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 between types 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. The type with additional support as `Num` is the integers.

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 with a simple example, 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) -> if eval b then eval t else 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` 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 omit the part starting with the = sign. The exported code will reuse the name in the FFI definition: `external f : ... = "f"`.

The type inferred for the above example 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. The `if/eif...then...else...` syntax is a syntactic sugar for `match/ematch ... with True -> ... | False -> ...`, and any such expressions are exported 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 can either add the `assert false` clauses:

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

The first assertion excludes independence of the first encoded type and the second argument. The second assertion excludes independence of the second encoded type and the third argument. Or we can add the `test` clause:

```
test b_not (equal (TInt, TList TInt) Zero Nil)
```

The test ensures that arguments of distinct types can be given. InvarGenT returns the expected type `equal:∀a,b.(Ty a, Ty b)→a→b→Bool`.

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. In the following artificial example, we abstract away the particular resulting location.

```
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`. Next consider a slightly less artificial, toy example of computer hardware configuration. It illustrates many aspects of existential types in InvarGenT. We introduce functions `config_mem_board` and `config_gpu` as external, i.e., ones whose definition is not type-checked by InvarGenT. Their types illustrate that existential types can be used in type annotations. Types `Slow` and `Fast`, although declared as data-types, are phantom types, i.e. are not inhabited and convey information as parameters of other types.

```
datatype Slow  datatype Fast  datatype Budget
datacons Small : Budget  datacons Medium : Budget  datacons Large : Budget
datatype Memory : type  datacons Best_mem : Memory Fast
```

```
datatype Motherboard : type  datacons Best_board : Motherboard Fast
external config_mem_board : Budget → ∃a. (Memory a, Motherboard a)
datatype CPU : type
datacons FastCPU : CPU Fast  datacons SlowCPU : CPU Slow
datatype GPU : type
datacons FastGPU : GPU Fast  datacons SlowGPU : GPU Slow
external config_gpu : Budget → ∃a. GPU a
datatype PC : type * type * type * type
datacons PC :
  ∀a,b,c,r. CPU a * GPU b * Memory c * Motherboard r ⟶ PC (a,b,c,r)
datatype Usecase  datacons Gaming : Usecase
datacons Scientific : Usecase  datacons Office : Usecase

let budget_to_cpu = efunction
  | Small -> SlowCPU | Medium -> FastCPU | Large -> FastCPU
let usecase_to_gpu budget = efunction
  | Gaming -> FastGPU | Scientific -> FastGPU
  | Office -> config_gpu budget
let rec configure = efunction
  | Small, Gaming -> configure (Small, Office)
  | Large, Gaming -> PC (FastCPU, FastGPU, Best_mem, Best_board)
  | budget, usecase ->
    let mem, board = config_mem_board budget in
    let cpu = budget_to_cpu budget in
    let gpu = usecase_to_gpu budget usecase in
    PC (cpu, gpu, mem, board)
```

InvarGenT infers the following types:

```
budget_to_cpu : Size → ∃a.CPU a
usecase_to_gpu : Usecase → ∃a.GPU a
configure : (Size, Usecase) → ∃a, b, c.PC (a, b, c, c)
```

The definition of `configure` illustrates explicit elimination of existential types by `let..in` definitions: by design, inlining of `cpu` or `gpu` definitions would make the function not typeable. The call to `config_gpu` and the recursive call to `configure` illustrate implicit elimination of existential types in return positions.

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 f =
  efunction LNil -> LNil
    | LCons (x, xs) ->
      eif f x then
          let ys = filter f xs in
          LCons (x, ys)
      else filter f xs
```

We get `filter`: `∀n, a.(a→Bool)→List (a, n)→ ∃k[0≤k ∧ k≤n].List (a, k)`. Note that we need to use both `efunction` and `eif` above, since every use of `function`, `match` or `if` 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 `if`, 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	$\tau_1 \rightarrow \tau_2$	`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. K_T stands for `True`, K_F for `False`, and K_u for `()`.

