Constraint-based type inference can be used for reconstruction of preconditions, invariants and postconditions of recursive functions written in languages with GADTs.

- InvarGenT infers preconditions and invariants as types of recursive definitions, and postconditions as existential types.
- Generalized Algebraic Data Types type system $\mathrm{MMG}(X)$ based on François Pottier and Vincent Simonet's $\mathrm{HMG}(X)$ but without type annotations.
- Extended to a language with existential types represented as implicitly defined and used GADTs.
- Type inference problem as satisfaction of second order constraints over a multi-sorted domain.
- Invariants found by *Joint Constraint Abduction under Quantifier Prefix*, postconditions found by *constraint generalization* – anti-unification for terms, extended convex hull.
- A numerical sort with linear equations and inequalities over rationals, and $k \doteq \min(m, n)$, $k \doteq \max(m, n)$ relations (reconstructed only for postconditions).

The Type System

Patterns (syntax-directed)

p-Empty

$$C \vdash 0 : \tau \longrightarrow \exists \varnothing [\boldsymbol{F}] \{\}$$

p-Wild

$$C \vdash 1 : \tau \longrightarrow \exists \varnothing [\boldsymbol{T}] \{\}$$

p-And

$$\frac{\forall i \quad C \vdash p_i : \tau \longrightarrow \Delta_i}{C \vdash p_1 \wedge p_2 : \tau \longrightarrow \Delta_1 \times \Delta_2}$$

p-Var

$$C \vdash x : \tau \longrightarrow \exists \varnothing [\boldsymbol{T}] \{x \mapsto \tau\}$$

p-Cstr

$$\frac{\forall i \quad C \wedge D \vdash p_i : \tau_i \longrightarrow \Delta_i \quad K :: \forall \bar{\alpha} \bar{\beta} [D] . \tau_1 \times \ldots \times \tau_n \to \varepsilon(\bar{\alpha}) \quad \bar{\beta} \# \mathrm{FV}(C)}{C \vdash K\, p_1 \ldots p_n : \varepsilon(\bar{\alpha}) \longrightarrow \exists \bar{\beta} [D] (\Delta_1 \times \ldots \times \Delta_n)}$$

Clauses

Clause

$$\frac{C \vdash p : \tau_1 \longrightarrow \exists \bar{\beta} [D] \Gamma' \quad C \wedge D, \Gamma\Gamma' \vdash e : \tau_2 \quad \bar{\beta} \# \mathrm{FV}(C, \Gamma, \tau_2)}{C, \Gamma \vdash p.e : \tau_1 \to \tau_2}$$

WhenClause

$$\frac{\begin{array}{ll} C \wedge D, \Gamma\Gamma' \vdash m_i : \mathrm{Num}(\tau_{m_i}) & e \neq \textbf{assert false} \wedge \ldots \wedge \\ C \wedge D, \Gamma\Gamma' \vdash n_i : \mathrm{Num}(\tau_{n_i}) & e \neq \lambda(p' \ldots \lambda(p'' . \textbf{assert false})) \\ C \vdash p : \tau_1 \longrightarrow \exists \bar{\beta} [D] \Gamma' & C \wedge D \wedge_i \tau_{m_i} \leqslant \tau_{n_i}, \Gamma\Gamma' \vdash e : \tau_2 \quad \bar{\beta} \# \mathrm{FV}(C, \Gamma, \tau_2) \end{array}}{C, \Gamma \vdash p \,\textbf{when}\, \wedge_i m_i \leqslant n_i . e : \tau_1 \to \tau_2}$$

NegClause

$$\frac{\begin{array}{ll} C \wedge D, \Gamma\Gamma' \vdash m_i : \mathrm{Num}(\tau_{m_i}) & e = \textbf{assert false} \vee \ldots \vee \\ C \wedge D, \Gamma\Gamma' \vdash n_i : \mathrm{Num}(\tau_{n_i}) & e = \lambda(p' \ldots \lambda(p'' . \textbf{assert false})) \\ C \vdash p : \tau_3 \longrightarrow \exists \bar{\beta} [D] \Gamma' & C \wedge D \wedge \tau_1 \doteq \tau_3 \wedge_i \tau_{m_i} \leqslant \tau_{n_i}, \Gamma\Gamma' \vdash e : \tau_2 \quad \bar{\beta} \# \mathrm{FV}(C, \Gamma, \tau_2) \end{array}}{C, \Gamma \vdash p \,\textbf{when}\, \wedge_i m_i \leqslant n_i . e : \tau_1 \to \tau_2}$$

