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

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: `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 $\lambda(\ldots)\,e$. Constructor: `Lam`.

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

`ClauseWhen`. p **when** $\wedge_i m_i \leqslant n_i.e$: Branch of pattern matching with a condition. Concrete syntax: e.g. `p when a <= b && c <= d -> 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: `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`.

`AssertLeq`. **assert num** $m \leqslant n$; e: add the inequality to the constraints. Concrete syntax: `assert num m <= n; ...` Constructor: `AssertLeq`. There is a similar construct for adding an equation over types of expressions – concrete syntax: `assert type m = n; ...` Constructor: `AssertEqty`.

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. The syntax of types and formulas:

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 var. with coef.	$\frac{1}{3}n$	`1/3 n`	`Alien(Lin(1,3,VNam(Num_sort,"n")))`	
type constructor	List	`List`		`TCons(CNamed...)`
number (type)	7	`7`		`Alien (Cst...)`
numeral (expr.)	7	`7`		`Num`
numerical sum (type)	$m+n$	`m+n`		`Alien (Add...)`
existential type	$\exists k, n[k \leqslant n].\tau$	`ex k n [k<=n].t`	$\exists$`k,n[k`$\leq$`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` $\to$ `t2`	`Fun`
equation	$a \doteq b$	`a = b`		`Eqty`
inequation	$k \leqslant n$	`k <= n`	`k` $\leq$ `n`	`A(Num_atom(Leq...))`
conjunction	$\varphi_1 \wedge \varphi_2$	`a=b && b=a`	`a=b` $\wedge$ `b=a`	built-in lists

For the syntax of expressions, we discourage non-ASCII symbols. Restating the first paragraph, below e, e_i stand for any expression, p, p_i stand for any pattern, x stands for any lower-case identifier and K for an upper-case identifier.

named value	x	`x` –lower-case identifier	`Var`
numeral (expr.)	7	`7`	`Num`
constructor	K	`K` –upper-case identifier	`Cons`
application	$e_1\, e_2$	`e1 e2`	`App`
non-br. function	$\lambda(p_1.\lambda(p_2.e))$	`fun (p1,p2) p3 -> e`	`Lam(...)`
branching function	$\lambda(p_1.e_1 ... p_n.e_n)$	`function p1->e1 | ... | pn->en`	`Lam(...)`
cond. branch	p **when** $\wedge_{i \in \mathbf{2}} m_i \leqslant n_i.e$	`p when m1<=n1 && m2<=n2 -> e`	`Lam(...)`
pattern match	$\lambda(p_1.e_1 ... p_n.e_n)\, e$	`match e with p1->e1 | ...`	`App(Lam...,e)`
if-then-else clause	$\lambda(K_T.e_1, K_F.e_2)\, e$	`if e then e1 else e2`	`App(Lam...,e)`
postcond. function	$\lambda[K](p_1.e_1 ... p_n.e_n)$	`efunction p1->e1 | ...`	`ExLam`
postcond. match	$\lambda[K](p_1.e_1 ... p_n.e_n)\, e$	`ematch e with p1->e1 | ...`	`App(ExLam...)`
eif-then-else clause	$\lambda[K](K_T.e_1, K_F.e_2)\, e$	`eif e then e1 else e2`	`App(ExLam...)`
rec. definition	**letrec** $x = e_1$ **in** e_2	`let rec x = e1 in e2`	`Letrec`
definition	**let** $p = e_1$ **in** e_2	`let p1,p2 = e1 in e2`	`Letin`
asserting dead br.	$\boldsymbol{F}$	`assert false`	`AssertFalse`
assert equal types	**assert type** $e_1 \doteq e_2; e_3$	`assert type e1 = e2; e3`	`AssertEqty`
assert inequality	**assert num** $e_1 \leqslant e_2; e_3$	`assert num e1 <= e2; e3`	`AssertLeq`

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

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

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	`datatype List : type * num`	`TypConstr`
value constructor	`datacons Cons : all n a. a * List(a,n)-->List(a,n+1)`	`ValConstr`
	`datacons 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`
let-declaration	`external let mult : Int→Int→Int = "( * )"`	`PrimVal`
rec. definition	`let rec f =...`	`LetRecVal`
non-rec. definition	`let p1,p2 =...`	`LetVal`
definition with test	`let rec f =... test e1; ...; en`	`LetRecVal`

Tests list expressions of type `Boolean` that at runtime have to evaluate to `True`. Type inference is affected by the constraints generated to typecheck the expressions.

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.

At any place between lexemes, regular comments encapsulated in `(*...*)` can occur. They are ignored during lexing. In front of all toplevel definitions and declarations, e.g. before a `datatype`, `datacons`, `external`, `let rec` or `let`, and in front of `let rec .. in` and `let .. in` nodes in expressions, documentation comments `(**...*)` can be put. Documentation comments at other places are syntax errors. Documentation comments are preserved both in generated interface files and in exported source code files. Documentation comments are the first parameters of corresponding constructors, and are of type `string option`.

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 ?, 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`. The constraints generated by `constr_gen_letrec`, resp. `constr_gen_let`, are analogous to those generated for `Letrec`, resp. for `Letin` or a `Lam` clause. If the LHS pattern of `LetVal` is just a name, e.g. `let x = ...`, then we call `constr_gen_letrec`, telling it not to bind the name `x`, and do the processing in the same way as for `Letrec`.

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, for `LetVal` – unless it just binds a value as discussed above – we 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 num 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

We reduce the constraint to alternation-minimizing prenex-normal form, as in the formalization. We explored the option to preserve the scope relations, but it was less suited for determining quantifier violations. We return a `var_scope`-producing variable comparison function. The branches we return from normalization have unified conclusions, since we need to unify 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 the disjunct. 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, unless one of them is non-recursive and the other is recursive. We call an implication branch recursive when an unary predicate variable χ (not a χ_K) appears in the conclusion *or* a binary predicate variable χ_K appears in the premise.

3 Abduction

Our formal specification of abduction 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 [4] 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 $\boldsymbol{U}(D \wedge C)$ and iteratively replaces subterms by fresh variables $\alpha \in \bar{\alpha}$ for a final solution $\exists \bar{\alpha} . A$. As our primary approach to mitigate some of the limitations of fully maximal answers, we start from $\boldsymbol{U}\big(\tilde{A}(D \wedge C)\big)$, where $\exists \bar{\alpha} . A$ is the solution to previous problems solved by the joint abduction algorithm, and $\tilde{A}(\cdot)$ is the corresponding substitution. Moreover, motivated by examples from Chuan-kai Lin [3], we intruduce variable-variable equations $\beta_1 \doteq \beta_2$, for $\beta_1 \beta_2 \subset \bar{\beta}$, not implied by $\tilde{A}(D \wedge C)$, as additional candidate answer atoms. During abduction $\mathrm{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i})$, we ensure that the (partial as well as final) answer $\exists \bar{\alpha} . A$ satisfies $\vDash \mathcal{Q} . A[\bar{\alpha} \bar{\beta} := \bar{t}\,]$ for some $\bar{t}$. We achieve this by normalizing the answer using parameterized unification under quantifiers $\boldsymbol{U}_{\bar{\alpha}\bar{\beta}}(\mathcal{Q}.A)$. $\bar{\beta}$ are the parameters of the invariants.

In fact, when performing unification, we check more than $\boldsymbol{U}_{\bar{\alpha}\bar{\beta}}(\mathcal{Q}.A)$ requires. We also ensure that the use of parameters will not cause problems in the `split` phase of the main algorithm. To this effect, we forbid substitution of a variable β_1 from $\bar{\beta}$ with a term containing a universally quantified variable that is not in $\bar{\beta}$ and to the right of β_1 in $\mathcal{Q}$. Also, we forbid substitution of a variable β_1 from $\bar{\beta}^{\,\chi}$ with a term containing a variable $\beta_2 \in \bar{\beta}^{\,\chi'}$ for $\chi \neq \chi'$.

In implementing [4] p. 13, we follow a top-down approach where bigger subterms are abstracted first – replaced by a 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. 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 there could be an infinite number of abduction answers, there is always a finite number of *fully maximal* answers, or more generally, a finite number of equivalence classes of formulas strictly stronger than a given conjunction of equations in the domain $T(F)$. 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 parameters and candidates for parameters of the invariants, $\bar{\beta}$, updated as we add new atoms to the partial answer; existential variables that are not to the left of parameters and are connected to parameters become parameters; we process atoms containing parameters first;
- 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.

