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 [?], 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`. $K p_1 ... p_n$: 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`. $K e_1 ... e_n$: 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`. $\lambda(c_1 ... c_n)$: Function defined by cases. Concrete syntax: `function ... | ...`; 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 $\lambda(...)\, e$. 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`. $\lambda[K](p_1.e_1 ... p_n.e_n)$: Function defined by cases and abstracting over the type of result. Concrete syntax: `efunction` and `ematch` keywords – e.g. `efunction Nil -> ... | Cons (x,xs) -> ...`; `ematch l with ...` Parsing introduces a fresh identifier for K, but normalization keeps only one K for nested `ExCases` (unless nesting in body of local definition or in argument of an application). Constructor: `ExLam`.

`LetIn, 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. Variables of various sorts have to start with specific letters: `a`,`b`,`c`,`r`,`s`,`t`, resp. `i`,`j`,`k`,`l`,`m`,`n`. Remaining letters are reserved for future sorts. The table below is analogous to information for expressions above. We pass variables environment `gamma` as function parameters. We keep constructors environment `sigma` as a global table. Existential type constructs introduce fresh identifiers K, we remember the identifiers in `ex_types` but store the information in `sigma`. Note that inferred existential types will not be nested, thanks to normalization of nested occurrences of $\lambda[K]$. The abstract syntax of types is not sort-safe, but type variables carry sorts determined by first letter of the variable. In the future, sorts could be inferred after parsing.

type variable: types	$\alpha, \beta, \gamma, \tau$	a,b,c,r,s,t,a1,...		TVar(VNam(Type_sort,...))
type variable: nums	k, m, n	i,j,k,l,m,n,i1,...		TVar(VNam(Num_sort,...))
type constructor	List	List		TCons(CNamed...)
number (type)	7	7		NCst
numeral (expr.)	7	7		Num
numerical sum (type)	$m+n$	m+n		Nadd
existential type	$\exists k, n[k \leqslant n].\tau$	ex k n [k<=n].t	∃k,n[k≤n].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	$k \leqslant n$	k <= n	k ≤ n	Leq
conjunction	$\varphi_1 \wedge \varphi_2$	a=b && b=a	a=b ∧ b=a	built-in lists

Parts of the logic hidden from the user:

unary predicate variable	$\chi(\beta)$	PredVarU(chi,TVar "b",loc)
binary predicate variable	$\chi_K(\gamma, \alpha)$	PredVarB(chi,TVar "g", TVar "a",loc)
not an existential type	$\not\exists(\alpha)$	NotEx(TVar "a",loc)
negation `assert false`	$\boldsymbol{F}$	CFalse loc

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. It faciliates exporting inference results as OCaml source code.

Functions `constr_gen_pat` and `envfrag_gen_pat` compute formulas according to table [?] in [?], and `constr_gen_expr` computes table [?]. 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 `Letin` syntax has two uses: binding values of existential types means "eliminating the quantification" – the programmer has control over the scope of the existential constraint. The second use is if the value is not of existential type, 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` needs to be used.

The second argument of the predicate variable $\chi_K(\gamma, \alpha)$ provides an "escape route" for free variables, i.e. precondition variables used in postcondition. In the implementation, we have user-defined existential types with explicit constraints in addition to inferred existential types. We expand the inferred existential types after they are solved into the fuller format. In the inferred form, the result type has a single parameter δ', without loss of generality because the actual parameters are passed as a tuple type. In the full format we recover after inference, we extract the parameters $\delta' \doteq (\bar{\beta})$, the non-local variables of the existential type, and the partially abstract type $\delta \doteq \tau$, and store them separately, i.e. $\varepsilon_K(\bar{\beta}) = \forall \bar{\beta}\, \exists \bar{\alpha}\, [D].\tau$. The variables $\bar{\beta}$ are instantiated whenever the constructor is used. For `LetVal`, we form existential types after solving the generated constraint, to have less intermediate variables in them.

Both during parsing and during inference, we inject new structure items to the program, which capture the existential types. In parsing, they arise only for `ValConstr` and `PrimVal` and are added by `Praser`. The inference happens only for `LetRecVal` and `LetVal` and injection is performed in `Infer`. During printing existential types in concrete syntax $\exists i{:}\, \bar{\beta}\, [\varphi].t$ for an occurrence $\varepsilon_K((\bar{r}))$, the variables $\bar{\alpha}$ coming from $\delta' \doteq (\bar{\alpha}) \in \varphi$ are substituted-out by $[\bar{\alpha} := \bar{r}\,]$.

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. silently pick one solution, if the wrong one is picked the subsequent program might fail;
4. perform backtracking search for the first solution that satisfies the subsequent program.

We use approach 3 as it is simplest to implement. Traditional type inference workflow rules out approach 2, approach 4 is computationally too expensive. We might use approach 1 in a future version of the system. Upon "multiple solutions" failure – or currently, when a wrong type or invariant is picked – 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`.

We treat a chain of single branch functions with only `assert false` in the body of the last function specially. We put all information about the type of the functions in the premise of the generated constraint. Therefore the user can use them to exclude unintended types. See the example `equal_assert.gadt`.

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. The branches we return from normalization have unified conclusions, since we need them for solving disjunctions anyway.

Releasing constraints from under `Or` is done iteratively, somewhat similar to how disjunction would be treated in constraint solvers. Releasing the sub-constraints is essential for eliminating cases of further `Or` constraints. When at the end more than one disjunct remains, we assume it is the traditional `LetIn` rule and select its disjunct (the first one in the implementation).

When one $\lambda[K]$ expression is a branch of another $\lambda[K]$ expression, the corresponding branch does not introduce an `Or` constraint – the case is settled syntactically to be the same existential type.

2.1.1 Implementation details

The unsolved constraints are particularly weak with regard to variables constrained by predicate variables. We need to propagate which existential type to select for result type of recursive functions, if any. The information is collected from implication branches by `simplify_brs` in `normalize`; it is used by `check_chi_exty`. `normalize` starts by flattening constraints into implications with conjunctions of atoms as premises and conclusions, and disjunctions with disjuncts and additional information. `flat_dsj` is used to flatten each disjunct in a disjunction, the additional information kept is `guard_cnj`, conjunction of atoms that hold together with the disjunction. `solve_dsj` takes the result of `flat_dsj` and tries to eliminate disjuncts. If only one disjunct is left, or we decide to pick `LetIn` anyway (`step>0`), we return it. Otherwise we return the filtered disjunction. `prepare_brs` cleans up the initial flattened constraints or the constraints released from disjunctions: it calls `simplify_brs` on implications and `flat_dsj` on each disjunction.

