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 *disjunction elimination* – e.g. 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).

Patterns (syntax-directed)

p-Empty
$$C \vdash 0 \colon \tau \longrightarrow \exists \varnothing[\boldsymbol{F}]\{\}$$

p-Wild
$$C \vdash 1 \colon \tau \longrightarrow \exists \varnothing[\boldsymbol{T}]\{\}$$

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

p-Var
$$C \vdash x \colon \tau \longrightarrow \exists \varnothing[\boldsymbol{T}]\{x \mapsto \tau\}$$

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

Clauses

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

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

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

Patterns (non-syntax-directed)

p-EqIn
$$\frac{\begin{array}{l} C \vdash p \colon \tau' \longrightarrow \Delta \\ C \vDash \tau \doteq \tau' \end{array}}{C \vdash p \colon \tau \longrightarrow \Delta}$$

p-SubOut
$$\frac{\begin{array}{l} C \vdash p \colon \tau \longrightarrow \Delta' \\ C \vDash \Delta' \leqslant \Delta \end{array}}{C \vdash p \colon \tau \longrightarrow \Delta}$$

p-Hide
$$\frac{\begin{array}{l} C \vdash p \colon \tau \longrightarrow \Delta \\ \bar{\alpha}\#\mathrm{FV}(\tau, \Delta) \end{array}}{\exists \bar{\alpha}.C \vdash p \colon \tau \longrightarrow \Delta}$$

Existential Type System extension – modified App, added rules

App
$$\frac{\begin{array}{l} C, \Gamma, \Sigma \vdash e_1 \colon \tau' \rightarrow \tau \\ C, \Gamma, \Sigma \vdash e_2 \colon \tau' \quad C \vDash \not{E}(\tau') \end{array}}{C, \Gamma, \Sigma \vdash e_1\, e_2 \colon \tau}$$

ExLetIn
$$\frac{\begin{array}{l} \varepsilon_K(\bar{\alpha}) \text{ in } \Sigma \quad C, \Gamma, \Sigma \vdash e_1 \colon \tau' \\ C, \Gamma, \Sigma \vdash K\, p.e_2 \colon \tau' \rightarrow \tau \end{array}}{C, \Gamma, \Sigma \vdash \textbf{let } p = e_1 \textbf{ in } e_2 \colon \tau}$$

ExIntro
$$\frac{\begin{array}{l} \mathrm{Dom}(\Sigma') \backslash \mathrm{Dom}(\Sigma) = \mathcal{E}(e) \\ C, \Gamma, \Sigma' \vdash n(e) \colon \tau \end{array}}{C, \Gamma, \Sigma \vdash e \colon \tau}$$

Expressions (syntax-directed)

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

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

Cstr
$$\frac{\begin{array}{l} \forall i\, C, \Gamma \vdash e_i \colon \tau_i \qquad C \vDash D \\ K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \ldots \tau_n \rightarrow \varepsilon(\bar{\alpha}) \end{array}}{C, \Gamma \vdash K\, e_1 \ldots e_n \colon \varepsilon(\bar{\alpha})}$$

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

App
$$\frac{\begin{array}{l} C, \Gamma \vdash e_1 \colon \tau' \rightarrow \tau \\ C, \Gamma \vdash e_2 \colon \tau' \end{array}}{C, \Gamma \vdash e_1\, e_2 \colon \tau}$$

LetRec
$$\frac{\begin{array}{ll} C, \Gamma' \vdash e_1 \colon \sigma & C, \Gamma' \vdash e_2 \colon \tau \\ \sigma = \forall \beta[\exists \bar{\alpha}.D].\beta & \Gamma' = \Gamma\{x \mapsto \sigma\} \end{array}}{C, \Gamma \vdash \textbf{letrec } x = e_1 \textbf{ in } e_2 \colon \tau}$$

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

Expressions (non-syntax-directed)

Gen
$$\frac{\begin{array}{l} C \wedge D, \Gamma \vdash e \colon \beta \\ \beta\bar{\alpha}\#\mathrm{FV}(\Gamma, C) \end{array}}{C \wedge \exists \beta\bar{\alpha}.D, \Gamma \vdash e \colon \forall \beta[\exists \bar{\alpha}.D].\beta}$$

Inst
$$\frac{\begin{array}{l} C, \Gamma \vdash e \colon \forall \bar{\alpha}[D].\tau' \\ C \vDash D[\bar{\alpha} := \bar{\tau}] \end{array}}{C, \Gamma \vdash e \colon \tau'[\bar{\alpha} := \bar{\tau}]}$$