Patterns (non-syntax-directed)

p-EqIn

$$\frac{C \vdash p : \tau' \longrightarrow \Delta \quad C \vDash \tau \doteq \tau'}{C \vdash p : \tau \longrightarrow \Delta}$$

p-SubOut

$$\frac{C \vdash p : \tau \longrightarrow \Delta' \quad C \vDash \Delta' \leqslant \Delta}{C \vdash p : \tau \longrightarrow \Delta}$$

p-Hide

$$\frac{C \vdash p : \tau \longrightarrow \Delta \quad \bar{\alpha} \# \mathrm{FV}(\tau, \Delta)}{\exists \bar{\alpha} . C \vdash p : \tau \longrightarrow \Delta}$$

Existential Type System extension – modified App, added rules

App

$$\frac{C, \Gamma, \Sigma \vdash e_1 : \tau' \to \tau \quad C, \Gamma, \Sigma \vdash e_2 : \tau' \quad C \vDash \not\in(\tau')}{C, \Gamma, \Sigma \vdash e_1\, e_2 : \tau}$$

ExLetIn

$$\frac{\varepsilon_K(\bar{\alpha}) \text{ in } \Sigma \quad C, \Gamma, \Sigma \vdash e_1 : \tau' \quad C, \Gamma, \Sigma \vdash K\, p . e_2 : \tau' \to \tau}{C, \Gamma, \Sigma \vdash \textbf{let}\ p = e_1\ \textbf{in}\ e_2 : \tau}$$

ExIntro

$$\frac{\mathrm{Dom}(\Sigma') \backslash \mathrm{Dom}(\Sigma) = \mathcal{E}(e) \quad C, \Gamma, \Sigma' \vdash n(e) : \tau}{C, \Gamma, \Sigma \vdash e : \tau}$$

Expressions (syntax-directed)

Var

$$\frac{\Gamma(x) = \forall \beta [\exists \bar{\alpha} . D] . \beta \quad C \vDash D}{C, \Gamma \vdash x : \beta}$$

AssertFalse

$$\frac{C \vDash \boldsymbol{F}}{C, \Gamma \vdash \textbf{assert false} : \tau}$$

Cstr

$$\frac{\forall i\, C, \Gamma \vdash e_i : \tau_i \quad C \vDash D \quad K :: \forall \bar{\alpha} \bar{\beta} [D] . \tau_1 \ldots \tau_n \to \varepsilon(\bar{\alpha})}{C, \Gamma \vdash K e_1 \ldots e_n : \varepsilon(\bar{\alpha})}$$

LetIn

$$\frac{C, \Gamma \vdash \lambda(p . e_2)\, e_1 : \tau}{C, \Gamma \vdash \textbf{let}\ p = e_1\ \textbf{in}\ e_2 : \tau}$$

App

$$\frac{C, \Gamma \vdash e_1 : \tau' \to \tau \quad C, \Gamma \vdash e_2 : \tau'}{C, \Gamma \vdash e_1\, e_2 : \tau}$$

LetRec

$$\frac{C, \Gamma' \vdash e_1 : \sigma \quad C, \Gamma' \vdash e_2 : \tau \quad \sigma = \forall \beta [\exists \bar{\alpha} . D] . \beta \quad \Gamma' = \Gamma\{x \mapsto \sigma\}}{C, \Gamma \vdash \textbf{letrec}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

Abs

$$\frac{\forall i\, C, \Gamma \vdash c_i : \tau_1 \to \tau_2}{C, \Gamma \vdash \lambda(c_1 \ldots c_n) : \tau_1 \to \tau_2}$$

Expressions (non-syntax-directed)

Gen

$$\frac{C \wedge D, \Gamma \vdash e : \beta \quad \beta \bar{\alpha} \# \mathrm{FV}(\Gamma, C)}{C \wedge \exists \beta \bar{\alpha} . D, \Gamma \vdash e : \forall \beta [\exists \bar{\alpha} . D] . \beta}$$

