InvarGenT: Implementation

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 implementation documentation focuses on source code, refers to separate technical reports on theory and algorithms.

1 Data Structures and Concrete Syntax

Following [1], we have the following nodes in the abstract syntax of patterns and expressions:

`p-Empty.` 0: Pattern that never matches. Concrete syntax: `!`. Constructor: `Zero`.

`p-Wild.` 1: Pattern that always matches. Concrete syntax: `_`. Constructor: `One`.

`p-And.` $p_1 \wedge p_2$: Conjunctive pattern. Concrete syntax: e.g. `p1 as p2`. Constructor: `PAnd`.

`p-Var.` x: Pattern variable. Concrete syntax: lower-case identifier e.g. `x`. Constructor: `PVar`.

`p-Cstr.` $K p_1 \ldots p_n$: Constructor pattern. Concrete syntax: e.g. `K (p1, p2)`. Constructor: `PCons`.

`Var.` x: Variable. Concrete syntax: lower-case identifier e.g. `x`. Constructor: `Var`. External functions are represented as variables in global environment.

`Cstr.` $K e_1 \ldots e_n$: Constructor expression. Concrete syntax: e.g. `K (e1, e2)`. Constructor: `Cons`.

`App.` $e_1\, e_2$: Application. Concrete syntax: e.g. `x y`. Constructor: `App`.

`LetRec.` $\mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2$: Recursive definition. Concrete syntax: e.g. `let rec f = function ... in ...` Constructor: `Letrec`.

`Abs.` $\lambda(c_1 \ldots c_n)$: Function defined by cases. Concrete syntax: for single branching via `fun` keyword, e.g. `fun x y -> f x y` translates as $\lambda(x.\lambda(y.(f x)\, y))$; for multiple branching via `match` keyword, e.g. `match e with ...` translates as $\lambda(\ldots)\, e$. Constructor: `Lam`.

`Clause.` $p.e$: Branch of pattern matching. Concrete syntax: e.g. `p -> e`.

`CstrIntro.` Does not figure in neither concrete nor abstract syntax. Scope of existential types is thought to retroactively cover the whole program.

`ExCases.` $\lambda[K](p_1.e_1 \ldots p_n.e_n)$: Function defined by cases and abstracting over the type of result. Concrete syntax: `function` and `ematch` keywords – e.g. `function Nil -> ... | Cons (x,xs) -> ...`; `ematch l with ...` Parsing introduces a fresh identifier for K. Constructor: `ExLam`.

`ExLetIn.` $\mathbf{let}\; p = e_1\; \mathbf{in}\; e_2$: Elimination of existentially quantified type. Concrete syntax: e.g. `let v = f e ... in ...` Constructor: `Letin`.

We also have one sort-specific type of expression, numerals.

For type and formula connectives, we have ASCII and unicode syntactic variants (the difference is only in lexer). Quantified variables can be space or comma separated. The table below is analogous to information for expressions above. Existential type construct introduces a fresh identifier for K. The abstract syntax of types is not sort-safe, but type variables carry sorts which are inferred after parsing. Existential type occurrence in user code introduces a fresh identifier, a new type constructor in global environment `newtype_env`, and a new value constructor in global environment `newcons_env` – the value constructor purpose is to store the content of the existential type, it is not used in the program.

type variable	x	`x`		`TVar`
type constructor	List	`List`		`TCons(CNamed...)`
number (type)	7	`7`		`NCst`
numeral (expr.)	7	`7`		`Num`
numerical sum (type)	$a+b$	`a+b`		`Nadd`
existential type	$\exists\alpha\beta[a\leqslant\beta].\tau$	`ex a b [a<=b].t`	`∃a,b[a≤b].t`	`TCons(Extype...)`
type sort	s_{ty}	`type`		`Type_sort`
number sort	s_R	`num`		`Num_sort`
function type	$\tau_1\rightarrow\tau_2$	`t1 -> t2`	`t1 → t2`	`Fun`
equation	$a\doteq b$	`a = b`		`Eqty`
inequation	$a\leqslant b$	`a <= b`	`a ≤ b`	`Leq`
conjunction	$\varphi_1\wedge\varphi_2$	`a=b && b=a`	`a=b ∧ b=a`	built-in lists

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`	`TypConstr`
value constructor	`newcons Cons : all n a. a * List(a,n) --> List(a,n+1)`	`ValConstr`
	`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)`	`PrimVal`
rec. definition	`let rec f =...`	`LetRecVal`
non-rec. definition	`let v =...`	`LetVal`

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.