Since an atom can be mistakenly discarded when some variable could be considered an invariant parameter but is not at the time, we process atoms incident with candidates for invariant parameters first. A variable becomes a candidate for a parameter if there is a parameter that depends on it. That is, we process atoms $x \doteq t$ such that $x \in \bar{\beta}$ first, and if equation $x \doteq t$ is added to the partial solution, we add to $\bar{\beta}$ existential variables in t. Note that $x \doteq t$ can stand for either $x := t$, or $y := x$ for $t = y$.

To simplify the search in presence of a quantifier prefix, we preprocess the initial candidate by eliminating universally quantified variables:

$$\begin{aligned} S &= \left[\, t_u^{-} := t_u^{\bar{\prime}} \,\right] \text{ for } \mathrm{FV}(t_u) \cap \overline{\beta_u} \neq \varnothing, \overline{\forall \beta_u} \subset \mathcal{Q} \text{ such that } \mathcal{M} \vDash D \Rightarrow \dot{S}\,, \\ \mathrm{Rev}_{\forall}(\mathcal{Q}, \bar{\beta}\,, D, C) &= \{c' | c \in C\,, \text{if } \mathcal{M} \vDash \mathcal{Q}.c[\bar{\beta} := \bar{t}\,] \text{ for some } \bar{t} \text{ then } c' = c \text{ else } c' = S(c)\} \end{aligned}$$

Note that S above is a substitution of subterms rather than of variables. To move further beyond fully maximal answers, we incorporate candidates $\beta_1 \doteq \beta_2$ for which the following conditions hold: $\beta_1\beta_2 \subset \bar{\beta}\,$, $\beta_1 := t_1 \in \boldsymbol{U}\big(\tilde{A}(D \wedge C)\big)$, $\beta_2 := t_2 \in \boldsymbol{U}\big(\tilde{A}(D \wedge C)\big)$ and $t_1 \doteq t_2$ is satisfiable. We also need to include the unifier of $t_1 \doteq t_2$ among the candidates, since otherwise the equation $\beta_1 \doteq \beta_2$ would not suffice to imply that of the atoms $\beta_1 \doteq t_1$, $\beta_2 \doteq t_2$ which belongs to the conclusion C. The *full candidates* $\boldsymbol{U}\big(\tilde{A}(D \wedge C)\big)$ and the *guess candidates* $\overline{\beta_1 := \beta_2; \boldsymbol{U}(t_1 \doteq t_2)}$ are kept apart, the guess candidates are guessed before the full candidates. By default, we additionally limit consideration to atoms $\beta_1 \doteq t_1$, $\beta_2 \doteq t_2$ where t_1, t_2 are not themselves variables.

To recapitulate, the implementation is:

- `abstract` is the entry point.
- If `guess_cand=[]` and `full_cand=[]` – no more candidates to add to the partial solution: check for repeated answers, skipping, and discarded answers.
- If `guess_cand=[]`, pick the next `full_cand` atom $\mathrm{FV}(c) \cap \bar{\beta} \neq \varnothing$ if any, reordering the candidates until one is found. Otherwise, pick the first `guess_cand` atom without reordering.
- `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.
    6. Keep a variant of the original atom, but with constants substituted-out by variable-constant equations from the premise. Redundant, and optional: only when `revert_cst` is true and `more_general` is false.
- Choices 2-6 are guarded by `try-with`, as tests raise `Contradiction` if they fail, choice 1 only tests `implies_concl` which returns a boolean.
- Default ordering of choices is 1, 6, 2, 4, 3, 5 – pushing 4 up minimizes the amount of branching in 5.
    - There is an option `more_general`, which reorders the choices to: 1, 6, 4, 2, 3, 5; however the option is not exposed in the interface because the cost of this reordering is prohibitive.
    - An option `richer_answers` reorders the choices to: 6, 1, 2, 4, 3, 5; it does not increase computational cost but sometimes leads to answers that are not most general.
    - If choice 6 would lead to more negative constraints contradicted than choice 1, we pick choice 6 first for a particular candidate atom.

- An option `prefer_guess` reorders choice 6 prior to choice 1, but only for guess candidates, i.e. `guess_cand`.

- Form initial candidates $\text{Rev}_\forall(\mathcal{Q}, \bar{\beta}, \boldsymbol{U}(D \wedge A_p), \boldsymbol{U}(A_p \wedge D \wedge C))$.
- Form the substitution of subterms for choice-6 counterparts of initial candidate atoms. For $\alpha_1 \doteq \tau, ..., \alpha_n \doteq \tau \in \boldsymbol{U}(D \wedge A_p)$, form the substitution of subterms $\alpha_1 := \alpha_i, ..., \alpha_n := \alpha_i, \tau := \alpha_i$ (excluding $\alpha_i := \alpha_i$) where α_i is the most upstream existential variable (or parameter) and τ is a constant.
    - Since for efficiency reasons we do not always remove alien subterms, we need to mitigate the problem of alien subterm variables causing violation of the quantifier prefix. To this effect, we include the premise equations from other sorts in the process generating the initial candidates and choice 6 candidates, but not as candidates. Not to lose generality of answer, we only keep a renaming substitution, in particular we try to eliminate universal variables.
- Sort the initial candidates by decreasing size, because shorter answer atoms are more valuable and dropping a candidate from participating in an answer is the first choice.
    - There is an argument in favor of sorting by increasing size: so that the replacements of step 2 are formed at a root position before they are used in step 5 – instead of forming a replacement at a subterm, and using it in step 5 at a root.
    - If ordering in increasing size turns out to be necessary, a workaround should be introduced to favor answers that, if possible, do not have parameters $\bar{\beta}$ as left-hand-sides.

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.

We perform a two-layer iterative deepening when `more_general` is false: in the first run we only try choices 1 and 6. It is an imperfect optimization since the running time gets longer whenever choices 2-5 are needed.

3.1.1 Heuristic for better answers to invariants

We implement an optional heuristic in forming the candidates proposed by choice 6. It may lead to better invariants when multiple maximally general types are possible, but also it may lead to getting the most general type without the need for backtracking across iterations of the main algorithm, which unfortunately often takes prohibitively long.

We look at the types of substitutions for the invariant-holding variables `bvs` in partial answer `ans`, and try to form the initial candidates for choice 6 so that the return type variables cover the most of argument types variables, for each `bvs` type found. We select from the candidates equations between any variable (case `sb`) or only non-argument-type variable (case `b_sb`), and a FV(argument types)\FV(return type) variable – we turn the equation so that the latter is the RHS. We locate the equations among the candidates that have a `bvs` variable or a FV(return type)\FV(argument types) variable as LHS. We apply the substitution `sb` (or `b_sb`) to the RHS of these equations (`b_sb` if the LHS is in `bvs`). We preserve the order of equations in the candidate list.

3.2 Joint constraint abduction

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 accumulate simple abduction answers into the partial abduction answer, we set aside branches that do not have any answer satisfiable with the partial answer so far. After all branches have been tried and the partial answer is not an empty conjunction (i.e. not $\top$), we retry the set-aside branches. If during the retry, any of the set-aside branches fails, we add the partial answer to discarded answers – which are avoided during simple abduction – and restart. Restart puts the set-aside branches to be tried first. If, when left with set-aside branches only, the partial answer is an empty conjunction, i.e. all the answer-contributing branches have been set aside, we fail – return $\bot$ from the joint abduction. This does not peform complete backtracking (no completeness guarantee), but is therefore quicker to report unsolvable cases and does sufficient backtracking. After an answer working for all branches has been found, we perform additional check, which encapsulates negative constraints introduced by the `assert false` construct. If the check fails, we add the answer to discarded answers and repeat the search.

If a partial answer becomes as strong as one of the discarded answers inside SCA, simple constraint abduction skips to find a different answer. The discarded answers are initialized with a discard list passed from the main algorithm.

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.

Searching for abduction answer can fail in only one way: we have set aside all the branches that could contribute to the answer. It is difficult to pin-point the culprit. We remember which branch caused the restart when the number of set-aside branches was the smallest. We raise exception `Suspect` that contains the conclusion of that branch.

The core algorithm shared across sorts is provided in the `Joint` module.

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`. [FIXME: this approach might not work.] This process is needed regardless of the "dissociation" issue, to uncover the full content of numeric sort constraints.