Inst

$$\frac{C, \Gamma \vdash e : \forall \bar{\alpha} [D] . \tau' \quad C \vDash D[\bar{\alpha} := \bar{\tau}]}{C, \Gamma \vdash e : \tau'[\bar{\alpha} := \bar{\tau}]}$$

DisjElim

$$\frac{C, \Gamma \vdash e : \tau \quad D, \Gamma \vdash e : \tau}{C \vee D, \Gamma \vdash e : \tau}$$

Hide

$$\frac{C, \Gamma \vdash e : \tau \quad \bar{\alpha} \# \mathrm{FV}(\Gamma, \tau)}{\exists \bar{\alpha} . C, \Gamma \vdash e : \tau}$$

Equ

$$\frac{C, \Gamma \vdash e : \tau \quad C \vDash \tau \doteq \tau'}{C, \Gamma \vdash e : \tau'}$$

FElim

$$\frac{}{\boldsymbol{F}, \Gamma \vdash e : \tau}$$

Existential Type System extension – ExIntro processing

$$\begin{array}{ll} n(e, K') = \textbf{let}\ x = n(e, \bot)\ \textbf{in}\ K'\, x & \text{for } K' \neq \bot \wedge l(e) = \boldsymbol{F} \\ n(x, \bot) = x & \\ n(\lambda \bar{c}, \bot) = \lambda(\overline{n(c, \bot)}) & \\ n(e_1\, e_2, K') = n(e_1, K')\, n(e_2, \bot) & \\ n(\lambda[K] \bar{c}, \bot) = \lambda(\overline{n(c, K)}) & \\ n(\lambda[K] \bar{c}, K') = \lambda(\overline{n(c, K')}) & \text{for } K' \neq \bot \\ n(p.e, K') = p.n(e, K') & \\ n(\textbf{let}\ p = e_1\ \textbf{in}\ e_2, K') = \textbf{let}\ p = n(e_1, \bot)\ \textbf{in}\ n(e_2, K') & \end{array}$$

AVL Trees – create

slide9.gadt:

```
datatype Avl : type * num
datacons Empty : ∀a. Avl (a, 0)
datacons Node :
  ∀a,k,m,n [k=max(m,n) ∧ 0⩽m ∧ 0⩽n ∧ n⩽m+2 ∧ m⩽n+2].
     Avl (a, m) * a * Avl (a, n) * Num (k+1) ⟶ Avl (a, k+1)

let height = function
  | Empty -> 0
  | Node (_, _, _, k) -> k

let create = fun l x r ->
  ematch height l, height r with
  | i, j when j <= i -> Node (l, x, r, i+1)
  | i, j when i <= j -> Node (l, x, r, j+1)
```

Result:

```
val height : ∀n, a. Avl (a, n) → Num n
val create :
  ∀k, n, a[k ⩽ n + 2 ∧ n ⩽ k + 2 ∧ 0 ⩽ k ∧ 0 ⩽ n].
  Avl (a, k) → a → Avl (a, n) → ∃i[i=max (k + 1, n + 1)].Avl (a, i)
```

slide11.gadt:

```
datatype Avl : type * num
datacons Empty : ∀a. Avl (a, 0)
datacons Node :
  ∀a,k,m,n [k=max(m,n) ∧ 0⩽m ∧ 0⩽n ∧ n⩽m+2 ∧ m⩽n+2].
     Avl (a, m) * a * Avl (a, n) * Num (k+1) ⟶ Avl (a, k+1)

let rec min_binding = function
  | Empty -> assert false
  | Node (Empty, x, r, _) -> x
  | Node ((Node (_,_,_,_) as l), x, r, _) -> min_binding l
```

shell:

```
# invargent slide11.gadt -inform
val min_binding : ∀n, a[1 ⩽ n]. Avl (a, n) → a
```

