this option.

`-term_abduction_timeout.` Limit on term simple abduction steps (default 700). Simple abduction works with a single implication branch, which roughly corresponds to a single branch – an execution path – of the program.

`-term_abduction_fail.` Limit on backtracking steps in term joint abduction (default 4). Joint abduction combines results for all branches of the constraints.

`-no_alien_prem.` Do not include alien (e.g. numerical) premise information in term abduction.

`-early_num_abduction.` Include recursive branches in numerical abduction from the start. By default, in the second iteration of solving constraints, which is the first iteration that numerical abduction is performed, we only pass non-recursive branches to numerical abduction. This makes it faster but less likely to find the correct solution.

`-early_postcond_abd.` Include postconditions from recursive calls in abduction from the start. We do not derive requirements put on postconditions by recursive calls on first iteration. The requirements may turn smaller after some derived invariants are included in the premises. This option turns off the special treatment of postconditions on first iteration.

`-num_abduction_rotations.` Numerical abduction: coefficients from $\pm 1/N$ to $\pm N$ (default 3). Numerical abduction answers are built, roughly speaking, by adding premise equations of a branch with conclusion of a branch to get an equation or inequality that does not conflict with other branches, but is equivalent to the conclusion equation/inequality. This parameter decides what range of coefficients is tried. If the highest coefficient in correct answer is greater, abduction might fail.

`-num_prune_at.` Keep less than N elements in abduction sums (default 6). By elements here we mean distinct variables – lack of constant multipliers in concrete syntax of types is just a syntactic shortcoming.

`-num_abduction_timeout.` Limit on numerical simple abduction steps (default 1000).

`-num_abduction_fail.` Limit on backtracking steps in numerical joint abduction (default 10).

`-no_num_abduction.` Turn off numerical abduction; will not ensure correctness. Numerical abduction uses a brute-force algorithm and will fail to work in reasonable time for complex constraints. However, including the effects of `assert false` clauses, and inference of post-conditions, do not rely on numerical abduction. If the numerical invariant of a typeable (i.e. correct) function follows from `assert false` facts alone, a call with `-no_num_abduction` may still find the correct invariant and postcondition.

`-weaker_pruning.` Do not assume integers as the numerical domain when pruning redundant atoms.

`-stronger_pruning.` Prune atoms that force a numerical variable to a single value under certain conditions; exclusive with `-weaker_pruning`.

`-disjelim_rotations.` Disjunction elimination: check coefficients from $1/N$ (default 3). Numerical disjunction elimination is performed by approximately finding the convex hull of the polytopes corresponding to disjuncts. A step in an exact algorithm involves rotating a side along a ridge – an intersection with another side – until the side touches yet another side. We approximate by trying out a couple of rotations: convex combinations of the inequalities defining the sides. This parameter decides how many rotations to try.

`-iterations_timeout.` Limit on main algorithm iterations (default 6). Answers found in an iteration of the main algorithm are propagated to use sites in the next iteration. However, for about four initial iterations, each iteration turns on additional processing which makes better sense with the results from the previous iteration propagated. At least three iterations will always be performed.

`-richer_answers.` Keep some equations in term abduction answers even if redundant. Try keeping an initial guess out of a list of candidate equations before trying to drop the equation from consideration. We use fully maximal abduction for single branches, which cannot find answers not implied by premise and conclusion of a branch. But we seed it with partial answer to branches considered so far. Sometimes an atom is required to solve another branch although it is redundant in given branch. `-richer_answers` does not increase computational cost but sometimes leads to answers that are not most general. This can always be fixed by adding a `test` clause to the definition which uses a type conflicting with the too specific type.

`-more_existential.` More general invariant at expense of more existential postcondition. To avoid too abstract postconditions, disjunction elimination can infer additional constraints over invariant parameters. In rare cases a weaker postcondition but a more general invariant can be beneficial.

`-show_extypes.` Show datatypes encoding existential types, and their identifiers with uses of existential types. The type system in InvarGenT encodes existential types as GADT types, but this representation is hidden from the user. Using `-show_extypes` exposes the representation as follows. The encodings are exported in `.gadti` files as regular datatypes named `exN`, and existential types are printed using syntax `∃N:...` instead of `∃...`, where `N` is the identifier of an existential type.

`-passing_ineq_trs.` Include inequalities in conclusion when solving numerical abduction. This setting leads to more inequalities being tried for addition in numeric abduction answer.

`-not_annotating_fun.` Do not keep information for annotating `function` nodes. This may allow eliminating more variables during initial constraint simplification.

`-annotating_letin.` Keep information for annotating `let..in` nodes. Will be set automatically anyway when `-full_annot` is passed.