2 Generating and Normalizing Formulas

We inject the existential type and value constructors during parsing for user-provided existential types, and during constraint generation for inferred existential types, into the list of toplevel items, which allows to follow [1] despite removing `extype` construct from the language. It also faciliates exporting inference results as OCaml source code.

Functions `constr_gen_pat` and `envfrag_gen_pat` compute formulas according to table 2 in [1], and `constr_gen_expr` computes table 3. Due to the presentation of the type system, we ensure in `ValConstr` that bounds on type parameters are introduced in the formula rather than being substituted into the result type. We preserve the FOL language presentation in the type `cnstrnt`, only limiting the expressivity in ways not requiring any preprocessing. The toplevel definitions (from type `struct_item`) `LetRecVal` and `LetVal` are processed by `constr_gen_letrec` and `constr_gen_let` respectively. They are analogous to `Letrec` and `Letin` or a `Lam` clause. We do not cover toplevel definitions in our formalism (without even a rudimentary module system, the toplevel is a matter of pragmatics rather than semantics).

Toplevel definitions are intended as boundaries for constraint solving. This way the programmer can decompose functions that could be too complex for the solver. `LetRecVal` only binds a single identifier, while `LetVal` binds variables in a pattern. To preserve the flexibility of expression-level pattern matching, `LetVal` has to pack the constraints $[\![\Sigma\vdash p\uparrow\alpha]\!]$ which the pattern makes available, into existential types. Each pattern variable is a separate entry to the global environment, therefore the connection between them is lost.

The formalism (in interests of parsimony) requires that only values of existential types be bound using `Letin` syntax. The implementation is enhanced in this regard: if the normalization step cannot determine which existential type is being eliminated, the constraint is replaced by one that would be generated for a pattern matching branch. This recovers the common use of the `let...in` syntax, with exception of polymorphic `let` cases, where `let rec` still needs to be used.

In the formalism, we use $\mathcal{E} = \{\varepsilon_K, \chi_K | K :: \forall\alpha\gamma[\chi_K(\alpha, \gamma)].\gamma \rightarrow \varepsilon_K(\alpha) \in \Sigma\}$ for brevity, as if all existential types $\varepsilon_K(\alpha)$ were related with a predicate variable $\chi_K(\alpha, \gamma)$. In the implementation, we have user-defined existential types with explicit constraints in addition to inferred existential types. We keep track of existential types in cell `ex_types`, storing arbitrary constraints. For `LetVal`, we form existential types after solving the generated constraint, to have less intermediate variables in them. The first argument of the predicate variable $\chi_K(\alpha, \gamma)$ provides an "escape route" for free variables, e.g. precondition variables used in postcondition. It is used for convenience in the formalism. In the implementation, after the constraints are solved, we expand it to pass each free variable as a separate parameter, to increase readability of exported OCaml code.

For simplicity, only toplevel definitions accept type and invariant annotations from the user. The constraints are modified according to the $[\![\Gamma, \Sigma \vdash \text{ce}: \forall\bar{\alpha}\,[D].\tau]\!]$ rule. Where `Letrec` uses a fresh variable β, `LetRecVal` incorporates the type from the annotation. The annotation is considered partial, D becomes part of the constraint generated for the recursive function but more constraints will be added if needed. The polymorphism of $\forall\bar{\alpha}$ variables from the annotation is preserved since they are universally quantified in the generated constraint.

The constraints solver returns three components: the *residue*, which implies the constraint when the predicate variables are instantiated, and the solutions to unary and binary predicate variables. The residue and the predicate variable solutions are separated into *solved variables* part, which is a substitution, and remaining constraints (which are currently limited to linear inequalities). To get a predicate variable solution we look for the predicate variable identifier association and apply it to one or two type variable identifiers, which will instantiate the parameters of the predicate variable. We considered several ways to deal with multiple solutions:

1. report a failure to the user;

2. ask the user for decision;

3. perform backtracking search for the first solution that satisfies the subsequent program.

We use an enhanced variant of approach 1 as it is closest to traditional type inference workflow. Upon "multiple solutions" failure the user can add `assert` clauses (e.g. `assert false` stating that a program branch is impossible), and `test` clauses. The `test` clauses are boolean expressions with operational semantics of run-time tests: the test clauses are executed right after the definition is executed, and run-time error is reported when a clause returns `false`. The constraints from test clauses are included in the constraint for the toplevel definition, thus propagate more efficiently than backtracking would. The `assert` clauses are: `assert = type e1 e2` which translates as equality of types of `e1` and `e2`, `assert false` which translates as `CFalse`, and `assert e1 <= e2`, which translates as inequality $n_1 \leqslant n_2$ assuming that `e1` has type `Num n1` and `e2` has type `Num n2`.