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

Hide
$$\frac{\begin{array}{l} C, \Gamma \vdash e \colon \tau \\ \bar{\alpha}\#\mathrm{FV}(\Gamma, \tau) \end{array}}{\exists \bar{\alpha}.C, \Gamma \vdash e \colon \tau}$$

Equ
$$\frac{\begin{array}{l} C, \Gamma \vdash e \colon \tau \\ C \vDash \tau \doteq \tau' \end{array}}{C, \Gamma \vdash e \colon \tau'}$$

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

Existential Type System extension – ExIntro processing

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

The Type System – Var

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

$$[\![\Gamma \vdash x\colon \tau]\!] = \exists\beta'\bar{\alpha}'.D[\beta\bar{\alpha} := \beta'\bar{\alpha}'] \wedge \beta' \doteq \tau$$

where $\Gamma(x) = \forall\beta[\exists\bar{\alpha}.D].\beta$, $\beta'\bar{\alpha}'\#\mathrm{FV}(\Gamma, \tau)$

slide3.gadt:

```
datatype Tau
external x : ∀b[b = Tau → Tau].b = "x"

let var_rule = x
```

shell:

```
# invargent slide3.gadt -inform
val var_rule : Tau → Tau
```

$$\frac{\begin{array}{l} C, \Gamma \vdash e_1 : \tau' \to \tau \\ C, \Gamma \vdash e_2 : \tau' \end{array}}{C, \Gamma \vdash e_1\, e_2 : \tau}$$

$$[\![\Gamma \vdash e_1\, e_2 : \tau]\!] = \exists \alpha . [\![\Gamma \vdash e_1 : \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2 : \alpha]\!], \alpha \# \mathrm{FV}(\Gamma, \tau)$$

slide4.gadt:

```
datatype Tau
datatype Tau'
datacons E2 : Tau'
external e1 : Tau' → Tau = "e1"

let app_rule = e1 E2
```

shell:

```
# invargent slide4.gadt -inform
val app_rule : Tau
```

$$\frac{\forall i\, C, \Gamma \vdash c_i \colon \tau_1 \to \tau_2}{C, \Gamma \vdash \lambda(c_1 \ldots c_n) \colon \tau_1 \to \tau_2}, \text{ where } c_i = p_i.e_i$$

$$[\![\Gamma \vdash \lambda\bar{c} \colon \tau]\!] = \exists \alpha_1 \alpha_2 . [\![\Gamma \vdash \bar{c} \colon \alpha_1 \to \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 \doteq \tau, \alpha_1 \alpha_2 \# \mathrm{FV}(\Gamma, \tau)$$

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

$$[\![\Gamma \vdash p.e \colon \tau_1 \to \tau_2]\!] = [\![\vdash p \downarrow \tau_1]\!] \wedge \forall \bar{\beta}. D \Rightarrow [\![\Gamma\Gamma' \vdash e \colon \tau_2]\!]$$

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

slide5.gadt: or $C \vdash Kx \colon \tau \longrightarrow \exists \bar{\alpha}\bar{\beta}[D]\{x \mapsto \tau_1\}$

```
let abs_gen_rules = fun x -> x
```

shell:

```
# invargent slide5.gadt -inform
val abs_gen_rules : ∀a. a → a
```

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

$$[\![\Gamma \vdash K e_1 \ldots e_n : \tau]\!] = \exists \bar{\alpha}'\bar{\beta}'.(\wedge_i [\![\Gamma \vdash e_i : \tau_i[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}']]\!] \wedge D[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}'] \wedge \varepsilon(\bar{\alpha}') \doteq \tau)$$

Type `Num` could be defined as:

```
datatype Num : num   datacons 1 : Num 1   datacons 2 : Num 2   ...
```

slide6.gadt:

```
datatype Box : num
datacons Small : ∀n,k [n ⩽ 7∧ k = n+6]. Num n ⟶ Box k

let gift = Small 4
let package = fun x -> Small (x + -3)
```

shell:

```
# invargent slide6.gadt -inform
val gift : Box 10
val package : ∀n[n ⩽ 10]. Num n → Box (n + 3)
```

AVL Trees – `singleton`

Height of left sibling, `m`, differs by at most 2 from height of the right sibling, `n`.