To face efficiency of numerical abduction with many variables, we modify the approach. On the first iteration of the main algorithm, we remove alien subterms both from the branches and from the answer (the `purge` function), but we do not perform other-sort abduction at all. On the next iteration, we do not purge alien subterms, neither from the branches nor from the answer, 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 6). They also have more information to work with, present in the instatiation of partial answers. However, this optimization violates completeness guarantees of the combination of sorts algorithm. To faciliate finding term abduction solutions that hold under the quantifiers, we substitute-out other sort variables, by variables more to the left in the quantifier, using equations from the premise.

The dissociation interacts with the discard list mechanism. Since dissociation introduces fresh variables, no answers with alien subterms would be syntactically identical. When checking whether a partial answer should be discarded, in case alien subterm dissociation is *on*, we ignore alien sort subterms in the comparison.

3.4 Simple constraint abduction for linear arithmetics

For checking validity or satisfiability, we use *Fourier-Motzkin elimination*. To avoid complexities we only handle the rational number domain. To extend the algorithm to integers, *Omega-test* procedure as presented in [1] needs to be adapted. The major operations are:

- *Elimination* of a variable takes an equation and selects a variable that is not upstream, i.e. to the left in $\mathcal{Q}$ regarding alternations, 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 is not 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. When implicit equalities have been found, we iterate the process to normalize them as well.

Our abduction algorithm follows a familiar incrementally-generate-and-test scheme as in term abduction. There are two new ideas, which can also be applied to abduction in other domains. We build a lazy list of possible transformations with linear combinations involving equations a implied by $D \wedge C$. We pair each inequality c in C with all inequalities d implied by D which share a variable with c and try out the abduction answers to $d \Rightarrow c$ as contributions to the partial abduction answer to $D \Rightarrow C$.

To simplify the search in presence of a quantifier prefix, we preprocess the initial candidate by eliminating universally quantified variables:

$$\begin{aligned} S &= [\overline{\beta_u} := \bar{t_u}] \text{ for } \overline{\forall \beta_u} \subset \mathcal{Q} \text{ such that } \mathcal{M} \vDash D \Rightarrow \dot{S}, \\ \text{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D, C) &= \{c' | c \in C, \text{if } \mathcal{M} \vDash \mathcal{Q}.c[\bar{\beta} := \bar{t}\,] \text{ for some } \bar{t} \text{ then } c' = c \text{ else } c' = S(c)\} \end{aligned}$$

Before accepting a new atom into the partial answer, we check that it would not violate the quantifier conditions from the *split* phase of the main algorithm, and that the partial answer is satisfiable with all implication branches of the joint abduction problem. As part of the quantifier conditions, we ensure that the escaping parameters are upward in the constraint before prenexization, i.e. are parameters of the type of a parent definition (containing a given definition in its body) rather than a parallel definition. The check of satisfiability we call *validation*. For domains other than the term domain, validation also involves instantiating use-sites of recursive definitions with parts of the partial answer, split in a simplified way.

Abduction algorithm:

1. Let $C^{=\prime} = \widetilde{A_i}(C^{=})$, resp. $C^{\leqslant\prime} = \widetilde{A_i}(C^{\leqslant})$ where $C^{=}$, resp. $C^{\leqslant}$ are the equations, resp. inequalities in C and $\widetilde{A_i}$ is the substitution according to equations in A_i. Let $D' = \widetilde{A_i}(D \wedge A_i)$ and $D^{=\prime}$ be the equations in D', i.e. substituted equations and implicit equalities. Let $D^{\leqslant\prime}$ be a solved form of inequalities.
    a. Let $C_0^{=} = \text{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D^{=\prime}, C^{=\prime})$ and $C_0^{\leqslant} = \text{Rev}_\forall\big(\mathcal{Q}, \bar{\beta}, D^{=\prime}, C^{\leqslant\prime}\big)$.
2. Prepare the initial transformations from atoms $a \in D^{=\prime}$:
    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.

   b. The final transformations have the form: $b \mapsto b + \Sigma_{a \in D} k_a^{s_a} a$.

3. Modify $C^{=\prime}$ to promote answers with variables rather than constants, as in term abduction: For $\alpha_1 \doteq \tau, ..., \alpha_n \doteq \tau \in C^{=\prime}$, form the substitution of subterms $\alpha_1 := \alpha_i, ..., \alpha_n := \alpha_i, \tau := \alpha_i$ (excluding $\alpha_i := \alpha_i$) where α_i is the most upstream existential variable (or parameter) and τ is a constant.
4. Start from $\text{Acc} := \{\}$ and $C_0 := C^{=\prime} \wedge C^{\leqslant\prime}$. Handle atoms a in $C_0 = a C_0'$, equations first.
5. Let $B = A_i \wedge D \wedge C_0' \wedge \text{Acc}$.
6. If a is a tautology ($0 \doteq 0$ or $c \leq 0$ for $c \leqslant 0$) or $B \Rightarrow C$, repeat with $C_0 := C_0'$. Corresponds to choice 1 of term abduction.
7. If $B \not\Rightarrow C$, for a candidate a' generated for a, starting with a, which passes validation against other branches in a joint problem: $\text{Acc} := \text{Acc} \cup \{a'\}$, or fail if all a' fail.
   a. If a is an inequality, let a_0 be an abduction answer to $d \Rightarrow \text{Rev}_\forall\big(\mathcal{Q}, \bar{\beta}, D^{=\prime}, \widetilde{\text{Acc}^{=}}(a)\big)$, where d belongs to the solved form inequalities implied by D'. Otherwise, let $a_0 = \widetilde{\text{Acc}^{=}}(a)$.
   b. Sort the candidates a_0 in order of decreasing value, described below. Let a' be a_0 with some $D^{=\prime}$-derived transformation applied.
      - If the branch constrint B already bounds a variable by a constant, we de-prioritize candidates a_0 that are other-side bounds of the variable by a constant. Otherwise, we increase the value of a_0 for each variable that is bound by a_0 on the side that it is unbound in B. We decrease the value of a_0 for other variables to favor smaller size.
      - Currently, we do not perform transformations when a_0 is an inequality, for simplicity and speed at cost of missing some answers. However, we eliminate the universal variables, i.e. we use $\text{Rev}_\forall$ above. If it proves necessary, we will also try the transformations for the inequalities. For example, by trying all candidates a_0 before proceeding to the next transformation.
   c. If $A_i \wedge (\text{Acc} \cup \{a'\})$ (resp. $A_i \wedge (\text{Acc} \cup \{a''\})$) does not pass validation for all a', backtrack.
   d. If $A_i \wedge (\text{Acc} \cup \{a'\})$ (resp. $A_i \wedge (\text{Acc} \cup \{a''\})$) passes validation, repeat from step 5 with $C_0 := C_0', \text{Acc} := \text{Acc} \cup \{a'\}$ (resp. $\text{Acc} := \text{Acc} \cup \{a''\}$).
8. The answers are $A_{i+1} = A_i \wedge \text{Acc}$.

We precompute the tranformation variants to try out. The parameter n is called `abd_rotations` and defaults to a small value (currently 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.

To find the abduction answers to $d \Rightarrow c$, pick a common variable $\alpha \in \text{FV}(d) \cap \text{FV}(c)$ or the constant $\alpha = 1$. We have four possibilities:

1. $d \Leftrightarrow \alpha \leqslant d_\alpha$ and $c \Leftrightarrow \alpha \leqslant c_\alpha$: the abduction answers are c and $d_\alpha \leqslant c_\alpha$,
2. $d \Leftrightarrow \alpha \leqslant d_\alpha$ and $c \Leftrightarrow c_\alpha \leqslant \alpha$: the abduction answer is only c,
3. $d \Leftrightarrow d_\alpha \leqslant \alpha$ and $c \Leftrightarrow \alpha \leqslant c_\alpha$: the abduction answer is only c,
4. $d \Leftrightarrow d_\alpha \leqslant \alpha$ and $c \Leftrightarrow c_\alpha \leqslant \alpha$: the abduction answers are c and $c_\alpha \leqslant d_\alpha$.

Thanks to cases (1) and (4) above, the abduction algorithm can find some answers which are not fully maximal. The joint constraint abduction algorithm can help in some of the remaining cases where fully maximal abduction is insufficient for some implications, by solving simpler implications first.