We collect information about existential return types of recursive definitions in `simplify_brs`:

1. solving the conclusion of a branch together with additional conclusions `guard_cnj`, to know the return types of variables,
2. registering existential return types for all variables in the substitution,
3. traversing the premise and conclusion to find new variables that are types of recursive definitions,
4. registering as "type of recursive definition" the return types for all variables in the substitution registered as types of recursive definitions,
5. traversing all variables known to be types of recursive definitions, and registering existential type with recursive definition (i.e. unary predicate variable) if it has been learned,
6. traversing all variables known to be types of recursive definitions again, and registering existential type of the recursive definition (if any) with the variable.

2.2 Simplification

During normalization, we remove from a nested premise the atoms it is conjoined with (as in "modus ponens").

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. We merge implications with the same premise.

3 Abduction

The formal specification of abduction in [4] 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 [3] 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 $\boldsymbol{U}(A(D \wedge C))$, where $\exists \bar{\alpha}\,.A$ is the solution to previous problems solved by the joint abduction algorithm, and $A(\cdot)$ is the corresponding substitution. During abduction $\mathrm{Abd}(\mathcal{Q}, \bar{\zeta}, \overline{D_i, C_i})$, we ensure that the (partial as well as final) answer $\exists \bar{\alpha}\,.A$ satisfies $\vDash \mathcal{Q}.A[\bar{\alpha}\bar{\zeta} := \bar{t}\,]$ for some $\bar{t}$. We achieve this by normalizing the answer using parameterized unification under quantifiers $\boldsymbol{U}_{\bar{\alpha}\bar{\zeta}}(\mathcal{Q}.A)$. $\bar{\zeta}$ are potential parameters of the invariants.

In fact, when performing unification, we check more than $\boldsymbol{U}_{\bar{\alpha}\bar{\zeta}}(\mathcal{Q}.A)$ requires. We also ensure that the use of already found parameters (represented by $\overline{\beta_\chi \bar{\beta}^{\,\chi}}$ in the main algorithm) will not cause problems in the `split` phase of the main algorithm.

In implementing [3] p. 13, 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 dropping the candidate atom maintains $T(F) \vDash A \wedge D \Rightarrow C$, we proceed to neighboring subterm or next equation. Otherwise, we try all of: replacing the subterm by the fresh variable; proceeding to subterms of the subterm; preserving the subterm; 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, one of which 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 several indicators to let it select the expected answer:

- a number that determines how many correct solutions to skip;
- a validation procedure that checks whether the partial answer meets a condition, in joint abduction the condition is consistency with premise and conclusion of each branch;
- the initial parameters of the invariants (represented by $\overline{\beta_\chi \bar{\beta}^{\,\chi}}$ in the main algorithm) to which all variables of every atom in the answer for which $\mathrm{FV}(c) \cap \bar{\zeta} \neq \varnothing$ should be *connected*;
- the quantifier $\mathcal{Q}$ (source `q`) so that the partial answer $\exists \bar{\alpha}\,.A$ (source `vs,ans`) can be checked for validity with parameters: $\vDash \mathcal{Q}.A[\bar{\alpha} := \bar{t}\,]$ for some $\bar{t}$;
- a discard list of partial answers to avoid (a tabu list) – implements backtracking, with answers from abductions raising "fallback" going there.

To ensure connected variables condition, if an atom $\mathrm{FV}(c) \cap \bar{\zeta} \neq \varnothing$, we only add it to the answer if it contains a connected variable, and since then count all its $\mathrm{FV}(c) \cap \bar{\zeta}$ variables as connected. The initially connected variables are the initial parameters of the invariants. If such atom does not have a connected variable, we move it through the candidates past an atom that would make it connected. If none would, we drop the atom. In terms of the notation from the main algorithm, every $\bar{\zeta}^{\,\chi}$ atom of the answer has to be connected to a $\beta_\chi \bar{\beta}^{\,\chi}$ atom using only the atoms in the answer.

Besides multisort joint constraint abduction `abd` we also provide multisort simple fully maximal constraint abduction `abd_s`.

To recapitulate, the implementation is:

- `abstract` is the entry point.
- If `cand=[]` – no more candidates to add to the partial solution: check for repeated answers, skipping, and discarded answers.
- Otherwise, pick the next atom and check if it's connected, if no, loop with reordered candidates (possibly without the atom).
- `step` works through a single atom of the form $x = t$.
- The `abstract`/`step` choices are:
    1. Try to drop the atom (if the partial answer + remaining candidates can still be completed to a correct answer).
    2. Replace the current subterm of the atom with a fresh parameter, adding the subterm to replacements; if at the root of the atom, check connected and validate before proceeding to remaining candidates.
    3. Step into subterms of the current subterm, if any, and if at the sort of types.
    4. Keep the current part of the atom unchanged; if at the root of the atom, check connected and validate before proceeding to remaining candidates.
    5. Replace the current subterm with a parameter introduced for an earlier occurrence; branch over all matching parameters; if at the root of the atom, check connected and validate before proceeding to remaining candidates.
- Choices 2-5 are guarded by `try-with`, as tests raise `Contradiction` if they fail, choice 1 only tests `implies_concl` which returns a boolean.
- We provide an option `more_general`, which when set to false reorders the choices into: 1, 4, 2, 3, 5 – pushing 4 up minimizes the amount of branching in 5.
- Recompute modifications of parameters due to partial answer, e.g. `cparams`, for clarity of joint constraint abduction; we could compute them incrementally and pass around.
- Form initial candidates $\boldsymbol{U}(A(D \wedge C))$. Revert substitutions $\alpha := \beta$ for $\forall \beta \in \mathcal{Q}$ and $\exists \alpha \in \mathcal{Q}$ to $\beta := \alpha$. This optimization mitigates quantifier violations later, although excludes $\alpha := \beta$ from direct participation in the answer.
    - Replace $\alpha_1 := \beta, ..., \alpha_n := \beta$ with $\beta := \alpha_1, \alpha_2 := \alpha_1, ..., \alpha_n := \alpha_1$.
- Sort the initial candidates by decreasing size.

The above ordering of choices ensures that more general answers are tried first. Moreover:

- choice 1 could be dropped as it is equivalent to choice 2 applied on the root term;
- choices 4 and 5 could be reordered but having choice 4 as soon as possible is important for efficiency.

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. In particular, our search scheme returns from joint abduction for types with a single answer, which eliminates deeper interaction between the sort of types and other sorts. Some amount of interaction is provided by the validation procedure, which checks for consistency of the partial answer, the premise and the conclusion of each branch, including consistency for 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 no front branch i has at least n_i answers, the search fails. After an answer working for all branches has been found, we perform additional check, which encapsulates negative constraints introduced by `assert false` construct. If the check fails, we increase the skip count of the head branch and repeat the search.