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`
if-then-else clause	$\lambda(K_T.e_1, K_F.e_2)\, e$	`if e then e1 else e2`
if-then-else condition	$\lambda(\_\ \mathbf{when}\, m \leqslant n.e_1, ...)\, K_u$	`if m <= n then e1 else e2`
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 | ...`
eif-then-else clause	$\lambda[K](K_T.e_1, K_F.e_2)\, e$	`eif e then e1 else e2`
eif-then-else condition	$\lambda[K](\_\ \mathbf{when}\, m \leqslant n.e_1, ...)\, K_u$	`eif m <= n then e1 else e2`
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.	**assert false**	`assert false`
runtime failure	**runtime failure** s	`runtime_failure s`
assert equal types	**assert type** $\tau_{e_1} \doteq \tau_{e_2}; e_3$	`assert type e1 = e2; e3`
assert inequality	**assert num** $e_1 \leqslant e_2; e_3$	`assert num e1 <= e2; e3`

A built-in fail at runtime with the given text message is only needed for introducing existential types: a user-defined equivalent of `runtime_failure` would introduce a spurious branch for generalization.

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`

Toplevel non-recursive `let` definitions are polymorphic as an exception. In expressions, `let...in` definitions are monomorphic, one should use the `let rec...in` syntax to get a polymorphic `let`-binding.

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.

There are variants of the if-then-else clause syntax supporting **when** conditions:

- `if m1 <= n1 && m2 <= n2 && ... then e1 else e2` is $\lambda(\_\ \mathbf{when} \wedge_i m_i \leqslant n_i.e_1, \_.e_2)\, K_u$,
- `if m <= n then e1 else e2` is $\lambda(\_\ \mathbf{when}\, m \leqslant n.e_1, \_\ \mathbf{when}\, n + 1 \leqslant m.e_2)\, K_u$ if integer mode is on (as in default setting),
- similarly for the `eif` variants.

We add the standard syntactic sugar for function definitions:

- `let` $p_1\ p_2\ ...\ p_n$ `=` e_1 `in` e_2 expands to `let` p_1 `= fun` $p_2\ ...\ p_n$ `->` e_1 `in` e_2
- `let rec` $l_1\ p_2\ ...\ p_n$ `=` e_1 `in` e_2 expands to `let rec` l_1 `= fun` $p_2\ ...\ p_n$ `->` e_1 `in` e_2
- top-level `let` and `let rec` definitions expand correspondingly.

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.

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.