We provide an optional optimization: we do not pass, in the first call to numerical abduction (the second iteration of the main algorithm), branches that contain unary predicate variables in the conclusion, i.e. we only use the "non-recursive" branches. Other optimizations that we use are: iterative deepening on the constant n used to generate k^s factors. We also constrain the algorithm by filtering out transformations that contain "too many" variables, which can lead to missing answers if the setting `abd_prune_at` – "too many" – is too low. Similarly to term abduction, we count the number of steps of the loop and fail if more than `abd_timeout_count` steps have been taken.

4 Constraint Generalization

Constraint generalization 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 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. We use "disjunction elimination" as a synonym of "constraint generalization".

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 $V = \{x_j, \overline{t_{i,j}} \mid \forall i \exists t_{i,j}. x_j \doteq t_{i,j} \in D_{i,s_{\text{ty}}}\}$.
   b. Let $G = \{\bar{\alpha}_j, u_j, \overline{\theta_{i,j}} \mid \theta_{i,j} = [\bar{\alpha}_j := \bar{g}_j^{\,i}], \theta_{i,j}(u_j) = t_{i,j}\}$ 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 $D_i^g = D_{i,s_{\text{ty}}} \wedge D_i^u$.
   d. Let $D_i^v = \{x \doteq y \mid x \doteq t_1 \in D_i^g, y \doteq t_2 \in D_i^g, D_i^g \vDash t_1 \doteq t_2\}$.
   e. Let $A_{s_{\text{ty}}} = \wedge_j x_j \doteq u_j \wedge \bigcap_i (D_i^g \wedge D_i^v)$ (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 $\bar{\alpha}_{s_{\text{ty}}} = \overline{\alpha_j}$.
   f. Let $\wedge_s D_{i,s}^u \equiv D_i^u$ for $D_{i,s}^u$ of sort s.
3. For sorts $s \neq s_{\text{ty}}$, let $\exists \bar{\alpha}_s. A_s = \text{DisjElim}_s(\overline{D_i^s \wedge D_{i,s}^u})$.
4. The answer is $\exists \overline{\overline{\alpha_i^j} \bar{\alpha}_s}. \wedge_s A_s$.

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

We follow the anti-unification algorithm provided in [5], 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 optimality 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 equalities 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. Separately, we compute the equations common to all branches, because the convex hull algorithm is not guaranteed to recover them.

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 6.3.

While reading section 6.3, you will notice that postconditions are not subjected to stratification. This is because the type system does not support nested existential types.

In `simplify_dsjelim`, we try to preserve alien variables that are parameters rather than substituting them by constants. A parameter can only equal a constant if not all branches have been considered for disjunction elimination. The parameter is both as informative as the constant, and less likely to contradict other branches.

4.3 Abductive disjunction elimination given quantifier prefix

Global variables here are the variables shared by all disjuncts, i.e. $\cap_i \mathrm{FV}(D_i)$, remaining variables are *non-global*. Recall that for numerical disjunction elimination, we either substitute-out a non-global variable in a branch if it appears in an equation, or we drop the inequalities it appers in if it is not part of any equation. Non-global variables can also pose problems for the term sort, by forcing disjunction elimination answers to be too general. When inferring the type for a function, which has a branch that does not use one of arguments of the function, the existential type inferred would hide the corresponding information in the result, even if the remaining branches assume the argument has a single concrete type. We would like the corresponding non-local variable to resolve to the concrete type suggested by other branches of the resulting constraint.

We extend the notion of disjunction elimination: substitution U and solved form $\exists \bar{\alpha}.A$ is an answer to *abductive disjunction elimination* problem $\overline{D_i}$ given a quantifier prefix $\mathcal{Q}$ when:

1. $(\forall i) \vDash U(D_i) \Rightarrow \exists \bar{\alpha} \setminus \mathrm{FV}(U).A$;
2. If $\alpha \in \mathrm{Dom}(U)$, then $(\exists \alpha) \in \mathcal{Q}$ – variables substituted by U are existentially quantified;
3. $(\forall i) \vDash \forall(\mathrm{Dom}(U))\exists(\mathrm{FV}(D_i) \setminus \mathrm{Dom}(U)).D_i$.

The sort-integrating algorithm essentially does not change:

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 $V = \{x_j, \overline{t_{i,j}} \mid \forall i \exists t_{i,j}. x_j \doteq t_{i,j} \in D_{i,s_{\text{ty}}}\}$.
   b. Let $G = \{\bar{\alpha}_j, g_j, u_j, \overline{\theta_{i,j}} \mid \theta_{i,j} = [\bar{\alpha}_j := \bar{g}_j^{\,i}], \theta_{i,j}(g_j) = (\wedge_{k \leqslant j} u_k)(t_{i,j})\}$ be the most specific anti-unifiers of $\overline{(\wedge_{k \leqslant j} u_k)(t_{i,j})}$ for each j, where $\wedge_j u_j$ is the resulting U.
   c. Let $D_i^u = \wedge_j \bar{\alpha}_j \doteq \bar{g}_j^{\,i}$ and $D_i^g = D_{i,s_{\text{ty}}} \wedge D_i^u$.
   d. Let $D_i^v = \{x \doteq y \mid x \doteq t_1 \in D_i^g, y \doteq t_2 \in D_i^g, D_i^g \vDash t_1 \doteq t_2\}$.
   e. Let $A_{s_{\text{ty}}} = \wedge_j x_j \doteq g_j \wedge \bigcap_i (D_i^g \wedge D_i^v)$ (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 $\bar{\alpha}_{s_{\text{ty}}} = \overline{\bar{\alpha}_j}$.
   f. Let $\wedge_s D_{i,s}^u \equiv D_i^u$ for $D_{i,s}^u$ of sort s.
3. For sorts $s \neq s_{\text{ty}}$, let $\exists \bar{\alpha}_s. A_s = \text{DisjElim}_s(\overline{D_i^s \wedge D_{i,s}^u})$.
4. The answer is substitution $U = \wedge_j u_j$ and solved form $\exists \overline{\overline{\alpha_i^j} \bar{\alpha}_s}. \wedge_s A_s$.

Our current generalized anti-unification algorithm:

$$
\begin{array}{rcl}
\text{au}_{U,G}(t;\ldots;t) & = & \varnothing, t, U, G \\
\text{au}_{U,G}(f(\bar{t}^{\,1});\ldots;f(\bar{t}^{\,n})) & = & \bar{\alpha}\,, f(\bar{g}\,), U', G' \\
\text{where } \bar{\alpha}\,, \bar{g}\,, U', G' & = & \text{aun}_{U,G}(\bar{t}^{\,1};\ldots;\bar{t}^{\,n}) \\
\text{au}_{U,G}(t_1;\ldots;t_n) & = & \varnothing, \alpha, U, G \\
\text{when} & & ([t_1;\ldots;t_n] \mapsto \alpha) \in G \\
\text{au}_{U,G}(\ldots;\beta_i;\ldots;f(\bar{t}^{\,j});\ldots \text{ as } \bar{t}\,) & = & \bar{\alpha}\bar{\alpha}', g, U'', G' \\
\text{where } \bar{\alpha}', g, U'', G' & = & \text{au}_{U',G}(\bar{t}\,[\beta_i := f(\bar{\alpha}\,)]) \\
U' & = & U[\beta_i := f(\bar{\alpha}\,)] \wedge \beta_i \doteq f(\bar{\alpha}\,) \\
\text{when} & & \exists \beta_i \in \mathcal{Q}, \text{ treat } \bar{\alpha} \text{ as quantified with } \exists \beta_i \\
\text{au}_{U,G}(\ldots;\beta_i;\ldots;\beta_j;\ldots \text{ as } \bar{t}\,) & = & \bar{\alpha}', g, U'', G' \\
\text{where } \bar{\alpha}', g, U'', G' & = & \text{au}_{U',G}(\bar{t}\,[\beta_i := \beta_j]) \\
U' & = & U[\beta_i := \beta_j] \wedge \beta_i \doteq \beta_j \\
\text{when} & & \exists \beta_i \in \mathcal{Q}, \beta_j \leqslant_{\mathcal{Q}} \beta_i \\
\text{au}_{U,G}(t_1;\ldots;t_n) & = & \alpha, \alpha, U, ([t_1;\ldots;t_n] \mapsto \alpha)G \\
\text{otherwise,} & & \text{where } \alpha \# \text{FV}(t_1;\ldots;t_n, U, G) \\
\text{aun}_{U,G}(\varnothing) & = & \varnothing, \varnothing, U, G \\
\text{aun}_{U,G}(\varnothing;\ldots) & = & \varnothing, \varnothing, U, G \\
\text{aun}_{U,G}(t_1^1, \bar{t}^{\,1};\ldots;t_1^n, \bar{t}^{\,n}) & = & \bar{\alpha}\bar{\alpha}', g\bar{g}', U'', G'' \\
\text{where } \bar{\alpha}\,, g, U', G' & = & \text{au}_{U,G}(t_1^1;\ldots;t_1^n) \\
\text{and } \bar{\alpha}', \bar{g}', U'', G'' & = & \text{aun}_{U',G'}(\bar{t}^{\,1};\ldots;\bar{t}^{\,n})
\end{array}
$$

The notational shorthand ...; β_i; ...; $f(\bar{t}^{\,j})$; ... represents the case where all terms are either existential variables or start with a function symbol f. Similarly, ...; β_i; ...; β_j; ... represents the case when there is a variable β_j such that all terms are either β_j or are existential variables to the right of β_j in the quantifier.