`-let_in_fallback.` Annotate `let..in` nodes in fallback mode of `.ml` generation. When verifying the resulting `.ml` file fails, a retry is made with `function` nodes annotated. This option additionally annotates `let..in` nodes with types in the regenerated `.ml` file.

Let us see another example where parameters allowing the solver do more work are needed:

```
$ ./invargent -inform -num_abduction_rotations 4 -num_abduction_timeout 2000 \
examples/flatten_quadrs.gadt
val flatten_quadrs :
  ∀n, a. List ((a, a, a, a), n) → List (a, 4 n)
InvarGenT: Generated file examples/flatten_quadrs.gadti
InvarGenT: Generated file examples/flatten_quadrs.ml
InvarGenT: Command "ocamlc -c examples/flatten_quadrs.ml" exited with code 0
```

Based on user feedback, we will likely increase the default values of parameters in a future version.

5 Limitations of Current InvarGenT Inference

Type inference for the type system underlying InvarGenT is undecidable. In some cases, the failure to infer a type is not at all problematic. Consider this example due to Chuan-kai Lin:

```
datatype EquLR : type * type * type
datacons EquL : ∀a, b. EquLR (a, a, b)
datacons EquR : ∀a, b. EquLR (a, b, b)
datatype Box : type
datacons Cons : ∀a. a ⟶ Box a
external let eq : ∀a. a → a → Bool = "(=)"

let vary = fun e y ->
  match e with
  | EquL, EquL -> eq y "c"
  | EquR, EquR -> Cons (match y with True -> 5 | False -> 7)
```

Although `vary` has multiple types, it is a contrived example unlikely to have an intended type. However, not all cases of failure to infer a type for a correct program are due to contrived examples. The problems are not insurmountable theoretically. The algorithms used in the inference can incorporate heuristics for special cases, and can be modified to do a more exhaustive search.

The following example illustrates a limitation of our numerical abduction algorithm that is not intrinsic to the numerical abduction problem. I.e. it might be fixed by a smarter algorithm.

```
datatype Elem
datatype List : num
datacons LNil : List 0
datacons LCons : <forall>n [0<leq>n]. Elem * List n <longrightarrow> List (n+1)
external length : <forall>n. List n <rightarrow> Num n = "length"

let rec append =
  function
    | LNil ->
      (function l when (length l + 1) <= 0 -> assert false | l -> l)
    | LCons (x, xs) ->
      (function l when (length l + 1) <= 0 -> assert false
      | l -> LCons (x, append xs l))
```

The expected type is `append : ∀a, n, k[0≤k]. List n→List k→List (n+k)`. When our algorithm discovers that the result is $n + k$, rather than n, it is already committed to requiring that the result is no less than 1. The answers on successive iterations of the main algorithm do not converge: if the length of the tail has to be at least one, then the length of the input list has to be at least two, etc.

The following example is a natural variant of a function from the `avl_tree.gadt` example.

```
let rec add = fun x -> efunction
  | Empty -> Node (Empty, x, Empty, 1)
  | Node (l, y, r, h) ->
    ematch compare x y with
    | EQ -> Node (l, x, r, h)
    | LT ->
      let l' = add x l in
      (ematch height l', height r with
       | hl', hr when hl' <= hr+2 -> create l' y r
       | hl', hr when hr+3 <= hl' -> rotr l' y r)
    | GT ->
      let r' = add x r in
      (ematch height r', height l with
       | hr', hl when hr' <= hl+2 -> create l y r'
       | hr', hl when hl+3 <= hr' -> rotl l y r')
```

The difference with the function in the `avl_tree.gadt` file amounts to computing `height r`, resp. `height l` near the places where they are used. The inference fails because of lack of sharing of information about `l` due to facts about `l' = add x l`, resp. about `r` due to facts about `r' = add x r`, with the other branch. More sophisticated algorithms might mitigate that.

We end with an example where there is little hope of improvement. The `rotr` and `rotl` functions in `avl_tree.gadt` use assertions to convey the preconditions. Ideally, we would like to be able to simply write an implementation similar to the following one:

```
let rotr = fun l x r ->
    ematch l with
    | Empty -> assert false
    | Node (ll, lx, lr, _) ->
      (ematch height ll, height lr with
      | m, n when n <= m ->
        let r' = create lr x r in
        create ll lx r'
      | m, n when m+1 <= n ->
        (ematch lr with
        | Empty -> assert false
        | Node (lrl, lrx, lrr, _) ->
          let l' = create ll lx lrl in
          let r' = create lrr x r in
          create l' lrx r'))
```

Unfortunately, it seems it would require too much "guesswork" from the inference algorithms.