equal1_wrong.gadt: compare two values of types as encoded

```
datatype Ty : type
datatype List : type
datacons Zero : Int
datacons Nil : ∀a. List a
datacons TInt : Ty Int
datacons TPair : ∀a, b. Ty a * Ty b ⟶ Ty (a, b)
datacons TList : ∀a. Ty a ⟶ Ty (List a)
external let eq_int : Int → Int → Bool = "(=)"
external let b_and : Bool → Bool → Bool = "(&&)"
external let b_not : Bool → Bool = "not"
external forall2 : ∀a, b. (a → b → Bool) → List a → List b → Bool = "forall2"

let rec equal1 = function
  | TInt, TInt -> fun x y -> eq_int x y
  | TPair (t1, t2), TPair (u1, u2) ->
    (fun (x1, x2) (y1, y2) ->
        b_and (equal1 (t1, u1) x1 y1)
              (equal1 (t2, u2) x2 y2))
  | TList t, TList u -> forall2 (equal1 (t, u))
  | _ -> fun _ _ -> False
```

Result: `val equal1 : ∀a, b. (Ty a, Ty b) → a → a → Bool`

Exercise 1. Find remaining three maximally general types of `equal1`.

Pick Intended Type with `assert false`

equal_assert.gadt:

[...]

```
let rec equal = function
  | TInt, TInt -> fun x y -> eq_int x y
  | TPair (t1, t2), TPair (u1, u2) ->
    (fun (x1, x2) (y1, y2) ->
        b_and (equal (t1, u1) x1 y1)
              (equal (t2, u2) x2 y2))
  | TList t, TList u -> forall2 (equal (t, u))
  | _ -> fun _ _ -> False
  | TInt, TList l -> (function Nil -> assert false)
  | TList l, TInt -> (fun _ -> function Nil -> assert false)
```

Result:

```
val equal : ∀a, b. (Ty a, Ty b) → a → b → Bool
```

Pick Intended Type with a `test` Clause

equal_test.gadt:

```
[...]
let rec equal = function
  | TInt, TInt -> fun x y -> eq_int x y
  | TPair (t1, t2), TPair (u1, u2) ->
    (fun (x1, x2) (y1, y2) ->
        b_and (equal (t1, u1) x1 y1)
              (equal (t2, u2) x2 y2))
  | TList t, TList u -> forall2 (equal (t, u))
  | _ -> fun _ _ -> False
test b_not (equal (TInt, TList TInt) zero Nil)
```

OCaml code generated – *equal_test.ml*:

```
[...]
let rec equal : type a b . ((a ty * b ty) -> a -> b -> bool) =
  (function (TInt, TInt) -> (fun x y -> eq_int x y)
    | (TPair (t1, t2), TPair (u1, u2)) ->
        (fun ((x1, x2)) ((y1, y2)) ->
          b_and (equal ((t1, u1)) x1 y1) (equal ((t2, u2)) x2 y2))
    | (TList t, TList u) -> forall2 (equal ((t, u)))
    | _ -> (fun _ _ -> false))
let () = assert (b_not (equal ((TInt, TList TInt)) zero Nil)); ()
```

InvarGenT Handles Many Non-pointwise Cases

Chuan-kai Lin developed an efficient type inference algorithm for GADTs, however in a type system restricted to so-called pointwise types.

Toy example – *non_pointwise_split.gadt*:

```
datatype Split : type * type
datacons Whole : Split (Int, Int)
datacons Parts : ∀a, b. Split ((Int, a), (b, Bool))
external let seven : Int = "7"
external let three : Int = "3"

let joint = function
  | Whole -> seven
  | Parts -> three, True
```

Needs non-default setting – *shell*:

```
# invargent non_pointwise_split.gadt -inform -richer_answers
val joint : ∀a. Split (a, a) → a
```

Exercise 2. Check that this is the correct type.

InvarGenT does not Handle Some Non-sensible Cases

A solution to at least one branch of implications, correspondingly of pattern matching, must be implied by the conjunction of the premise and the conclusion of the branch. I.e., some branch must be solvable without arbitrary guessing. If solving a branch requires guessing, for some ordering of branches, the solution to already solved branches must be a good guess.

non_pointwise_vary.gadt:

```
datatype EquLR : type * type * type
datacons EquL : ∀a, b. EquLR (a, a, b)
datacons EquR : ∀a, b. EquLR (a, b, b)
datatype Box : type
datacons Cons : ∀a. a ⟶ Box a
external let eq : ∀a. a → a → Bool = "(=)"
let vary = fun e y ->
  match e with
  | EquL, EquL -> eq y "c"
  | EquR, EquR -> Cons (match y with True -> 5 | False -> 7)
```

shell:

```
# invargent non_pointwise_vary.gadt -inform
File "non_pointwise_vary.gadt", line 11, characters 18-60:
No answer in type: term abduction failed
```