The task of disjunction elimination is to find postconditions. Usually, it is beneficial to make the postcondition, i.e. the existential type that is the return type of a function, more specific, at the expense of making the overal type of the function less general. To this effect, disjunction elimination, just as abduction takes invariant parameters $\bar{\beta}$. We replace conditions $\exists \beta_i \in \mathcal{Q}$ above by $\beta_i \in \bar{\beta} \vee \exists \beta_i \in \mathcal{Q}$. Recall that the right-hand-side (RHS) variable β_j can in general be universally quantified: $\forall \beta_j \in \mathcal{Q}$. We exclude universal non-parameter RHS when a parameter is present: if for any β_i, $\beta_i \in \bar{\beta}$, then for all β_i including RHS, $\beta_i \in \bar{\beta} \vee \exists \beta_i \in \mathcal{Q}$. Note that having weaker postconditions also results in correct types, just not the intended ones. In rare cases a

weaker postcondition but a more general invariant can be beneficial. To this effect, the option `-more_existential` turns off generating the substitution entries when the RHS is a variable, i.e. the case $\mathrm{au}_{U,G}(...;\beta_i;...;\beta_j;...\text{ as }\bar{t}\,)$ is skipped.

Due to greater flexibility of the numerical domain, abductive extension of numerical disjunction elimination does not seem necessary and is turned off by default. It could take a similar form, we experiment with the following heuristic. If atoms specific to a disjunct (i.e. not shared by all disjuncts) do not contain a variable, do not include the disjunct when considering inclusion of inequalities containing the variable in the constraint generalization answer.

4.4 Incorporating negative constraints

We call a *negative constraint* an implication $D \Rightarrow \boldsymbol{F}$ in the normalized constraint $\mathcal{Q}.\wedge_i (D_i \Rightarrow C_i)$, and we call D the *negated constraint*. Such constraints are generated for pattern matching branches whose right-hand-side is the expression `assert false`. A generic approach to account for negative constraints is as follows. We check whether the solution found by multisort abduction contradicts the negated constraints. If some negated constraint is satisfiable, we discard the answer and "fall back" to try finding another answer. Unfortunately, this approach is insufficient when the answer sought for is not among the maximally general answers to the remaining, *positive part* of the constraint. Therefore, we introduce *negation elimination*.

For the numerical sort, our implementation of negation elimination can produce too strong constraints when the numerical domain is intended to contain non-integer rational numbers, and can be turned off by the `-no_int_negation` option. It can also produce too weak constraints, because it produces at most one atom for a given negated constraint. If the abduction answer for terms does already contradict a negated constraint D, we are done. Otherwise, let $D = c_1 \wedge ... \wedge c_n$. For numerical sort atoms c_i, either drop them from consideration or convert their negation $\neg c_i$ into d_i or $d_{i_1} \vee d_{i_2}$ as follows. The conversion assumes that the numerical domain is integers. We convert an inequality $w \leqslant 0$, e.g. $\frac{1}{3}x - \frac{1}{2}y - 2 \leqslant 0$, to $-k\,w + 1 \leqslant 0$, e.g. $3y - 2x + 13 \leqslant 0$, where k is the common denominator so that $k\ w$ has only integer numbers. Note that $\neg(w \leqslant 0) \Leftrightarrow w > 0 \Leftrightarrow -w < 0 \Leftrightarrow -k\,w < 0 \Leftarrow -k\,w + 1 \leqslant 0$. Similarly, we convert an equation $w \doteq 0$ to inequalities $-k\,w + 1 \leqslant 0 \vee k\,w + 1 \leqslant 0$. Note that $\neg(w \geqslant 0) \Leftrightarrow w < 0 \Leftrightarrow k\,w < 0 \Leftarrow k\,w + 1 \leqslant 0$. In both cases the implications $\Leftarrow$ would be equivalences if the numerical domain was integers rather than rational numbers. At present, we ignore *opti* atoms. The disjunct d_i is a conjunction of inequalities if c_i is a *subopti* atom.

Assuming that each negative constraint points to a single atomic fact, we try to find one disjunct d_i, resp. d_{i_1} or d_{i_2}, corresponding to $\neg c_i$, discarding those disjuncts that contradict any implication branch. Specifically, let $\mathcal{Q}.\wedge_i (D_i \Rightarrow C_i)$ be the constraint we solve, and let $\exists\bar{\alpha}\,.A$ be a term abduction answer for $\mathcal{Q}.\wedge_{i:C_i \neq \boldsymbol{F}} (D_i \Rightarrow C_i)$. We search for i such that for all k with $C_k \neq \boldsymbol{F}$ and D_k satisfiable, $d_i \wedge A \wedge D_k \wedge C_k$ is satisfiable. We provide a function $\mathrm{NegElim}(\neg D, \overline{B_i}\,) = d_{i_0}$, where d_{i_0} is the biggest, syntactically, such atom, and $B_i = A \wedge D_i \wedge C_i$.

Unfortunately, for the sort of terms we do not have well-defined representation of dis-equalities not using negations. The generic approach of relying on backtracking to find another abduction answer is more useful for terms than for the numeric sort. It falls short, however, when negation was intended to prevent the answer from being too general. Ideally, we would introduce disequation atoms $\tau \dot{\neq} \tau$ and follow the scheme we use for the numerical sort. For now, we only cover a very specific use of negation, to discriminate among type-level "enumeration". We limit negation elimination to considering atoms of the form $\beta \doteq \varepsilon_1$, and contradict them by introducing atoms $\beta \doteq \varepsilon_2$, for types $\varepsilon_1 \neq \varepsilon_2$ without parameters which we call *phantom enumerations*. The variables β are limited to the answer variables generated in the previous iteration of the main algorithm. The nullary datatype constructor ε_2 is picked so that the atom is valid, using the same validation procedure as the one passed to the abduction algorithm. The heuristic defines phantom enumerations as nullary phantom types that do not share datype parameter position (in GADT constructor definitions) with non-enumeration types. When the equations derived for different negated constraints involve a common variable as the left-hand-side, we select a common right-hand-side. In the end, we only introduce the negation elimination result to the answer when a single disjunct remains.

Since the *discard* (or taboo) list used by backtracking is based on complete answers, it is preferable to perform negation elimination prior to abduction. Otherwise, the search might fall into a loop where abduction keeps returning the same answer, since it is more general than the discarded answer incorporating negation elimination result.

5 *opti* and *subopti*: *minimum* and *maximum* relations in `num`

We extend the numerical domain with relations *opti* and *subopti* defined below. Operations *min* and *max* can then be defined using it. Let k, v, w be any linear combinations. Note that the relations are introduced to smuggle in a limited form of disjunction into the solved forms.

$$\begin{aligned}
\text{opti}(v, w) &= v \leqslant 0 \wedge w \leqslant 0 \wedge (v \doteq 0 \vee w \doteq 0) \\
k \doteq \min(v, w) &\equiv \text{opti}(k - v, k - w) \\
k \doteq \max(v, w) &\equiv \text{opti}(v - k, w - k) \\
\text{subopti}(v, w) &= v \leqslant 0 \vee w \leqslant 0 \\
k \leqslant \max(v, w) &\equiv \text{subopti}(k - v, k - w) \\
\min(v, w) \leqslant k &\equiv \text{subopti}(v - k, w - k)
\end{aligned}$$

In particular, $\text{opti}(v, w) \equiv \max(v, w) \doteq 0$ and $\text{subopti}(v, w) \equiv \min(v, w) \leqslant 0$. We call an *opti* or *subopti* atom *directed* when there is a variable n that appears in v and w with the same sign. We do not prohibit undirected *opti* or *subopti* atoms, but we do not introduce them, to avoid bloat.