When a branch has no more solutions to offer – its skip factor n_i has reached the number of fully maximal solutions to that branch – we move it to a separate *runouts* list and continue search starting from a different branch. We do not increase its skip factor, but we check the runouts directly after the first branch, so that conflicting branches can be located. When the first branch conflicts with the runouts, we increase its skip factor and repeat. We keep a count of conflicts for the runouts so that in case of overall failure, we can report a branch likely to be among those preventing abduction.

We remember SCA answers when skipping over them, not to return the same answer for different skip factors. But we also remember all JCA partial answers that led to resetting. If a partial answer becomes as strong as one of them, we can reset without further checking. If an empty partial answer led to resetting, no answer exists. The failed partial answers are initialized with the discarded answers, passed from the main algorithm.

As described in [7], 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.

When searching for abduction answer fails, we raise exception `Suspect` that contains the partial answer conjoined with conclusion of an implication that failed to produce an answer compatible with remaining implications.

3.3 Abduction for terms with Alien Subterms

The JCAQPAS problem is more complex than simply substituting alien subterms with variables and performing joint constraint abduction on resulting implications. The ability to "outsource" constraints to the alien sorts enables more general answers to the target sort, in our case the term algebra $T(F)$. Term abduction will offer answers that cannot be extended to multisort answers.

One might mitigate the problem by preserving the joint abduction for terms algorithm, and after a solution $\exists\bar{\alpha}.A$ is found, "*dissociating*" the alien subterms (including variables) in A as follows. We replace every alien subterm n_s in A (including variables, even parameters) with a fresh variable α_s, which results in A' (in particular $A'[\overline{\alpha_s} := \bar{n_s}] = A$). Subsets $A^i_p \wedge A^i_c = A^i \subset \overline{\alpha_s \doteq n_s}$ such that $\exists\bar{\alpha}\overline{\alpha_s}.A', \overline{A^i_p, A^i_c}$ is a JCAQPAS answer will be recovered automatically by a residuum-finding process at the end of `ans_typ`. This process is needed regardless of the "dissociation" issue, to uncover the full content of numeric sort constraints.

For efficiency, we use a slightly different approach. On the first iteration of the main algorithm, we dissociate alien subterms, but we do not perform other-sort abduction at all. On the next iteration, we do not perform dissociation, as we expect the dissociation in the partial answers (to predicate variables) from the first step to be sufficient. Other-sort abduction algorithms now have less work, because only a fraction of alien subterm variables α_s remain in the partial answers (see main algorithm in section 5). They also have more information to work with, present in the instatiation of partial answers.

3.4 Simple 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 [1]. 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 [1] 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.

Our algorithm follows a familiar incrementally-generate-and-test scheme:

1. Build a lazy list of possible transformations with linear combinations involving $a \in D$.
    a. For equations a, add combinations $k^s\, a + b$ for $k = -n...n$, $s = -1, 1$ to the stack of transformations to be tried for atoms $b \in C$.
    b. For inequalities a, add combinations $k^s\, a + b$ for $k = 0...n$, $s = -1, 1$ to the stack of trasformations to be tried only for inequalities $b \in C$.
2. Start from $A := C, \text{Acc} := \{\}$. Try atoms $A = a\, A'$ in some order.
3. Let $B = A_i \wedge D \wedge A' \wedge \text{Acc}$.
4. If $B \Rightarrow C$, repeat with $A := A'$.
5. If $B \not\Rightarrow C$, for a transformation a' of a which passes validation against other branches in a joint problem: $\text{Acc} := \text{Acc} \cup \{a'\}$, or fail if all a' fail.
    a. Let a' be a with some transformations applied.
    b. If $A_i \wedge (\text{Acc} \cup \{a'\})$ does not pass `validate` for all a', fail.
    c. Optionally, if $A_i \wedge (\text{Acc} \cup \{a'\})$ passes `validate` for inequality a, add combinations to the stack of transformations as in step (1b).
    d. If $A_i \wedge (\text{Acc} \cup \{a'\})$ passes `validate`, repeat with $A := A', \text{Acc} := \text{Acc} \cup \{a'\}$. (Choice point.)
6. The answers are $A_{i+1} = A_i \wedge \text{Acc}$.

Note that if an equation $a \in \text{Acc}$, linear combinations with it $a + b \wedge \text{Acc}$ would remain equivalent to original $b \wedge \text{Acc}$. For inequalities $a \in \text{Acc}$, combinations $a + b \wedge \text{Acc}$ are weaker than $b \wedge \text{Acc}$, thus the optional step (5c). We precompute the tranformation variants to try out. The parameter n is called `abd_rotations` and defaults to a small value (2 or 3).

To check whether $B \Rightarrow C$, we check for each $c \in C$:

- if $c = x \doteq y$, that $A(x) = A(y)$, where $A(\cdot)$ is the substitution corresponding to equations and implicit equalities in A;
- if $c = x \dot{\leqslant} y$, that $B \wedge y \dot{<} x$ is not satisfiable.

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

Our algorithm finds only fully maximal answers. Unfortunately this means that for many invariant inference problems, some implication branches in initial steps of the main algorithm are insolvable. That is, when the instantiations of the invariants found so far are not informative enough, the expected answer of the JCA problem is not a fully maximal SCA answer to these branches. We mitigate this problem by removing, in the first call to numerical abduction (the second iteration of the main algorithm), branches that contain unary predicate variables in the conclusion.

3.5 Joint constraint abduction for linear arithmetic

We follow the scheme we established in joint constraint abduction for terms.

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 no front branch i has at least n_i answers, the search fails. [FIXME: After an answer working for all branches has been found, we perform additional check, which encapsulates negative constraints introduced by `assert false` construct. If the check fails, we increase the skip count of the head branch and repeat the search. – in term abduction only?]

When a branch has no more solutions to offer – its skip factor n_i has reached the number of fully maximal solutions to that branch – we move it to a separate *runouts* list and continue search starting from a different branch. We do not increase its skip factor, but we check the runouts directly after the first branch, so that conflicting branches can be located. When the first branch conflicts with the runouts, we increase its skip factor and repeat. We keep a count of conflicts for the runouts so that in case of overall failure, we can report a branch likely to be among those preventing abduction.

We remember SCA answers when skipping over them, not to return the same answer for different skip factors. But we also remember all JCA partial answers that led to resetting. If a partial answer becomes as strong as one of them, we can reset without further checking. If an empty partial answer led to resetting, no answer exists. The failed partial answers are initialized with the discarded answers, passed from the main algorithm.

When searching for abduction answer fails, we raise exception `Suspect` that contains the partial answer conjoined with conclusion of an implication that failed to produce an answer compatible with remaining implications.