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/avl_tree.gadt
```

Below we demonstrate what happens with insufficiently high parameter setting. 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 -term_abduction_timeout 100 examples/equal_assert.gadt
File "examples/equal_assert.gadt", line 19, characters 24-38:
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.

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. *Constraint generalization* 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, constraint generalization, 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.

`-allow_dead_code`. Allow more programs with dead code than would otherwise pass.

`-force_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.

`-convergence_step`. The iteration at which to start truncating postconditions by only keeping atoms present in the previous iteration, to force convergence (default 8).

`-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. However, it often succeeds because of other mechanisms used by the abduction algorithm.

`-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).

`-affine_penalty`. How much to penalize an abduction candidate inequality for containing a constant term (default 4). Too small a value may lead to divergence, e.g. in some examples abduction will pick an answer $a+1$, which in the following step will force an answer $a+2$, then $a+3$, etc.

`-reward_constrn`. How much to reward introducing a constraint on so-far unconstrained varialbe, or penalize if negative (default 2).

`-complexity_penalty`. How much to penalize an abduction candidate inequality for complexity of its coefficients; the coefficient of either the linear or power scaling of the coefficients (default 2.5).

`-abd_lin_thres_scaling`. Scale the complexity cost of coefficients linearly with a jump of the given height after coefficient 1 (default 2.0).

`-abd_pow_scaling`. Scale the complexity cost of coefficients according to the given power.

`-prefer_bound_to_local`. Prefer a bound coming from outer scope, to inequality between two local parameters. In numerical abduction heuristic, such bounds are usually doubly penalized: for having a constant, and non-locality of parameters.

`-prefer_bound_to_outer`. Prefer a bound coming from outer scope, to inequality between two outer scope parameters. Outer-scope constraints sometimes lead to an answer not general enough.

`-only_off_by_1`. Limit the effect of `-prefer_bound_to_local` and `-prefer_bound_to_outer` to inequalities with a constant 1. This corresponds to an upper bound of an index into a zero-indexed array/matrix/etc.

`-same_with_assertions`. Do not treat definitions with positive assertions (`assert num`, `assert type`) specially. The special treatment is currently equivalent to passing `-reward_constrn -1` and `-prefer_bound_to_local`.

`-concl_abd_penalty`. Penalize abductive guess when the supporting argument comes from the partial answer, instead of from the current premise (default 4). Guesses involving the partial answer are less secure, for example they depend on the order in which the constraint to explain is being processed.

`-more_general_num`. Filter out less general abduction candidate atoms (does not guarantee overall more general answers). The filtering is currently not performed by default to save on computational cost.

`-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 postconditions, 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.

`-if_else_no_when`. Do not add `when` clause to the `else` branch of an `if` expression with a single inequality as condition. Expressions `if`, resp. `eif`, with a single inequality as the condition are expanded into expressions `match`, resp. `ematch`, with `when` conditions on both the `True` branch and the `False` branch. I.e. `if m <= n then e1 else e2` is expanded into `match () with _ when m <= n -> e1 | _ when n+1 <= m -> e2`. Passing `-if_else_no_when` will result in expansion `match () with _ when m <= n -> e1 | _ -> e2`. The same effect can be achieved for a particular expression by artificially incresing the number of inequalities: `if m <= n && m <= n then e1 else e2`.

`-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.

-postcond_opti_limit. Limit the number of atoms $x = \min(a, b)$, $x = \max(a, b)$ in (intermediate and final) postconditions (default 4). Unfortunately, inference time is exponential in the number of atoms of this form. The final postconditions usually have few of these atoms, but a greater number is sometimes needed in the intermediate steps of the main loop.

-postcond_subopti_limit. Limit the number of atoms $\min(a, b) \leqslant x$, $x \leqslant \max(a, b)$ in (intermediate and final) postconditions (default 4). Unfortunately, inference time is exponential in the number of atoms of this form. The final postconditions usually have few of these atoms, but a greater number is sometimes needed in the intermediate steps of the main loop.

-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.

-prefer_guess. Try to guess equality-between-parameters before considering other possibilities. Implied by -richer_answers but less invasive.

-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 have a look at tests from the examples direcotory that need a non-default parameter setting.

```
$ ./invargent -inform examples/non_pointwise_leq.gadt
File "examples/non_pointwise_leq.gadt", line 12, characters 14-60:
No answer in type: Answers do not converge

Perhaps increase the -iterations_timeout parameter or try the -more_existential
option or -prefer_guess option.
$ ./invargent -inform -prefer_guess examples/non_pointwise_leq.gadt
val leq : ∀a. Nat a → NatLeq (a, a)
InvarGenT: Generated file examples/non_pointwise_leq.gadti
InvarGenT: Generated file examples/non_pointwise_leq.ml
InvarGenT: Command "ocamlc -w -25 -c examples/non_pointwise_leq.ml" exited with
code 0
```

Other examples that need the `-prefer_guess` option: `non_pointwise_zip1_simpler.gadt`, `non_pointwise_zip1_simpler2.gadt`, `non_pointwise_zip1_modified.gadt`. On the other hand, `non_pointwise_zip1.gadt` is inferred with default settings.

The response from the system does not always include an option which would make the inference succeed.