For simplicity, we do not support *min* and *max* as subterms in concrete syntax. Instead, we parse atoms of the form `k=min(...,...)`, resp. `k=max(...,...)` into the corresponding *opti* atoms, where `k` is any numerical term. Similarly for *subopti*. We also print directed *opti* and *subopti* atoms using the syntax with min and max expressions. Not to pollute the syntax with a new keyword, we use concrete syntax `min|max(...,...)` for parsing arbitrary, and printing non-directed, *opti* atoms, and `min||max(...,...)` for *subopti* respectively.

If need arises, in a future version, we can extend *opti* to a larger arity N.

5.1 Normalization, validity and implication checking

In the solved form producing function `solve_aux`, we treat *opti* clauses in an efficient but incomplete manner, doing a single step of constraint solving. We include the *opti* terms in processed inequalities. After equations have been solved, we apply the substitution to the *opti* and *subopti* disjunctions. When one of the *opti* resp. *subopti* disjunct terms becomes contradictory or the disjunct terms become equal, we include the other in implicit equalities, resp. in inequalities to solve. When one of the *opti* or *subopti* terms becomes tautological, we drop the disjunction. Recall that we iterate calls of `solve_aux` to propagate implicit equalities.

We do not perform case splitting on *opti* and *subopti* disjunctions, therefore some contradictions may be undetected. However, abduction and disjunction elimination currently perform upfront case splitting on *opti* and *subopti* disjunctions, sometimes leading to splits that a smarter solver would avoid.

5.2 Abduction

We eliminate *opti* and *subopti* in premises by expanding the definition and converting the branch into two branches, e.g. $D \wedge (v \doteq 0 \vee w \doteq 0) \Rightarrow C$ into $(D \wedge v \doteq 0 \Rightarrow C) \wedge (D \wedge w \doteq 0 \Rightarrow C)$. Recall that an *opti* atom also implies inequalities $v \leqslant \wedge w \leqslant 0$ assumed to be in D above. This is one form of *case splitting*: we consider cases $v \doteq 0$ and $w \doteq 0$, resp. $v \leqslant 0$ and $w \leqslant 0$, separately. We do not eliminate *opti* and *subopti* in conclusions. Rather, we consider whether to keep or drop it in the answer, like with other candidate atoms. The transformations apply to an *opti* atom by applying to both its arguments.

Generating a new *opti* atom for inclusion in an answer means finding a pair of equations such that the following conditions hold. Each equation, together with remaining atoms of an answer but without the remaining equation selected, is a correct answer to a simple abduction problem. The equations selected share a variable and are oriented so that the variable appears with the same sign in them. The resulting *opti* atom passes the validation test for joint constraint abduction. We may implement generating new *opti* atoms for abduction answers in a future version, when need arises. Currently, we only generate new *opti* and *subopti* atoms for postconditions, i.e. during disjunction elimination.

5.3 Disjunction elimination

We eliminate *opti* and *subopti* atoms prior to finding the extended convex hull of $\overline{D_i}$ by expanding the definition and converting the disjunction $\vee_i D_i$ to disjunctive normal form. This is another form of case splitting.

In addition to finding the extended convex hull, we need to discover *opti* relations that are implied by $\vee_i D_i$. We select these faces of the convex hull which also appear as an equation in some disjuncts. Out of these faces, we find all minimal covers of size 2, i.e. pairs of faces such that in each disjunct, either one or the other linear combination appears as an equation. We only keep pairs of faces that share a same-sign variable. For the purposes of detecting *opti* relations, we need to perform transitive closure of the extended convex hull equations and inequalities, because the redundant inequalities might be required to find a cover.

Finding *subopti* atoms is similar. We find all minimal covers of size 2, i.e. pairs of inequalities such that one or the other appears in each disjunct. We only keep pairs of inequalities that share a same-sign variable.

We provide an `initstep_heur` function for the numerical domain to remove *opti* atoms of the form $k \doteq \min(c, v)$, $k \doteq \min(v, c)$, $k \doteq \max(c, v)$ or $k \doteq \min(v, c)$ for a constant c, similarly for *subopti* atoms, while in initial iterations where disjunction elimination is only performed for non-recursive branches.

6 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 6.5.

6.1 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.

6.2 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 for $\mathrm{Split}\big(\mathcal{Q}, \bar\alpha\,, A, \overline{\bar\beta^{\,\chi}}\,, \overline{A^0_\chi}\,\big)$. Note that due to existential types predicates, we actually compute $\mathrm{Split}\Big(\mathcal{Q}, \bar\alpha\,, A, \overline{\bar\beta^{\,\beta_\chi}}\,, \overline{A^0_{\beta_\chi}}\,\Big)$, i.e. we index by β_χ (which can be multiple for a single χ) rather than χ. We retain the notation indexing by χ 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}{rrcl}
& \alpha \prec \beta & \equiv & \alpha <_{\mathcal{Q}} \beta \vee \big(\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}}\,\big) \\
& A_{\alpha\beta} & = & \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_0 & = & A \setminus A_{\alpha\beta} \\
& A^1_\chi & = & \{c \in A_0 | \forall \alpha \in \mathrm{FV}(c) \setminus \bar\beta^{\,\chi}. (\exists\alpha) \in \mathcal{Q} \vee \alpha <_{\mathcal{Q}} \beta_\chi\} \\
& A^2_\chi & = & \mathrm{Atomized}\big(\overline{\bar\beta^{\,\chi}}\,, A^1_\chi\big) \\
& A^3_\chi & = & A^2_\chi \setminus \cup_{\chi'} A^1_{\chi'} \\
& \text{if} & & \mathcal{M} \not\models \mathcal{Q}.(A \setminus \cup_\chi A^2_\chi)[\bar\alpha := \bar t\,] \ \text{ for all } \bar t \\
& \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^2_\chi) \wedge \mathcal{M} \models \mathcal{Q}.(\cup_\chi A^+_\chi \Rightarrow A)[\bar\alpha := \bar t\,] \ \text{ for some } \bar t: \\
& \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 \vee \cup_\chi A^3_\chi \neq \varnothing \ \text{ then} \\
& \mathcal{Q}', 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}} \wedge_\chi A^3_\chi, \\
& & & \qquad \overline{\bar\beta^{\,\chi} \cup \bar\alpha^{\,\chi}}\,, \overline{A_\chi}\,\big) \\
& \text{return} & & \mathcal{Q}', A_{\alpha\beta} \wedge A'_{\mathrm{res}}, \overline{\exists \bar\alpha^{\,\chi} \bar\alpha'^{\chi}. A'_\chi} \\
& \text{else return} & & \mathcal{Q} \exists (\bar\alpha\ \setminus \cup_\chi \bar\alpha^{\,\chi}), A_{\alpha\beta} \wedge 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.

Description of the algorithm in more detail:

1. $\alpha \prec \beta \ \equiv\ \alpha <_{\mathcal{Q}} \beta \vee \big(\alpha \leqslant_{\mathcal{Q}} \beta \wedge \beta \not<_{\mathcal{Q}} \alpha \wedge \alpha \in \overline{\bar\beta^{\,\chi}} \wedge \beta \notin \overline{\bar\beta^{\,\chi}}\,\big)$ 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. $A_0 \ =\ A \setminus A_{\alpha\beta}$ where $A_{\alpha\beta} \ =\ \big\{\beta \doteq \alpha \in A \big| \beta \in \overline{\bar\beta^{\,\chi}} \wedge (\exists\alpha) \in \mathcal{Q} \wedge \beta \prec \alpha\big\}$ Discard the "scaffolding" information, in particular the A_+ equations introduced by Strat in an earlier iteration.

3. $A^1_\chi \ =\ \{c \in A_0 | \forall \alpha \in \mathrm{FV}(c) \setminus \bar\beta^{\,\chi}. (\exists\alpha) \in \mathcal{Q} \vee \alpha <_{\mathcal{Q}} \beta_\chi\}$ Gather atoms pertaining to predicate χ, which should not remain in the residuum.