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 [6], 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 $V = \{x_j, \overline{t_{i,j}} \,|\forall i \exists t_{i,j}. x_j \doteq t_{i,j} \in D_{i,s_{\text{ty}}}\}$.
   b. Let $G = \{\bar{\alpha}_j, u_j, \overline{\theta_{i,j}} \,|\theta_{i,j} = [\bar{\alpha}_j := \bar{g}^i_j], \theta_{i,j}(u_j) = t_{i,j}\}$ be the most specific anti-unifiers of $\overline{t_{i,j}}$ for each j.
   c. Let $D^u_i = \wedge_j \bar{\alpha}_j \doteq \bar{g}^i_j$ and $D^g_i = D_{i,s_{\text{ty}}} \wedge D^u_i$.
   d. Let $D^v_i = \{x \doteq y | x \doteq t_1 \in D^g_i, y \doteq t_2 \in D^g_i, D^g_i \vDash t_1 \doteq t_2\}$.
   e. Let $A_{s_{\text{ty}}} = \wedge_j x_j \doteq u_j \wedge \bigcap_i (D^g_i \wedge D^v_i)$ (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 $\bar{\alpha}_{s_{\text{ty}}} = \overline{\bar{\alpha}_j}$.
   f. Let $\wedge_s D^u_{i,s} \equiv D^u_i$ for $D^u_{i,s}$ of sort s.
3. For sorts $s \neq s_{\text{ty}}$, let $\exists \bar{\alpha}_s. A_s = \text{DisjElim}_s(\overline{D^s_i \wedge D^u_{i,s}})$.
4. The answer is $\exists \overline{\overline{\alpha^j_i} \bar{\alpha}_s} . \wedge_s A_s$.

We simplify the result by substituting-out redundant answer variables.

We follow the anti-unification algorithm provided in [8], fig. 2.

4.1 Extended convex hull

[2] 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 [2] 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.

4.2 Issues in inferring postconditions

Although finding recursive function invariants – predicate variables solved by abduction – could theoretically fail to converge for both the type sort and the numerical sort constraints, neither problem was observed. Finding existential type constraints can only fail to converge for numerical sort, because solutions are expected to decrease in strength. But such diverging numerical constraints are commonplace. The main algorithm starts by performing disjunction elimination only on implication branches corresponding to non-recursive cases, i.e. without binary predicate variables in premise (or unary predicate variables in conclusion). This generates a stronger constraint than the correct one. Subsequent iterations include all branches in disjunction elimination, weakening the constraints, and so still weaker constraints are fed to disjunction elimination in each following step. To ensure convergence of the numerical part, starting from some step of the main loop, we compare consecutive solutions and extrapolate the trend. Currently we simply intersect the sets of atoms, but first we expand equations into pairs of inequalities.

Disjunction elimination limited to non-recursive branches, the initial iteration of postcondition inference, will often generate constraints that contradict other branches. For another iteration to go through, the partial solutions need to be consistent. Therefore we filter the constraints using the same validation mechanism as in abduction. We add atoms to a constraint greedily, but to favor relevant atoms, we do the filtering while computing the connected component of disjunction elimination result. See the details of the main algorithm in section 5.3.

While reading section 5.3, you will notice that postconditions are not subjected to stratification. This is because the type system does not support nested existential types.

5 Solving for Predicate Variables

As we decided to provide the first solution to abduction problems, we similarly simplify the task of solving for predicate variables. Instead of a tree of solutions being refined, we have a single sequence which we unfold until reaching fixpoint or contradiction. Another choice point besides abduction in the original algorithm is the selection of invariants that leaves a consistent subset of atoms as residuum. Here we also select the first solution found. We introduce a form of backtracking, described in section 5.5.

5.1 Invariant Parameter Candidates

We start from the variables β_χ that participate in negative occurrences of predicate variables: $\chi(\beta_\chi)$ or $\chi(\beta_\chi, \alpha)$. We select sets of variables $\beta_\chi \bar{\zeta}^{\,\chi}$, the *parameter candidates*, by considering all atoms of the generated constraints. $\bar{\zeta}^{\,\chi}$ are existential variables that: are connected with β_χ in the hypergraph whose nodes are variables and hyperedges are atoms of the constraints, and are not connected with $\beta_{\chi'}$ for any $\beta_{\chi'}$ that is within scope of β_χ. If $\mathrm{FV}(c) \cap \beta_\chi \bar{\zeta}^{\,\chi} \neq \varnothing$ for an atom c, and β_χ is not in the scope of $\beta_{\chi'}$ for which $\mathrm{FV}(c) \cap \beta_{\chi'} \bar{\zeta}^{\,\chi'} \neq \varnothing$, then c is a candidate for atoms of the solution of χ.

5.2 Solving for Predicates in Negative Positions

The core of our inference algorithm consists of distributing atoms of an abduction answer among the predicate variables. The negative occurrences of predicate variables, when instantiated with the updated solution, enrich the premises so that the next round of abduction leads to a smaller answer (in number of atoms).

Let us discuss the algorithm from [?] for $\mathrm{Split}\big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^{\,\chi}}, \overline{\bar{\zeta}^{\,\chi}}, \overline{\rho^{\chi}}, \overline{A^0_\chi}\big)$. Note that due to existential types predicates, we actually compute $\mathrm{Split}\Big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^{\,\beta_\chi}}, \overline{\bar{\zeta}^{\,\beta_\chi}}, \overline{\rho^{\beta_\chi}}, \overline{A^0_{\beta_\chi}}\Big)$, i.e. we index by β_χ (which can be multiple for a single χ) rather than χ. We retain the notation from [?] here as it better conveys the intent. We do not pass quantifiers around to reflect the source code: the helper function `loop avs ans sol` of function `split` corresponds to $\mathrm{Split}\big(\bar{\alpha}, A, \overline{A^0_{\beta_\chi}}\big)$.

