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 $[\![\Gamma, \Sigma \vdash \mathrm{ce}: \forall\bar{\alpha}[D].\tau]\!]$ 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.

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.

Our initial implementation of simple constraint abduction for terms follows [?] p. 13. It 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 $\exists \bar{\alpha}.A$. We follow top-down approach where bigger subterms are abstracted first – replaced by fresh variable. Subterms of a subterm are tried only when replacing the subterm no longer maintains $T(F) \vDash A \wedge D \Rightarrow C$. Otherwise, the subterm is replaced, together with an arbitrary selection of other occurrences of the subterm. Rather than branching on subsets of replacements and performing the replacements straight away, we perform replacement of a subterm when we encounter the subterm. This results in a single pass over all subterms considered. TODO: this probably leads 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. We decided to compute all answers at once using lists, instead of computing them lazily using streams.

Bibliography

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