4. $A^2_\chi \ =\ \mathrm{Atomized}\big(\overline{\bar\beta^{\,\chi}}\,, A^1_\chi\big)$ An atomized form of A^1_χ wrt. $\overline{\bar\beta^{\,\chi}}$: a conjunction of atoms equivalent to A^1_χ containing some of the implications of A^1_χ, see the formal exposition of InvarGenT. Actually, rather than computing the atomized form upfront, we generate its contribution to A^+_χ while performing Fourier-Motzkin elimination to check $\mathcal{M} \models \mathcal{Q}.(\cup_\chi A^+_\chi \Rightarrow A)$.

5. $A^3_\chi = A^2_\chi \setminus \cup_{\chi'} A^1_{\chi'}$ We prune atoms coming from atomization that are already present in the invariants. Actually, in the implementation we prune by redundancy with the invariants assigned so far.

6. if $\mathcal{M} \not\models \mathcal{Q}.(A \setminus \cup_\chi A^1_\chi)[\bar\alpha := \bar t\,]$ for all $\bar t$ 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.

7. for all $\overline{A^+_\chi}$ min. w.r.t. $\subset$ s.t. $\wedge_\chi(A^+_\chi \subset A^2_\chi) \wedge \mathcal{M} \models \mathcal{Q}.(\cup_\chi A^+_\chi \Rightarrow A)[\bar\alpha := \bar t\,]$ for some $\bar t$: 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.

8. 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.

9. $\bar\alpha^\chi_+,\ A^L_\chi,\ A^R_\chi = \text{Strat}(A^+_\chi,\ \bar\beta^{\,\chi})$ 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 will add $\bar\alpha^\chi_+$ to $\bar\beta^{\,\chi}$.

10. $A_\chi = A^0_\chi \cup A^L_\chi$ is the updated solution formula for χ, where A^0_χ is the solution from previous round.

11. $\bar\alpha^\chi_0 = \bar\alpha \cap \text{FV}(A_\chi)$ are the additional solution parameters coming from variables generated by abduction.

12. $\bar\alpha^{\,\chi} = \Big(\bar\alpha^\chi_0 \setminus \bigcup_{\chi' <_{\mathcal{Q}} \chi} \bar\alpha^{\chi'}_0\Big)\bar\alpha^\chi_+$ The final solution parameters also include the variables generated by Strat.

13. $A_+ = \cup_\chi A^R_\chi$ 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_+.

14. if $\cup_\chi \bar\alpha^{\,\chi} \neq \varnothing \vee \cup_\chi A^3_\chi \neq \varnothing$ then – If new parameters or new atoms have been introduced, we need to redistribute the remaining atoms to make $\mathcal{Q}'.A_{\text{res}}$ valid again.

15. $\mathcal{Q}', A'_{\text{res}}, \overline{\exists \bar\alpha'^{\chi}.A'_\chi} \in \text{Split}\big(\mathcal{Q}\big[\overline{\forall \bar\beta^{\,\chi}} := \overline{\forall(\bar\beta^{\,\chi} \cup \bar\alpha^{\,\chi})}\big], \bar\alpha \setminus \cup_\chi \bar\alpha^{\,\chi}, A_{\text{res}} \wedge_\chi A^3, \overline{\bar\beta^{\,\chi} \cup \bar\alpha^{\,\chi}}, \overline{A_\chi}\big)$ Recursive call includes $\bar\alpha^\chi_+$ in $\bar\beta^{\,\chi}$ so that, among other things, A_+ are redistributed into A_χ.

16. return $\mathcal{Q}', A'_{\text{res}}, \overline{\exists \bar\alpha^{\,\chi} \bar\alpha'^{\chi}.A'_\chi}$ 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.

17. else return $\mathcal{Q}\exists(\bar\alpha \setminus \cup_\chi \bar\alpha^{\,\chi}), A_{\text{res}}, \overline{\exists \bar\alpha^{\,\chi}.A_\chi}$ 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 $\text{Split}(\bar\alpha\,,\ A) := \text{Split}(\bar\alpha\,,\ A,\ \bar{\boldsymbol{T}})$. The complete algorithm for solving predicate variables is presented in the next section.

6.3 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. Let $\chi_{\varepsilon K}$ be the invariant which will constrain the existential type ε_K, or $\top$. When $\chi_{\varepsilon K} = \top$, then we define $\bar{\beta}^{\,\chi_{\varepsilon K},k} = \varnothing$. Step k of the loop of the final algorithm:

$$
\begin{aligned}
\overline{\exists \bar{\beta}^{\,\chi,k}.F_\chi} &= S_k \\
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, && (1)\\
&\quad \overline{C_j^\alpha} = \{C \,|\, D \Rightarrow C \in S_k(\Phi) \wedge D \subseteq D_K^\alpha\} \\
U_{\chi_K}, \exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K} &\quad \mathrm{DisjElim}\Big(\overline{\delta \doteq \alpha \wedge D_K^\alpha \wedge_j C_j^\alpha}_{\,\alpha \in \overline{\alpha_3^{i,K}}}\Big) && (2)\\
\exists \bar{\alpha}_e^{\chi_K}.G'_{\chi_K} &= \mathrm{Simpl}(\mathrm{FV}(G_{\chi_K}) \backslash \delta \bar{\beta}^{\,\chi_{\varepsilon K},k}.G_{\chi_K}) && (3)\\
\vec{\tau}_{\varepsilon_K} &= \overrightarrow{\big\{\alpha \in \mathrm{FV}(G'_{\chi_K}) \backslash \delta\delta' \bar{\alpha}_g^{\chi_K} \big| (\exists \alpha) \in \mathcal{Q} \vee \alpha \in \overline{\beta^{\chi}} \backslash \overline{\beta^{\chi_K}}\big\}} \\
\Xi(\exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K}) &= \exists \mathrm{FV}(\vec{\tau}_{\varepsilon_K}, G'_{\chi_K}) \backslash \delta\delta'.\delta' \doteq \vec{\tau}_{\varepsilon_K} \wedge G'_{\chi_K} \\
R_g(\chi_K) = \exists \bar{\alpha}^{\,\chi_K}.F_{\chi_K} &= H(R_k(\chi_K), \Xi(\exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K})) && (4)\\
P_g(\chi_K) &= \delta' \doteq \vec{\tau}_{\varepsilon_K} \wedge \exists \bar{\alpha}^{\,\chi_K}.F_{\chi_K} \\
F'_\chi &= F_\chi[\varepsilon_K(\vec{\tau}_{\mathrm{old}}) := \varepsilon_K(\vec{\tau}_{\varepsilon_K})[\mathrm{recover}(\vec{\tau}_{\varepsilon_K}, \vec{\tau}_{\mathrm{old}})]] \\
S'_k &= \overline{\exists \bar{\beta}^{\,\chi,k} \{\alpha \in \mathrm{FV}(F'_\chi) | \beta_\chi <_{\mathcal{Q}} \alpha\}.F'_\chi} && (5)\\
\mathcal{Q}'.\wedge_i (D_i \Rightarrow C_i) \wedge_j (D_j^- \Rightarrow \boldsymbol{F}) &= R_g^- P_g^+ S'_k(\Phi \wedge_{\chi_K} U_{\chi_K}) \\
\exists \bar{\alpha}.A_0 &= \mathrm{Abd}\big(\mathcal{Q}', \bar{\beta} = \overline{\beta_\chi \beta^{\chi}}, \overline{D_i, C_i}\big) && (6)\\
A &= A_0 \wedge_j \mathrm{NegElim}\big(\neg \mathrm{Simpl}\big(\mathrm{FV}(D_j^-) \backslash \overline{\beta^{\chi}}.D_j^-\big), A_0, \overline{D_i, C_i}\big) \\
&\quad \text{At later iterations, check negative constraints.} && (7)\\
\big(\mathcal{Q}^{k+1}, A_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\,\beta_\chi}.A_{\beta_\chi}}\big) &= \mathrm{Split}\big(\mathcal{Q}', \bar{\alpha}, A, \overline{\beta_\chi \beta^{\chi}}\big) \\
\vec{\tau}'_{\varepsilon_K} &= \overrightarrow{\mathrm{FV}\big(\widetilde{A_{\mathrm{res}}(\vec{\tau}_{\varepsilon_K})}\big)} \\
R_{k+1}(\chi_K) &= \exists \bar{\beta}^{\,\chi_K,k} \overline{\bar{\alpha}^{\,\beta_{\chi_K}}} \bar{\alpha}^{\,\chi_K} \backslash \mathrm{FV}(\vec{\tau}'_{\varepsilon_K}).\delta' \doteq \vec{\tau}'_{\varepsilon_K} \wedge \widetilde{A_{\mathrm{res}}}(F_{\chi_K} \backslash \delta' \doteq ...) \\
&\quad \wedge_{\chi_K} A_{\beta_{\chi_K}}\Big[\, \overline{\beta_{\chi_K} \beta^{\,\beta_{\chi_K}}} := \overline{\delta \bar{\beta}^{\,\chi_K,k}} \,\Big] && (8)\\
&= R'_g(\chi_K) \\
S_{k+1}(\chi) &= \exists \bar{\beta}^{\,\chi,k}.\mathrm{Simpl}\big(\exists \overline{\bar{\alpha}^{\,\beta_\chi}}.F'_\chi \wedge A_{\beta_\chi}\big[\, \overline{\beta_\chi \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 (\forall \beta_{\chi_K}) A_{\beta_{\chi_K}} = \boldsymbol{T}, \\
&\quad k > 1 \\
\text{then return} &\quad A_{\mathrm{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 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 $\Big[\, \overline{\beta_{\chi_K} \beta^{\,\beta_{\chi_K}}} := \overline{\delta \bar{\beta}^{\,\chi_K,k}} \,\Big]$ 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 parameter instances of the postcondition are connected with the argument variables. As a matter of implementation, the substitution recover$(\vec{\tau}_{\varepsilon_K}, \vec{\tau}_{\text{old}})$ is actually derived from the way the variables are freshened in step 5 above. Its effect is currently implemented as a substitution over the intermediate formula $F_\chi[\varepsilon_K(\vec{\tau}_{\text{old}}) := \varepsilon_K(\vec{\tau}_{\varepsilon_K})]$.