$$
\begin{array}{rrl}
\alpha \prec \beta & \equiv & \alpha <_{\mathcal{Q}} \beta \vee \big(\alpha \leqslant_{\mathcal{Q}} \beta \wedge \beta \not\leqslant_{\mathcal{Q}} \alpha \wedge \alpha \in \overline{\bar{\beta}^{\,\chi}} \wedge \beta \notin \overline{\bar{\beta}^{\,\chi}}\big) \\
A_0 & = & A \setminus \big\{\beta \doteq \alpha \in A \big| \beta \in \overline{\bar{\beta}^{\,\chi}} \wedge (\exists \alpha) \in \mathcal{Q} \wedge \beta \prec \alpha\big\} \\
A^1_\chi & = & \mathrm{Connected}(\bar{\beta}^{\,\chi}, A_0) \\
A^2_\chi & = & \{c \in A^1_\chi | c \text{ is not localized in branch without } \chi \text{ in premise}\} \\
A^3_\chi & = & A^2_\chi \setminus \cup_{\chi' >_{\mathcal{Q}} \chi} A^2_{\chi'} \\
A^4_\chi & = & \mathrm{Connected}(\bar{\beta}^{\,\chi}, A^3_\chi) \\
\text{if} & & \not\models \forall \bar{\alpha} \mathcal{Q}.A \setminus \cup_\chi A^4_\chi \\
\text{then return} & & \bot \\
\text{for all } \overline{A^+_\chi} \text{ min. w.r.t. } \subset \text{ s.t.} & & \wedge_\chi (A^+_\chi \subset A^4_\chi) \wedge \models \forall \bar{\alpha} \mathcal{Q}.A \setminus \cup_\chi A^+_\chi\text{:} \\
\text{if} & & \mathrm{Strat}(A^+_\chi, \bar{\beta}^{\,\chi}) \text{ returns } \bot \text{ for some } \chi \\
\text{then return} & & \bot \\
\text{else } \bar{\alpha}^\chi_+, A^L_\chi, A^R_\chi & = & \mathrm{Strat}(A^+_\chi, \bar{\beta}^{\,\chi}) \\
A_\chi & = & A^0_\chi \cup A^L_\chi \\
\bar{\alpha}^\chi_0 & = & \bar{\alpha} \cap \mathrm{FV}(A_\chi) \\
\bar{\alpha}^{\,\chi} & = & \left(\bar{\alpha}^\chi_0 \setminus \bigcup_{\chi' <_{\mathcal{Q}} \chi} \bar{\alpha}^{\chi'}_0\right) \bar{\alpha}^\chi_+ \\
A_+ & = & \cup_\chi A^R_\chi \\
A_{\mathrm{res}} & = & A_+ \cup \widetilde{A_+}(A \setminus \cup_\chi A^+_\chi) \\
\text{if} & & \cup_\chi \bar{\alpha}^\chi_+ \neq \varnothing \text{ then} \\
\mathcal{Q}', \overline{\bar{\alpha}^{\,\chi}_+}, A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}'^{\chi}.A'_\chi} & \in & \mathrm{Split}\big(\mathcal{Q}\big[\overline{\forall \bar{\beta}^{\,\chi}} := \overline{\forall (\bar{\beta}^{\,\chi} \cup \bar{\alpha}^{\,\chi})}\big], \bar{\alpha} \setminus \cup_\chi \bar{\alpha}^{\,\chi}, A_{\mathrm{res}}, \overline{\bar{\beta}^{\,\chi} \cup \bar{\alpha}^{\,\chi}}, \overline{A_\chi}\big) \\
\text{return} & & \mathcal{Q}', \overline{\bar{\alpha}^\chi_+ \bar{\alpha}^{\,\chi}_+}, A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\,\chi} \bar{\alpha}'^{\chi}.A'_\chi} \\
\text{else return} & & \forall (\bar{\alpha} \setminus \cup_\chi \bar{\alpha}^{\,\chi}) \mathcal{Q}, \overline{\bar{\alpha}^{\,\chi}_+}, A_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\,\chi}.A_\chi}
\end{array}
$$

where $\mathrm{Strat}(A, \bar{\beta}^{\,\chi})$ is computed as follows: for every $c \in A$, and for every $\beta_2 \in \mathrm{FV}(c)$ such that $\beta_1 <_{\mathcal{Q}} \beta_2$ for $\beta_1 \in \bar{\beta}^{\,\chi}$, if β_2 is universally quantified in $\mathcal{Q}$, then return $\bot$; otherwise, introduce a fresh variable α_f, replace $c := c[\beta_2 := \alpha_f]$, add $\beta_2 \doteq \alpha_f$ to A^R_χ and α_f to $\bar{\alpha}^\chi_+$, after replacing all such β_2 add the resulting c to A^L_χ.

As an additional measure to detect misleading splits early, we pass an approximate information about which branch D_i, C_i of the JCA problem atoms $a \in A$ originate from: $\rho(a)$, and which branches contain χ in premise: ρ^χ, so that an atom a is put into solution A^+_χ only if D_i contains χ.

Description of the algorithm in more detail:

1. $\alpha \prec \beta \equiv \alpha <_{\mathcal{Q}} \beta \vee \left(\alpha \leqslant_{\mathcal{Q}} \beta \wedge \beta \not<_{\mathcal{Q}} \alpha \wedge \alpha \in \overline{\bar{\beta}^{\chi}} \wedge \beta \notin \overline{\bar{\beta}^{\chi}}\right)$ The variables $\overline{\bar{\beta}^{\chi}}$ are the answer variables of the solution from the previous round. We need to keep them apart from other variables even when they're not separated by quantifier alternation.
2. $A_0 = A \setminus \left\{ \beta \doteq \alpha \in A \middle| \beta \in \overline{\bar{\beta}^{\chi}} \wedge (\exists \alpha) \in \mathcal{Q} \wedge \beta \prec \alpha \right\}$ We handle information carried in $\beta \doteq \alpha$ by substituting α with β.
3. $A^1_\chi = \text{Connected}(\bar{\beta}^{\chi}, A_0)$ Initial filtering of candidates to have less work at later stages. Connected$(\bar{\beta}, A_0)$ is the subset of atoms of A_0 reachable from nodes $\bar{\beta}$, where atoms are considered directly connected when they share a variable.
4. $A^2_\chi = \{c \in A^1_\chi | c \text{ is not localized in branch without } \chi \text{ in premise}\}$ Guard against misplacing abduction answer atoms. Unfortunately localization information can be too general for this step alone to be sufficient at splitting the answer.
5. $A^3_\chi = A^2_\chi \setminus \cup_{\chi' >_{\mathcal{Q}} \chi} A^2_{\chi'}$ If a premise has an χ' atom then it has an χ atom for $\beta^{\chi'}$ downstream of b^{χ}.
6. $A^4_\chi = \text{Connected}(\bar{\beta}^{\chi}, A^3_\chi)$ Disconnected atoms do not contribute to the invariant over the parameters $\bar{\beta}^{\chi}$. Note that the solution atoms are not yet separated from the residuum atoms so the final solution might still have disconnected components.
7. if $\not\models \forall \bar{\alpha} \mathcal{Q}. A \setminus \cup_\chi A^4_\chi$ then return $\bot$ Failed solution attempt. A common example is when the use site of recursive definition, resp. the existential type introduction site, is not in scope of a defining site of recursive definition, resp. an existential type elimination site, and has too strong requirements.
8. for all $\overline{A^+_\chi}$ minimal w.r.t. $\subset$ such that $\wedge_\chi (A^+_\chi \subset A^4_\chi) \wedge \models \mathcal{Q}. A \setminus \cup_\chi A^+_\chi$: Select invariants such that the residuum $A \setminus \cup_\chi A^+_\chi$ is consistent. The final residuum A_{res} represents the global constraints, the solution for global type variables. The solutions A^+_χ represent the invariants, the solution for invariant type parameters.
9. if $\text{Strat}(A^+_\chi, \bar{\beta}^{\chi})$ returns $\bot$ for some χ then return $\bot$ In the implementation, we address stratification issues already during abduction.
10. $\bar{\alpha}^{\chi}_+$, A^L_χ, $A^R_\chi = \text{Strat}(A^+_\chi, \bar{\beta}^{\chi})$ is computed as follows: for every $c \in A^+_\chi$, and for every $\beta_2 \in \text{FV}(c)$ such that $\beta_1 <_{\mathcal{Q}} \beta_2$ for $\beta_1 \in \bar{\beta}^{\chi}$, if β_2 is universally quantified in $\mathcal{Q}$, then return $\bot$; otherwise, introduce a fresh variable α_f, replace $c := c[\beta_2 := \alpha_f]$, add $\beta_2 \doteq \alpha_f$ to A^R_χ and α_f to $\bar{\alpha}^{\chi}_+$, after replacing all such β_2 add the resulting c to A^L_χ.
    - We add $\bar{\alpha}^{\chi}_+$ to $\bar{\beta}^{\chi}$.
11. $A_\chi = A^0_\chi \cup A^L_\chi$ is the updated solution formula for χ, where A^0_χ is the solution from previous round.
12. $\bar{\alpha}^{\chi}_0 = \bar{\alpha} \cap \text{FV}(A_\chi)$ are the additional solution parameters coming from variables generated by abduction.
13. $\bar{\alpha}^{\chi} = \left(\bar{\alpha}^{\chi}_0 \setminus \bigcup_{\chi' <_{\mathcal{Q}} \chi} \bar{\alpha}^{\chi'}_0\right) \bar{\alpha}^{\chi}_+$ The final solution parameters also include the variables generated by Strat.
14. $A_+ = \cup_\chi A^R_\chi$ and $A_{\text{res}} = A_+ \cup \widetilde{A_+}(A \setminus \cup_\chi A^+_\chi)$ is the resulting global constraint, where $\widetilde{A_+}$ is the substitution corresponding to A_+.
15. if $\cup_\chi \bar{\alpha}^{\chi}_+ \neq \varnothing$ then – If Strat generated new variables, we need to redistribute the $\widetilde{A_+}(A \setminus \cup_\chi A^+_\chi)$ atoms to make $\mathcal{Q}'. A_{\text{res}}$ valid again.
16. $\mathcal{Q}', \overline{\bar{\alpha}^{\chi\prime}_+}, A'_{\text{res}}, \overline{\exists \bar{\alpha}'^{\chi}. A'_\chi} \in \text{Split}\left(\mathcal{Q}\left[\overline{\forall \bar{\beta}^{\chi}} := \overline{\forall (\bar{\beta}^{\chi} \cup \bar{\alpha}^{\chi})}\right], \bar{\alpha} \setminus \cup_\chi \bar{\alpha}^{\chi}, A_{\text{res}}, \overline{\bar{\beta}^{\chi} \cup \bar{\alpha}^{\chi}}, \overline{A_\chi}\right)$ Recursive call includes $\bar{\alpha}^{\chi}_+$ in $\bar{\beta}^{\chi}$ so that, among other things, A_+ are redistributed into A_χ.

17. return $\mathcal{Q}', \overline{\bar{\alpha}_+^{\chi} \bar{\alpha}_+^{\chi\prime}}, A'_{\text{res}}, \overline{\exists \bar{\alpha}^{\chi} \bar{\alpha}^{\prime\chi}.A'_{\chi}}$ We do not add $\forall \bar{\alpha}$ in front of $\mathcal{Q}'$ because it already includes these variables. $\overline{\bar{\alpha}_+^{\chi} \bar{\alpha}_+^{\chi\prime}}$ lists all variables introduced by Strat.

18. else return $\forall(\bar{\alpha} \setminus \cup_{\chi} \bar{\alpha}^{\chi}) \mathcal{Q}, \overline{\bar{\alpha}_+^{\chi}}, A_{\text{res}}, \overline{\exists \bar{\alpha}^{\chi}.A_{\chi}}$ Note that $\bar{\alpha} \setminus \cup_{\chi} \bar{\alpha}^{\chi}$ does not contain the current $\bar{\beta}^{\chi}$, because $\bar{\alpha}$ does not contain it initially and the recursive call maintains that: $\bar{\alpha} := \bar{\alpha} \setminus \cup_{\chi} \bar{\alpha}^{\chi}, \bar{\beta}^{\chi} := \bar{\beta}^{\chi} \bar{\alpha}^{\chi}$.

Finally we define $\text{Split}(\bar{\alpha}, A) := \text{Split}(\bar{\alpha}, A, \bar{\top})$. The complete algorithm for solving predicate variables is presented in the next section.

5.3 Solving for Existential Types Predicates and Main Algorithm

The general scheme is that we perform disjunction elimination on branches with positive occurrences of existential type predicate variables on each round. Since the branches are substituted with the solution from previous round, disjunction elimination will automatically preserve monotonicity. We retain existential types predicate variables in the later abduction step.

What differentiates existential type predicate variables from recursive definition predicate variables is that the former are not local to context in our implementation, we ignore the `CstrIntro` rule, while the latter are treated as local to context in the implementation. It means that free variables need to be bound in the former while they are retained in the latter.