Exercise 3. Find a type or two for `vary`. Check that the type does not meet the above requirement.

avl_tree.gadt:

```
[...]
let rec add = fun x -> efunction
  | Empty -> Node (Empty, x, Empty, 1)
  | Node (l, y, r, h) ->
    ematch compare x y, height l, height r with
    | EQ, _, _ -> Node (l, x, r, h)
    | LT, hl, hr ->
      let l' = add x l in
      (ematch height l' with
       | hl' when hl' <= hr+2 -> create l' y r
       | hl' when hr+3 <= hl' -> rotr (l', y, r))
    | GT, hl, hr ->
      let r' = add x r in
      (ematch height r' with
       | hr' when hr' <= hl+2 -> create l y r'
       | hr' when hl+3 <= hr' -> rotl (l, y, r'))
```

Result:

```
val add : ∀a,n.a→Avl(a, n)→∃k[k ≤ n+1 ∧ 1 ≤ k ∧ n ≤ k].Avl(a, k)
```

avl_tree.gadt:

```
[...]
let rotr = efunction
    | Empty, x, r -> assert false
    | Node (ll, lx, lr, hl), x, r ->
      assert num 0 <= height r;  (* Need assertions to guide inference *)
      assert type hl = height r + 3;
      (ematch height ll, height lr with
      | m, n when n <= m ->
        let r' = create lr x r in
        create ll lx r'
      | m, n when m+1 <= n ->
        (ematch lr with
        | Empty -> assert false
        | Node (lrl, lrx, lrr, _) ->
          let l' = create ll lx lrl in
          let r' = create lrr x r in
          create l' lrx r'))
```

Result:

```
val rotr : ∀a,n[0≤n].(Avl(a,n+3),a,Avl(a,n))→∃k[n+3≤k∧k≤n+4].Avl(a,k)
```

Theorem 1. Correctness *(expressions)*. $[\![\Gamma \vdash \mathrm{ce} : \tau]\!], \Gamma \vdash \mathrm{ce} : \tau$.

Theorem 2. Completeness *(expressions)*. *If* $\mathrm{PV}(C, \Gamma) = \varnothing$ *and* $C, \Gamma \vdash \mathrm{ce} : \tau$, *then there exists an interpretation of predicate variables* $\mathcal{I}$ *such that* $\mathcal{I}, \mathcal{M} \vDash C \Rightarrow [\![\Gamma \vdash \mathrm{ce} : \tau]\!]$.

Remarks on Solving

- We use an extension of *fully maximal Simple Constraint Abduction* – "fully maximal" is the restriction that we do not guess facts not implied by premise and conclusion of a given implication.
- Without existential types, the problem in principle is caused by the complexity of constraint abduction – not known to be decidable. Given a correct program with appropriate `assert false` clauses, and using an oracle for Simple Constraint Abduction, the intended type scheme will ultimately be found.
  - This could be shown formally, but the proof is very tedious.
- Without existential types, the problem in practice is that although the *fully maximal* restriction, when not imposed on all branches but with accumulating the solution as discussed on slide 22, seems sufficient for practical programs, fully maximal SCA is still exponential in the size of input.
- With existential types, there are no guarantees. The intended solution to the postconditions could be missed by the algorithm.
  - We have not yet found a definite, and practical, such counterexample.

$\exists \bar{\alpha}.A$ is a JCAQP$_{\mathcal{M}}$ answer (answer to a *Joint Constraint Abduction under Quantifier Prefix problem* $\mathcal{Q}.\wedge_i (D_i \Rightarrow C_i)$ for model $\mathcal{M}$) when:

- A is a conj. of atoms in "solved form",
- $\bar{\alpha} \# \mathrm{FV}(\wedge_i (D_i \Rightarrow C_i))$,
- *relevance*: $\mathcal{M} \vDash \wedge_i (D_i \wedge A \Rightarrow C_i)$,
- *validity*: $\mathcal{M} \vDash \forall \bar{\alpha} \mathcal{Q}.A$,
- *consistency*: $\mathcal{M} \vDash \wedge_i \forall \bar{\alpha} \exists (\bar{\alpha}^c).D_i \wedge A$.