We start with $S_0 := \bar{\boldsymbol{T}}$ and $R_0 := \bar{\boldsymbol{T}}$. 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 $c \in G$. Connected$_0(\delta, (\bar{\alpha}, G))$ additionally removes atoms c: FV$(c) \# \delta\bar{\alpha}$. 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. $H(R_k, R_{k+1})$ does not perform simplification, only pruning.

The condition $(\forall \beta_{\chi_K}) A_{\beta_{\chi_K}} = \boldsymbol{T}$ is required for correctness of returned answer. Fixpoint of postconditions $R_k(\chi_K)$ is not a sufficient stopping condition, because it does not imply $A_{\beta_{\chi_K}} = \boldsymbol{T}$, the same $A_{\beta_{\chi_K}} \neq \boldsymbol{T}$ may be introduced in consecutive iterations. This is not the case for invariants, where $S_k(\chi)$ and $S_{k+1}(\chi)$ differ by the portion A_{β_χ} of abduction answer.

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 $\{D_j^-\} = \{D_i | \boldsymbol{F} \in C_i\}$. After the main algorithm ends we check that for each D_j^-, $S_k(D_i)$ fails. Optionally, but by default, we perform the check in each iteration, starting from third, i.e. `iter_no>1`. Turning this option *on* gives a way to express negative constraints for the term domain. The option should be turned *off* when a single iteration (plus fallback backtracking described below) is insufficient to solve for the invariants. Moreover, for domains with negation elimination, i.e. the numerical domain, we incorporate negative constraints as described in section 4.4. The corresponding computation is $A_0 \wedge_j \text{NegElim}\big(\neg\text{Simpl}\big(\text{FV}(D_j^-) \backslash \overline{\beta^\chi}.D_j^-\big), \overline{D_i, C_i}\big)$ in the specification of the main algorithm. The simplification reduces the number of atoms in the formula and keeps those that are most relevant to finding invariants and postconditions. For convenience, negation elimination is called from the `Abduction.abd` function, which returns $\exists\bar{\alpha}.A$.

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 $D_i \in \Phi_{\boldsymbol{F}}$, $\mathcal{M} \not\models \exists \text{FV}(S_k(D_i)).S_k(D_i)$ fails. In step $k-1$, we maintain a discard list of different answers found not to work in this step and previous steps: 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.

6.4 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. Abduction solves for invariants and helps solve for postconditions:

1. $j_0 \leqslant k < j_2$ Only term abduction – invariants of type shapes – is performed, for all branches.

2. $k < j_1$ Do not perform abduction for postconditions. Remove atoms with variables containing postcondition parameters from conclusions sent to abduction.

3. $j_2 \leqslant k < j_3$ Both term abduction and numerical abduction are performed, but numerical abduction only for non-recursive branches.

4. $j_3 \leqslant k$ Abduction is performed on all branches – type and numerical invariants are found.

Default settings is $[j_0;\ j_1;\ j_2;\ j_3] = [0;\ 1;\ 1;\ 2]$. j_1 is not tied to j_0, j_2, j_3. We have options: `early_postcond_abd` and `early_num_abduction` that set $j_1 = 0$ and $j_3 = 1$ respectively. 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 $[k_0; k_1; k_2; k_3] = [0; 0; 2; 5]$. We have an option `converge_at` for setting k_3.

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.

6.5 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 $\wedge_\chi(A^+_\chi \subset A^1_\chi) \wedge \mathcal{M} \vDash \mathcal{Q}.A \setminus \cup_\chi A^+_\chi$ is an incremental validity checker. It starts with $A \setminus \cup_\chi A^1_\chi$ and tries to add as many atoms $c \in \cup_\chi A^1_\chi$ as possible to what in effect becomes A_{res}.

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 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.

7 Generating OCaml/Haskell Source and Interface Code

We have a single basis from which to generate all generated output files: `.gadti`, `.ml`, and in the future `.hs` – `annot_item`. It contains a superset of information in `struct_item`: type scheme annotations on introduced names, and source code annotated with type schemes at recursive definition nodes. We use `type a.` syntax instead of `'a.` syntax because the former supports inference for GADTs in OCaml. A benefit of the nicer `type a.` syntax is that nested type schemes can have free variables, which will be correctly captured by the outer type scheme. For completeness we sometimes need to annotate all `function` nodes with types. To avoid clutter, we start by only annotating `let rec` nodes, and in case `ocamlc -c` fails on generated code, we re-annotate by putting type schemes on `let rec` nodes and types on `function` nodes. If need arises, `let-in` node annotations can also be introduced in this fallback – the corresponding `Lam` constructors store the types. We provide an option to annotate the definitions on `let-in` nodes. Type annotations are optional because they introduce a slight burden on the solver – the corresponding variables cannot be removed by the initial simplification of the constraints, in `Infer`. `let-in` node annotations are more burdensome than `function` node annotations.

Annotated items `annot_item` use "nice" named variables instead of identifier-based variables. The renaming is computed by `nice_ans`, called at the toplevel and at each `let rec` in the source code.

In the signature declarations `ITypConstr` and `IValConstr` for existential types, we replace `Extype` with `CNam` as indentifiers of constructors, to get informative output for printing the various result files. We print constraint formulas and alien subterms in the original InvarGenT syntax, commented out.

The types `Int`, `Num` and `Bool` should be considered built-in. `Int` and `Bool` follow the general scheme of exporting a datatype constructor with the same name, only lower-case. However, numerals `0`, `1`, ... are always type-checked as `Num 0`, `Num 1`... A parameter `-num_is` decides the type alias definition added in the generated code: `-num_is bar` adds `type num = bar` in front of an `.ml` file, by default `int`. Numerals are exported as integers passed to a `bar_of_int` function. The variant `-num_is_mod` exports numerals by passing to a `Bar.of_int` function. Special treatment for `Bool` amounts to exporting `True` and `False` as `true` and `false`, unlike other constants. In addition, pattern matching `match ... with True -> ... | False -> ...`, i.e. the corresponding beta-redex, is exported as `if ... then ... else ...`.

In declarations `PrimVal`, which have concrete syntax starting with the word `external`, we provide names assumed to have given type scheme. The syntax has two variants, differing in the way the declaration is exported. It can be either an `external` declaration in OCaml, which is the binding mechanism of the *foreign function interface*. But in the InvarGenT form `external let`, the declaration provides an OCaml definition, which is exported as the toplevel `let` definition of OCaml. It has the benefit that the OCaml compiler will verify this definition, since InvarGenT calls `ocamlc -c` to verify the exported code.

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] Chuan-kai Lin. *Practical type inference for the GADT type system*. PhD dissertation, Portland State University, Department of Computer Science, 2010.

[4] 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_3$0.

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