In the algorithm we operate on all predicate variables, and perform additional operations on existential type predicate variables; there are no operations pertaining only to recursive definition predicate variables. The equation numbers (N) below are matched by comment numbers `(* N *)` in the source code. Step k of the loop of the final algorithm:

$$
\begin{aligned}
\overline{\exists \bar{\beta}^{\chi,k}.F_{\chi}} &= S_k \\
\text{Prune}(A) &= A \backslash \overline{\alpha_{\alpha}^{K} \doteq \ldots} && (1) \\
&\quad \text{TODO: but it's not enough...} \\
S'_k = \overline{\exists \bar{\beta}^{\chi,k}.F'_{\chi}} &= \overline{\exists \bar{\beta}^{\chi,k}.\text{Prune}(F_{\chi})} \\
D_K^{\alpha} \Rightarrow C_K^{\alpha} \in R_k^{-} S'_k(\Phi) &= \text{all such that } \chi_K(\alpha, \alpha_{\alpha}^{K}) \in C_K^{\alpha}, && (2) \\
&\quad \overline{C_j^{\alpha}} = \{C \mid D \Rightarrow C \in S_k(\Phi) \wedge D \subseteq D_K^{\alpha}\} \\
\exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K} &= \text{Connected}\Big(\delta, \text{DisjElim}\Big(\overline{\delta \doteq \alpha \wedge D_K^{\alpha} \wedge_j C_j^{\alpha}}_{\alpha \in \overline{\alpha_3^{i,K}}}\Big)\Big) && (3) \\
\exists \bar{\alpha}^{\chi_K}.G'_{\chi_K} &= \text{Simpl}(\text{FV}(G_{\chi_K}).G_{\chi_K}) && (4) \\
\Xi(\exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K}) &= \exists \bar{\alpha}^{\chi_K}.\delta' \doteq \overrightarrow{\overline{\text{FV}(G_{\chi_K}) \backslash \bar{\alpha}_g^{\chi_K}}} \wedge G'_{\chi_K} \\
R_g(\chi_K) = \exists \bar{\alpha}^{\chi_K}.F_{\chi_K} &= \Xi(\exists \bar{\alpha}_g^{\chi_K}.H(R_k(\chi_K), G_{\chi_K})) && (5) \\
P_g(\chi_K) &= \delta' \doteq \overrightarrow{\overline{\text{FV}(G_{\chi_K}) \backslash \bar{\alpha}_g^{\chi_K}}} \\
\mathcal{Q}'. \wedge_i (D_i \Rightarrow C_i) &= R_g^{-} P_g^{+} S_k(\Phi) \\
&\quad \text{At later iterations, check negative constraints.} && (6) \\
\exists \bar{\alpha}.A &= \text{Abd}\big(\mathcal{Q}' \backslash \overline{\forall \beta_{\chi} \bar{\beta}^{\chi}}, \overline{\beta_{\chi} \bar{\beta}^{\chi}}, \overline{\zeta^{\chi}}, \overline{D_i, C_i}\big) && (7) \\
\big(\mathcal{Q}^{k+1}, \overline{\bar{\alpha}_+^{\chi}}, A_{\text{res}}, \overline{\exists \bar{\alpha}^{\beta_{\chi}}.A_{\beta_{\chi}}}\big) &= \text{Split}\big(\mathcal{Q}', \bar{\alpha}, A, \overline{\beta_{\chi} \bar{\beta}^{\chi}}, \overline{\zeta^{\chi}}\big) \\
R_{k+1}(\chi_K) &= \exists \bar{\beta}^{\chi_K,k}.\text{Simpl}\Big(\overline{\exists \bar{\alpha}^{\chi_K} \bar{\alpha}^{\beta_{\chi_K}}}.F_{\chi_K} \\
&\qquad \wedge_{\chi_K} A_{\beta_{\chi_K}}\Big[\overline{\beta_{\chi_K} \bar{\beta}^{\beta_{\chi_K}}} := \overline{\delta \bar{\beta}^{\chi_K,k}}\Big]\Big) && (8) \\
S_{k+1}(\chi) &= \exists \bar{\beta}^{\chi,k}.\text{Simpl}\big(\overline{\exists \bar{\alpha}^{\beta_{\chi}}}.F'_{\chi} \\
&\qquad \wedge A_{\beta_{\chi}}\big[\overline{\beta_{\chi} \bar{\beta}^{\chi}} := \overline{\delta \bar{\beta}^{\chi,k}}\big]\big) && (9) \\
\text{if} &\quad (\forall \chi) S_{k+1}(\chi) \subseteq S_k(\chi), && (10) \\
&\quad (\forall \chi_K) R_{k+1}(\chi_K) = R_k(\chi_K), \\
&\quad k > 1 \\
\text{then return} &\quad A_{\text{res}}, S_{k+1}, R_{k+1} \\
\text{repeat} &\quad k := k + 1 && (11)
\end{aligned}
$$

Note that Split returns $\overline{\exists\bar{\alpha}^{\,\beta_\chi}.A_{\beta_\chi}}$ rather than $\overline{\exists\bar{\alpha}^{\,\chi}.A_\chi}$. This is because in case of existential type predicate variables χ_K, there can be multiple negative position occurrences $\chi_K(\beta_{\chi_K},\cdot)$ with different β_{χ_K} when the corresponding value is used in multiple **let** ... **in** expressions. The variant of the algorithm to achieve completeness as conjectured in [?] would compute all answers for variants of Abd and Split algorithms that return multiple answers. Unary predicate variables $\chi(\beta_\chi)$ can also have multiple negative occurrences in the normalized form, but always with the same argument β_χ. The substitution $\left[\,\overline{\beta_{\chi_K}\bar{\beta}^{\,\beta_{\chi_K}}} := \overline{\delta\bar{\beta}^{\,\chi_K,k}}\,\right]$ replaces the instance parameters introduced in $\chi_K(\beta_{\chi_K},\cdot)$ by the formal parameters used in $R_k(\chi_K)$. The renamings from instance parameters to formal parameters are stored as `q.b_renaming`.

Even for a fixed K, the same branch $D_K^\alpha \Rightarrow C_K^\alpha$ can contribute to multiple disjuncts, with different $\alpha = \alpha_3^{i,K}$. Substitution R_g^- substitutes only negative occurrences of χ_K, i.e. it affects only the premises. Substitution P_g^+ substitutes only positive occurrences of χ_K, i.e. it affects only the conclusions. P_g^+ ensures that the parameters of the postcondition are instantiated as the variables from which they were derived. At position 1 we remove the parameter constraints from solutions, since they will be reintroduced by P_g with values adjusted to the postconditions derived in the next step.

We start with $S_0 := \bar{\top}$ and $R_0 := \bar{\top}$. S_k grow in strength by definition. The disjunction elimination parts G_{χ_K} of R_1 and R_2 are computed from non-recusrive branches only. Starting from R_2, R_k are expected to decrease in strength, but monotonicity is not guaranteed because of contributions from abduction: mostly in form of $A_{\beta_{\chi_K}}$, but also from stronger premises due to S_k.

Connected(α, G) is the connected component of hypergraph G containing node α, where nodes are variables FV(G) and hyperedges are atoms $c \in G$. In initial iterations, when the branches $D_K^\alpha \Rightarrow C_K^\alpha$ are selected from non-recursive branches only, we include a connected atom only if it is satisfiable in all branches. $H(R_k, R_{k+1})$ is a convergence improving heuristic, with $H(R_k, R_{k+1}) = R_{k+1}$ for early iterations and "roughly" $H(R_k, R_{k+1}) = R_k \cap R_{k+1}$ later.