2.1 Normalization

Rather than reducing to prenex-normal form as in our formalization, we preserve the scope relations and return a `var_scope`-producing variable comparison function. Also, since we use `let-in` syntax to both eliminate existential types and for traditional (but not polymorphic) binding, we drop the `Or1` constraints (in the formalism they ensure that `let-in` binds an existential type). During normalization, we compute unifiers of the type sort part of conclusions. This faciliates solving of the disjunctions in `ImplOr2` constraints. We monitor for contradiction in conclusions, so that we can stop the `Contradiction` exception and remove an implication in case the premise is also contradictory.

Releasing constraints from under `ImplOr2`, when the corresponding `let-in` is interpreted as standard binding (instead of eliminating existential type), violates declarativeness. We cannot include all the released constraints in determining whether non-nested `ImplOr2` constraints should be interpreted as eliminating existential types. While we could be more "aggresive" (we can modify the implementation to release the constraints one-by-one), it shouldn't be problematic, because nesting of `ImplOr2` corresponds to nesting of `let-in` definitions.

After normalization, we simplify the constraints by removing redundant atoms. We remove atoms that bind variables not occurring anywhere else in the constraint, and in case of atoms not in premises, not universally quantified. The simplification step is not currently proven correct and might need refining.

3 Abduction

The formal specification of abduction in [7] provides a scheme for combining sorts that substitutes number sort subterms from type sort terms with variables, so that a single-sort term abduction algorithm can be called. Since we implement term abduction over the two-sorted datatype `typ`, we keep these *alien subterms* in terms passed to term abduction.

3.1 Simple constraint abduction for terms

Our initial implementation of simple constraint abduction for terms follows [?] p. 13. The mentioned algorithm only gives *fully maximal answers* which is loss of generality w.r.t. our requirements. To solve $D \Rightarrow C$ the algorithm starts with with $\boldsymbol{U}(D \wedge C)$ and iteratively replaces subterms by fresh variables $\alpha \in \bar{\alpha}$ for a final solution $\exists \bar{\alpha}.A$. To mitigate some of the limitations of fully maximal answers, we start from $\boldsymbol{U}_{\bar{\alpha}}(A(D \wedge C))$, where $\exists \bar{\alpha}.A$ is the solution to previous problems solved by the joint abduction algorithm, and $A(\cdot)$ is the corresponding substitution. We follow top-down approach where bigger subterms are abstracted first – replaced by fresh variable, together with an arbitrary selection of other occurrences of the subterm. If replacing a subterm by fresh variable maintains $T(F) \vDash A \wedge D \Rightarrow C$, we proceed to neighboring subterm or next equation. If $T(F) \vDash A \wedge D \Rightarrow C$ does not hold, we try all of: proceeding to subterms of the subterm; replacing the subterm by the fresh variable; replacing the subterm by variables corresponding to earlier occurrences of the subterm. This results in a single, branching pass over all subterms considered. TODO: avoiding branching when implication holds might lead to another loss of generality, does it? Finally, we clean up the solution by eliminating fresh variables when possible (i.e. substituting-out equations $x \doteq \alpha$ for variable x and fresh variable α).

Although our formalism stresses the possibility of infinite number of abduction answers, there is always finite number of *fully maximal* answers that we compute. The formalism suggests computing them lazily using streams, and then testing all combinations – generate and test scheme. Instead, we use a search scheme that tests as soon as possible. The simple abduction algorithm takes a partial solution – a conjunction of candidate solutions for some other branches – and checks if the solution being generated is satisfiable together with the candidate partial solution. The algorithm also takes a number that determines how many correct solutions to skip.

3.2 Joint constraint abduction for terms

We further lose generality by using a heuristic search scheme instead of testing all combinations of simple abduction answers. If natural counterexamples are found, rather than ones contrived to demonstrate that our search scheme is not complete, it can be augmented. In particular, our search scheme returns from joint abduction for types with a single answer, which eliminates any interaction between the sort of types and other sorts.

We maintain an ordering of branches. We accumulate simple abduction answers into the partial abduction answer until we meet branch that does not have any answer satisfiable with the partial answer so far. Then we start over, but put the branch that failed in front of the sequence. If a branch i is at front for n_ith time, we skip the initial $n_i - 1$ simple abduction answers in it. If the front branch i does not have at least n_i answers, the search fails.