```
$ ./invargent -inform examples/liquid_simplex_step_3a.gadt
File "examples/liquid_simplex_step_3a.gadt", line 7, characters 49-1651:
No answer in type: Answers do not converge

Perhaps do not pass the -no_dead_code flag.
Perhaps increase the -iterations_timeout parameter or try one of the options:
-more_existential, -prefer_guess, -prefer_bound_to_local.
Perhaps some definition is used with requirements on
its inferred postcondition not warranted by the definition.
$ ./invargent -inform -prefer_bound_to_local -only_off_by_1 \
examples/liquid_simplex_step_3a.gadt
val main_step3_test : ∀k, n[1 ⩽ n ∧ 3 ⩽ k]. Matrix (n, k) → Float
InvarGenT: Generated file examples/liquid_simplex_step_3a.gadti
InvarGenT: Generated file examples/liquid_simplex_step_3a.ml
File "examples/liquid_simplex_step_3a.ml", line 43, characters 8-9:
Warning 26: unused variable m.
InvarGenT: Command "ocamlc -w -25 -c examples/liquid_simplex_step_3a.ml" exited
with code 0
```

The other examples that need the `-prefer_bound_to_local` option, but not the `-only_off_by_1` option: `liquid_simplex_step_6a_2.gadt`, `liquid_tower_harder.gadt`.

```
$ ./invargent -inform examples/pointwise_zip2_harder.gadt
val zip2 : ∀a, b. Zip2 (a, b) → a → b
InvarGenT: Generated file examples/pointwise_zip2_harder.gadti
InvarGenT: Generated file examples/pointwise_zip2_harder.ml
File "examples/pointwise_zip2_harder.ml", line 19, characters 21-32:
Error: This kind of expression is not allowed as right-hand side of ‘let rec’
InvarGenT: Command "ocamlc -w -25 -c examples/pointwise_zip2_harder.ml" exited
with code 2
InvarGenT: Regenerated file examples/pointwise_zip2_harder.ml
File "examples/pointwise_zip2_harder.ml", line 21, characters 21-32:
Error: This kind of expression is not allowed as right-hand side of ‘let rec’
InvarGenT: Command "ocamlc -w -25 -c examples/pointwise_zip2_harder.ml" exited
with code 2
$ ./invargent -inform -no_ml examples/pointwise_zip2_harder.gadt
val zip2 : ∀a, b. Zip2 (a, b) → a → b
```

```
InvarGenT: Generated file examples/pointwise_zip2_harder.gadti
```

The example `pointwise_zip2_harder.gadt` is not compatible with the pass-by-value semantics. We can avoid the complaint of the OCaml compiler by passing either the `-no_ml` flag or the `-no_verif` flag. More interestingly, we can notice that the file `pointwise_zip2_harder.ml` is generated twice. This happens because InvarGenT, noticing the failure, generates an OCaml source with more type information, as if the `-full_annot` option was used.

The examples `liquid_fft_simpler.gadt` and `liquid_fft_full_asserted.gadt` contain assertions, but are nearly as hard as `liquid_fft.gadt`, `liquid_fft_full.gadt` respectively. They need the option `-same_with_assertions` to not switch to settings tuned for cases where assertions capture the harder aspects of the invariants to infer.

Unfortunately, inference fails for some examples regardless of parameters setting. We discuss them in the next section.

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 example `pointwise_head.gadt` fails because of the limitations of the `type` sort in representing disequalities.

```
datatype Z
datatype S : type
datatype List : type * num
datacons LNil : ∀a. List(a, Z)
datacons LCons : ∀a, b. a * List(a, b) ⟶ List(a, S b)

let head = function
  | LCons (x, _) -> x
  | LNil -> assert false
```

If we omit the `LNil` branch, we get the technically correct but inadequate type `∀a, b. List(a, b) → a`, because the type system does not guarantee exhaustiveness of the pattern matching. The intended type is `∀a, b. List(a, S b) → a`.

The example `non_pointwise_fd_comp_harder.gadt` is inferred an insufficiently general type `∀a, b. FunDesc (b, b) → FunDesc (b, a) → FunDesc (b, a)`.

