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	$\tau_1 \to \tau_2$	`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. 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 (and in future, structure items) 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`, where `let rec` still needs to be used.

Bibliography

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