As described in [?], to check validity of answers, we use a modified variant of unification under quantifiers: unification with parameters, where the parameters do not interact with the quantifiers and thus can be freely used and eliminated. Note that to compute conjunction of the candidate answer with a premise, unification does not check for validity under quantifiers.

Because it would be difficult to track other sort constraints while updating the partial answer, we discard numeric sort constraints in simple abduction algorithm, and recover them after the final answer for terms (i.e. for the type sort) is found.

3.3 Joint constraint abduction for linear arithmetic

We use *Fourier-Motzkin elimination*. To avoid complexities we initially only handle rational number domain, but if need arises we will extend to integers using *Omega-test* procedure as presented in [?]. The major operations are:

- *Elimination* of a variable takes an equation and selects a variable that isn't upstream of any other variable of the equation, and substitutes-out this variable from the rest of the constraint. The solved form contains an equation for this variable.
- *Projection* of a variable takes a variable x that isn't upstream of any other variable in the unsolved part of the constraint, and reduces all inequalities containing x to the form $x \dot{\leqslant} a$ or $b \dot{\leqslant} x$, depending on whether the coefficient of x is positive or negative. For each such pair of inequalities: if $b = a$, we add $x \doteq a$ to implicit equalities; otherwise, we add the inequality $b \dot{\leqslant} a$ to the unsolved part of the constraint.

We use elimination to solve all equations before we proceed to inequalities. The starting point of our algorithm is [?] section *4.2 Online Fourier-Motzkin Elimination for Reals*. We add detection of implicit equalities, and more online treatment of equations, introducing known inequalities on eliminated variables to the projection process.

There are usually infinitely many answers to the simple constraint abduction problem whenever equations or implicit equalities are involved. The algorithm we develop follows our presentation in [?], but only considers answers achieved from canonical answers (cf. [?]) by substitution of some occurrences of variables according to some equations in the premise.

When we derive a substitution from a set of equations, we eliminate variables that are maximally downstream, and using a fixed total order among variables in the same quantifier alternation. Algorithm:

1. Let $A_i = A_i^= \wedge A_i^{\leqslant}$ be the answer to previous SCA problems where $A_i^=$ are equations and $A_i^{\leqslant}$ are inequalities, and $D \Rightarrow C$ be the current problem.
2. Let $D^{\doteq} \wedge D^= \wedge D^{\leqslant} = A_i^=(D)$, $C^{\doteq} \wedge C^= \wedge C^{\leqslant} = A_i^=(C)$ and $\mathrm{DC}^{\doteq} \wedge \mathrm{DC}^= \wedge \mathrm{DC}^{\leqslant} = A_i^=(D \wedge C)$, where $D^{\doteq}$, resp. $C^{\doteq}$, $\mathrm{DC}^{\doteq}$ are equations, $D^=$, resp. $C^=$, $\mathrm{DC}^=$ are implicit equalities of $A_i^=(D)$, resp. $A_i^=(C)$, $A_i^=(D \wedge C)$.
3. Let $\theta = \mathrm{DC}^{\doteq} \wedge \mathrm{DC}^=$. Let $D' = \theta(D^{\leqslant})$ and $C' = \theta(\mathrm{DC}^{\leqslant})$, where $\theta(\cdot)$ is the substitution corresponding to θ.
4. Let $A^{\leqslant}$ be a core of C' w.r.t. D'. (Choice point 1.)
5. Let $A^= = [D^{\doteq} \wedge D^=](\theta)$, where $[D^{\doteq} \wedge D^=](\cdot)$ is a substitution corresponding to equations in $D^{\doteq} \wedge D^=$.
6. Let $A^{='}$ resp. $A^{\leqslant'}$ be $A^=$ resp. $A^{\leqslant}$ with some occurrences of variables substituted according to some equations in $D^{\doteq} \wedge D^=$, but disregarding the order of variables. (Choice point 2.)
7. The answers are $A_{i+1} = A_i \wedge A^{\leqslant'} \wedge A^{='}$.

Actually in the initial implementation, in step (6) we discard even more solutions. Rather than replacing some occurrences of variables in a given choice, we perform a full substitution: either replace all occurrences using a given equation, or none. We might revert to a more thorough exploration as descirbed in step (6), similar to choices made in abduction for terms. First we need to collect a library of test cases.

We use the `nums` library for exact precision rationals.