Open problem: decidable?

Answer $\exists \bar{\alpha}.A$ to SCAQP$_{\mathcal{M}}$ problem $\mathcal{Q}.D \Rightarrow C$ is *fully maximal* when

$$\mathcal{M} \vDash (\exists \bar{\alpha}.D \wedge A) \Leftrightarrow D \wedge C.$$

Nondeterministic algorithm for $T(F)$:

```
FMA(D,C):
    if (D ⇒ C) return ⊤
    let A be standard form of D ∧ C
    loop
        let 𝒜 be next(A)
        if (∀A′ ∈ 𝒜.A′ ∧ D ⇏ C) return A
        choose A ∈ 𝒜 s.t. A ∧ D ⇒ C
```

$\mathrm{next}(A) = \{\mathrm{repl}(S, A) | S$: some positions of a term $t\}$

$\mathrm{repl}(S, A)$ replaces terms at positions S by a new parameter.

Joint Constraint Abduction Algorithm

- When solving multiple implications $\wedge_i(D_i \Rightarrow C_i)$, keep a partial answer $\exists \bar{\alpha}_p.A_p$.
- To solve a particular implication $D_i \Rightarrow C_i$, find $\exists \bar{\alpha}.A$ s.t. $\exists \bar{\alpha}\bar{\alpha}_p.A \wedge A_p$ is the SCAQP$_{\mathcal{M}}$ answer: $\mathcal{M} \vDash (\exists \bar{\alpha}\bar{\alpha}_p.D \wedge A \wedge A_p) \Leftrightarrow D \wedge C$ etc.
  - In the algorithm for $T(F)$ on previous slide, the initial A is still $D_i \wedge C_i$, but the implication checks are modified: we check $A' \wedge A_p \wedge D \not\Rightarrow C$, $A \wedge A_p \wedge D_i \Rightarrow C_i$.
- In case of success, $\exists \bar{\alpha}_{p+1}.A_{p+1} = \exists \bar{\alpha}\bar{\alpha}_p.A \wedge A_p$.
- A general way to deal with failure would be: try all permutations $i_1, ..., i_p, ...$
- In practice we need to give up sooner.
  - If branch $D_i \Rightarrow C_i$ fails when approached with partial answer $\exists \bar{\alpha}_p.A_p$, $A_p \neq \top$, restart from $D_i \Rightarrow C_i$ with $A_{p+1} = \top$.
  - If branch $D_i \Rightarrow C_i$ fails for $A_p = \top$, restart from the next unsolved branch (e.g. $D_{i+1} \Rightarrow C_{i+1}$) with $A_{p+1} = \top$ and put $D_i \Rightarrow C_i$ aside.
  - Solve branches set aside after other branches.
  - **Fail** if solving an aside branch fails.
- Backtracking: a discard list, aka. taboo list, of visited answers.

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

$$\mathrm{Rev}_{\forall}(\mathcal{Q}, \bar{\beta}, D, C) = \{S(c) | c \in D \wedge C, S = [\bar{\beta_u} := \bar{t_u}] \text{ for } \overline{\forall \beta_u} \subset \mathcal{Q}, \mathcal{M} \models D \Rightarrow \bar{\beta_u} \doteq \bar{t_u}, \mathcal{M} \models \mathcal{Q}.S(c)[\bar{\beta} := \bar{t}] \text{ for some } \bar{t}\}$$

- We cannot use Herbrandization because it does not preserve equivalence.
  - Herbrandization works for general abduction defined by logical validity, but not for constraint abduction defined by validity in a fixed model.

References

- Vincent Simonet and Francois Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), JAN 2007.
- 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.
- M. Sulzmann, T. Schrijvers and P. J. Stuckey. Type inference for GADTs via Herbrand constraint abduction.
- Kenneth W. Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 505–519. Springer, 2007.
- Peter Bulychev, Egor Kostylev and Vladimir Zakharov. Anti-unification algorithms and their applications in program analysis. In Amir Pnueli, Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer Berlin / Heidelberg, 2010.
- 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*.