We introduced the `assert false` construct into the programming language to indicate that a branch of code should not be reached. Type inference generates for it the logical connective $\boldsymbol{F}$ (falsehood). We partition the implication branches D_i, C_i into $\{D_i, C_i | \boldsymbol{F} \notin C_i\}$ which are fed to the algorithm and $\Phi_{\boldsymbol{F}} = \{(D_i, C_i) | \boldsymbol{F} \in C_i\}$. After the main algorithm ends we check that for each $(D_i, C_i) \in \Phi_{\boldsymbol{F}}$, $S_k(D_i)$ fails. Optionally, but by default, we perform the check in each iteration. Turning this option *on* gives a limited way to express negative constraints. With the option *off*, the inferred type is the same as it would be without the impossible pattern matching branch in the program, but the check statically guarantees that the branch is in fact impossible. The option should be turned *off* when a single iteration (plus fallback backtracking described below) is insufficient to solve for the invariants.

We implement backtracking using a tabu-search-like discard list. When abduction raises an exception: for example contradiction arises in the branches $S_k(\Phi)$ passed to it, or it cannot find an answer and raises `Suspect` with information on potential problem, we fall-back to step $k-1$. Similarly, with checking for negative constraints *on*, when the check of negative branches $(D_i, C_i) \in \Phi_{\boldsymbol{F}}$, $\not\models \exists \text{FV}(S_k(D_i)).S_k(D_i)$ fails. In step $k-1$, we maintain a discard list of different answers found in this step: initially empty, after fall-back we add there the latest answer. We redo step $k-1$ starting from $S_{k-1}(\Phi)$. Infinite loop is avoided because answers already attempted are discarded. When step $k-1$ cannot find a new answer, we fall back to step $k-2$, etc. We store discard lists for distinct sorts separately and we only add to the discard list of the sort that caused fallback. Unfortunately this means a slight loss of completeness, as an error in another sort might be due to bad choice in the sort of types. The loss is "slight" because of the dissociation step described previously. Moreover, the sort information from checking negative branches is likewise only approximate.

5.4 Stages of iteration

Changes in the algorithm between iterations were mentioned above but not clearly exposed. Invariant inference and postcondition inference go through similar stages. Invariants, solved by abduction:

1. $k = 0$ Only term abduction – invariants of type shapes – is performed, for all branches.

2. $k=1$ Both term abduction and numerical abduction are performed, but numerical abduction only for non-recursive branches.
3. $k=2$ Abduction is performed on all branches – type and numerical invariants are found.

In a single iteration, disjunction elimination precedes abduction.

1. $k_0 \leqslant k < k_1$ Term disjunction elimination – invariants of type shapes – is performed, only for non-recursive branches.
2. $k_1 \leqslant k < k_2$ Both term and numerical disjunction elimination are performed, but only for non-recursive branches.
3. $k_2 \leqslant k < k_3$ Disjunction elimination is performed on all branches – type and numerical postconditions are found.
4. $k_3 \leqslant k$ Additionally, we enforce convergence by intersecting the result with the previous-iteration result.

Our current choice of parameters is $[k_0; k_1; k_2; k_3] = $ `disj_step = [|0;1;2;4|]`.

When existential types are used, the expected number of iterations is $k = 5$ (six iterations), because the last iteration needs to verify that the last-but-one iteration has found the correct answer. The minimal number of iterations is $k = 2$ (three iterations), so that all branches are considered.

5.5 Implementation details

We represent $\vec{\alpha}$ as a tuple type rather than as a function type. We modify the quantifier $\mathcal{Q}$ imperatively, because it mostly grows monotonically, and when variables are dropped they do not conflict with fresh variables generated later.

The code that selects $\wedge_\chi(A^+_\chi \subset A^{\text{cap}}_\chi) \wedge \vDash \mathcal{Q}.A \setminus \cup_\chi A^+_\chi$ is an incremental validity checker. It starts with $A \setminus \cup_\chi A^{\text{cap}}_\chi$ and tries to add as many atoms $c \in \cup_\chi A^{\text{cap}}_\chi$ as possible to what in effect becomes A_{res}. The remaining atoms are distributed among $A^+_{\beta_\chi}$ by putting them into the last β_χ in $\mathcal{Q}$, i.e. the first `b` in the `q.negbs` list, for which $x \prec \big(\overline{\beta^{\,\chi}\zeta^{\,\chi}} \cap \text{FV}(c)\big) \cap \beta^{\,\chi}\zeta^{\,\chi} \neq \varnothing$.

We count the number of iterations of the main loop, a fallback decreases the iteration number to the previous value. The main loop decides whether multisort abduction should dissociate alien subterms – in the first iteration of the loop – or should perform abductions for other sorts – in subsequent iteration. See discussion in subsection 3.3. In the first two iterations, we remove branches that contain unary predicate variables in the conclusion (or binary predicate variables in the premise, keeping only `nonrec` branches), as discussed at the end of subsection 3.4 and beginning of subsection 4.2. As discussed in subsection 4.2, starting from the fourth iteration ($k = 3$), we enforce convergence on solutions for binary predicate variables.

Computing abduction is the "axis" of the main loop. If anything fails, the previous abduction answer is the culprit. We add the previous answer to the discard list and retry, without incrementing the iteration number. If abduction and splitting succeeds, we reset the discard list and increment the iteration number. We use recursion for backtracking, instead of making `loop` tail-recursive.

Bibliography

[1] Sergey Berezin, Vijay Ganesh and David L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 521–536. Berlin, Heidelberg, 2003. Springer-Verlag.

[2] Komei Fukuda, Thomas M. Liebling and Christine Lütolf. Extended convex hull. In *Proceedings of the 12th Canadian Conference on Computational Geometry, Fredericton, New Brunswick, Canada, August 16-19, 2000*. 2000.

[3] Michael Maher and Ge Huang. On computing constraint abduction answers. In Iliano Cervesato, Helmut Veith and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 421–435. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89439-1_30.

[4] Łukasz Stafiniak. Joint constraint abduction problems. 2011. The International Workshop on Unification.

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

[6] Łukasz Stafiniak. Constraint disjunction elimination problems. Manuscript, 2013. Available at: http://www.ii.uni.wroc.pl/~lukstafi/pubs/disjelim.pdf

[7] Łukasz Stafiniak. Joint constraint abduction problems. Manuscript, 2013. Available at: http://www.ii.uni.wroc.pl/~lukstafi/pubs/abduction-revised.pdf

[8] B Østvold. A functional reconstruction of anti-unification. Technical Report, Norwegian Computing Center, Oslo, Norway, 2004.