4 Disjunction Elimination

Disjunction elimination answers are the maximally specific conjunctions of atoms that are implied by each of a given set of conjunction of atoms. In case of term equations the disjunction elimination algorithm is based on the *anti-unification* algorithm. In case of linear arithmetic inequalities, disjunction elimination is exactly finding the convex hull of a set of possibly unbounded polyhedra. We roughly follow [?], but depart from the algorithm presented there because we employ our unification algorithm to separate sorts. Since as a result we do not introduce variables for *alien subterms*, we include the variables introduced by anti-unification in constraints sent to disjunction elimination for their respective sorts.

The adjusted algorithm looks as follows:

1. Let $\wedge_s D_{i,s} \equiv \boldsymbol{U}(D_i)$ where $D_{i,s}$ is of sort s, be the result of our sort-separating unification.
2. For the sort s_{ty}:
   a. Let $V = \{x_j, \overline{t_{i,j}} \, | \forall i \exists t_{i,j}. x_j \doteq t_{i,j} \in D_{i,s_{\mathrm{ty}}}\}$.
   b. Let $G = \{\bar{\alpha}_j, u_j, \overline{\theta_{i,j}} \, | \theta_{i,j} = [\bar{\alpha}_j := \bar{g}_j^i], \theta_{i,j}(u_j) = t_{i,j}\}$ be the most specific anti-unifiers of $\overline{t_{i,j}}$ for each j.
   c. Let $D_i^u = \wedge_j \bar{\alpha}_j \doteq \bar{g}_j^i$ and $D_i^g = D_{i,s_{\mathrm{ty}}} \wedge D_i^u$.
   d. TO BE CONTINUED. Let $D_i^v = \{x \doteq y | x \doteq t_1 \in D_i^g, y \doteq t_2 \in D_i^g, D_i^g \wedge_s D_i^{v,s} \vDash t_1 \doteq t_2\}$ (work in other sorts already done in $D_i^{v,s}$).
   e. Let $A_{s_{\mathrm{ty}}} = \wedge_j x_j \doteq u_j \bigcap_i (D_i^g \wedge D_i^v)$ (where conjunctions are treated as sets of conjuncts and equations are ordered so that only one of $a \doteq b$, $b \doteq a$ appears anywhere), and $\bar{\alpha}_{s_{\mathrm{ty}}} = \overline{\bar{\alpha}_j}$.
   f. Let $\wedge_s D_{i,s}^u \equiv D_i^u$ for $D_{i,s}^u \in \mathcal{L}_s$.
3. For sorts $s \neq s_{\mathrm{ty}}$, let $\exists \bar{\alpha}_s. A_s = \mathrm{DisjElim}_s\left(\overline{D_i^s \wedge D_i^{t,s} \wedge D_{i,s}^t \wedge D_{i,s}^u}\right)$.
4. The answer is $\exists \overline{\overline{\alpha_i^j}} \overline{\bar{\alpha}_s} . \wedge_s A_s$.

4.1 Anti-unification

4.2 Extended convex hull

[3] provides a polynomial-time algorithm to find the half-space represented convex hull of closed polytopes. It can be generalized to unbounded polytopes – conjunctions of linear inequalities. Our implementation is inspired by this algorithm but very much simpler, at cost of losing the maximality requirement.

First we find among the given inequalities those which are also the faces of resulting convex hull. The negation of such inequality is not satisfiable in conjunction with any of the polytopes – any of the given sets of inequalities. Next we iterate over *ridges* touching the selected faces: pairs of the selected face and another face from the same polytope. We rotate one face towards the other: we compute a convex combination of the two faces of a ridge. We add to the result those half-spaces whose complements lie outside of the convex hull (i.e. negation of the inequality is unsatisfiable in conjunction with every polytope). For a given ridge, we add at most one face, the one which is farthest away from the already selected face, i.e. the coefficient of the selected face in the convex combination is smallest. We check a small number of rotations, where the algorithm from [3] would solve a linear programming problem to find the rotation which exactly touches another one of the polytopes.

When all variables of an equation $a \doteq b$ appear in all branches D_i, we can turn the equation $a \doteq b$ into pair of inequalities $a \leqslant b \wedge b \leqslant a$. We eliminate all equations and implicit inequalities which contain a variable not shared by all D_i, by substituting out such variables. We pass the resulting inequalities to the convex hull algorithm.

Bibliography

[1] Łukasz Stafiniak. A gadt system for invariant inference. Manuscript, 2012. Available at: http://www.ii.uni.wroc.pl/~lukstafi/pubs/EGADTs.pdf