```
datatype FunDesc : type * type
datacons FDI : ∀a. FunDesc (a, a)
```

```
datacons FDC : ∀a, b. b ⟶ FunDesc (a, b)
datacons FDG : ∀a, b. (a → b) ⟶ FunDesc (a, b)
external fd_fun : ∀a, b. FunDesc (a, b) → a → b

let fd_comp fd1 fd2 =
  let o f g x = f (g x) in
  match fd1 with
    | FDI -> fd2
    | FDC b ->
      (match fd2 with
        | FDI -> fd1
        | FDC c -> FDC (fd_fun fd2 b)
        | FDG g -> FDC (fd_fun fd2 b))
    | FDG f ->
      (match fd2 with
        | FDI -> fd1
        | FDC c -> FDC c
        | FDG g -> FDG (o (fd_fun fd2) f))
```

This happens because the second argument `fd2` is not expanded when `fd1` is equal to `FDI`. Type inference cannot carry out the different reasoning steps leading to the more general type.

In the example `liquid_bsearch2_harder4.gadt` it turns out to be too hard to infer the full postcondition.

```
datatype Array : type * num
external let array_make :
  ∀n, a [0≤n]. Num n → a → Array (a, n) = "fun a b -> Array.make a b"
external let array_get :
∀n, k, a [0≤k ∧ k+1≤n]. Array (a, n) → Num k → a = "fun a b -> Array.get a b"
external let array_length :
  ∀n, a [0≤n]. Array (a, n) → Num n = "fun a -> Array.length a"
datatype LinOrder
datacons LE : LinOrder
datacons GT : LinOrder
datacons EQ : LinOrder
external let compare : ∀a. a → a → LinOrder =
  "fun a b -> let c = Pervasives.compare a b in
              if c < 0 then LE else if c > 0 then GT else EQ"
external let equal : ∀a. a → a → Bool = "fun a b -> a = b"
external let div2 : ∀n. Num (2 n) → Num n = "fun x -> x / 2"

let bsearch key vec =
  let rec look key vec lo hi =
    eif lo <= hi then
        let m = div2 (hi + lo) in
        let x = array_get vec m in
        ematch compare key x with
          | LE -> look key vec lo (m + (-1))
          | GT -> look key vec (m + 1) hi
          | EQ -> eif equal key x then m else -1
    else -1 in
  look key vec 0 (array_length vec + (-1))
```

We get the result type $\exists$`n[0 ⩽ n + 1].Num n` instead of $\exists$`k[k ⩽ n ∧ 0 ⩽ k + 1].Num k`. The inference of the intended type succeeds after we introduce an appropriate assertion, e.g. `assert num -1 <= hi`. Alternatively, we could include a use case for `bsearch` where the full postcondition is required.

Examples liquid_simplex_step_3.gadt, liquid_simplex_step_4.gadt and liquid_gauss_rowMax.gadt result in uninformative, empty postconditions, because to tell more would require inspecting the behavior of the respective function across recursive calls. To save space, we list just the function definition from liquid_simplex_step_3.gadt:

```
let rec enter_var arr2 n j c j' =
  eif j' + 2 <= n then
    let c' = matrix_get arr2 0 j' in
    eif less c' c then enter_var arr2 n j' c' (j'+1)
    else enter_var arr2 n j c (j'+1)
  else j
```

Fortunately, if the function is used in the same toplevel definition in which it is defined, use-site requirements facilitate the inference of the intended postcondition.

The example liquid_gauss_harder.gadt poses too big a challenge for InvarGenT. To get it pass the inference, we streamline one of the nested definitions, to not introduce another, unnecessary level of nesting. This gives the example liquid_gauss2.gadt, which needs to be run with the option -prefer_bound_to_local. Additionally, we can relax the constraint on the processed portion of the matrix, coming from the restriction on the matrix size intended in the original source of the FFT examples. In liquid_gauss.gadt, the whole matrix is processed and the inferred type is most general, under the default settings – no need to pass any options to InvarGenT. The reason liquid_gauss_harder.gadt is too difficult for InvarGenT is that the nesting interferes with the propagation of use-site constraints to the postcondition of the nested definition (the loop inside rowMax). Inference works for liquid_gauss_harder_asserted.gadt, because the assertion provides the required information to infer the rowMax invariants directly.