Resulting height is `max(m,n)+1`. Height value is stored with the node.

slide7.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 singleton = fun x -> Node (Empty, x, Empty, 1)
```

shell:

```
# invargent slide7.gadt -inform
val singleton : ∀a. a → Avl (a, 1)
```

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

$$\begin{array}{rcl} [\![\Gamma\vdash p\,\textbf{when}\,\wedge_i m_i\leqslant n_i.e\colon\tau_1\rightarrow\tau_2]\!] & = & \exists\overline{\alpha_i^1\alpha_i^2}.[\![\vdash p\downarrow\tau_1]\!]\wedge\forall\bar{\beta}.D\Rightarrow\\ & & \quad\wedge_i[\![\Gamma\Gamma'\vdash m_i\colon\mathrm{Num}(\alpha_i^1)]\!]\wedge_i[\![\Gamma\Gamma'\vdash n_i\colon\mathrm{Num}(\alpha_i^2)]\!]\\ \text{when } e\neq\textbf{assert false}\wedge\ldots\wedge & & \quad\wedge(\wedge_i\alpha_i^1\leqslant\alpha_i^2\Rightarrow[\![\Gamma\Gamma'\vdash e\colon\tau_2]\!])\\ e\neq\lambda(p'\ldots\lambda(p''.\textbf{assert false})) & & \text{where } \exists\bar{\beta}[D]\Gamma' \text{ is } [\![\vdash p\uparrow\tau_1]\!],\ \bar{\beta}\overline{\alpha_i^1\alpha_i^2}\#\mathrm{FV}(\Gamma,\tau_1,\tau_2)\end{array}$$

slide8.gadt:

```
datatype Signed : num
datacons Pos : ∀n [0 ⩽ n]. Num n ⟶ Signed n
datacons Neg : ∀n [n ⩽ 0]. Num n ⟶ Signed n
let foo = function
  | i when 7 <= i -> Pos (i + -7)
  | i when i <= 7 -> Neg (i + -7)
```

Result: `val foo : ∀n. Num (n + 7) → Signed n` *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)
```

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

$$\begin{array}{rcl} [\![\Gamma\vdash p\,\textbf{when}\wedge_i m_i\leqslant n_i.e\colon\tau_1\rightarrow\tau_2]\!] & = & \exists\alpha_3\overline{\alpha_i^1\alpha_i^2}.[\![\vdash p\downarrow\alpha_3]\!]\wedge\forall\bar{\beta}.D\Rightarrow \\ & & \quad\wedge_i[\![\Gamma\Gamma'\vdash m_i\colon\mathrm{Num}(\alpha_i^1)]\!]\wedge_i[\![\Gamma\Gamma'\vdash n_i\colon\mathrm{Num}(\alpha_i^2)]\!] \\ \text{when } e=\textbf{assert false}\vee\ldots\vee & & \quad\wedge(\alpha_3\doteq\tau_1\wedge_i\alpha_i^1\leqslant\alpha_i^2\Rightarrow[\![\Gamma\Gamma'\vdash e\colon\tau_2]\!]) \\ e=\lambda(p'\ldots\lambda(p''.\textbf{assert false})) & & \\ & & \text{where } \exists\bar{\beta}[D]\Gamma' \text{ is } [\![\vdash p\uparrow\alpha_3]\!],\ \bar{\beta}\alpha_3\overline{\alpha_i^1\alpha_i^2}\#\mathrm{FV}(\Gamma,\tau_1,\tau_2)\end{array}$$

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

In the solver, we assume for negation, that the numerical domain is integers, while in general we take it to be rational numbers.

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
```

$$\frac{\begin{array}{ll} C,\Gamma' \vdash e_1 \colon \sigma & C,\Gamma' \vdash e_2 \colon \tau \\ \sigma = \forall\beta[\exists\bar{\alpha}.D].\beta & \Gamma' = \Gamma\{x \mapsto \sigma\} \end{array}}{C,\Gamma \vdash \mathbf{letrec}\, x = e_1 \,\mathbf{in}\, e_2 \colon \tau}$$

$$\begin{array}{rcl} [\![\Gamma \vdash \mathbf{letrec}\, x = e_1 \,\mathbf{in}\, e_2 \colon \tau]\!] & = & (\forall\beta(\chi(\beta) \Rightarrow [\![\Gamma\{x \mapsto \forall\beta[\chi(\beta)].\beta\} \vdash e_1 \colon \beta]\!])) \wedge \\ & & (\exists\alpha.\chi(\alpha)) \wedge [\![\Gamma\{x \mapsto \forall\beta[\chi(\beta)].\beta\} \vdash e_2 \colon \tau]\!] \\ & & \text{where } \beta\#\mathrm{FV}(\Gamma,\tau), \chi\#\mathrm{PV}(\Gamma) \end{array}$$

$\chi(\cdot)$ is a second order variable.

slide12.gadt:

```
datatype List : type * num
datacons LNil : ∀a. List (a, 0)
datacons LCons : ∀a, n [0≤n]. a * List (a, n) ⟶ List (a, n+1)

let rec map = fun f ->
  function LNil -> LNil
    | LCons (x, xs) -> LCons (f x, map f xs)
```

Result: `val map : ∀n, a, b. (a → b) → List (a, n) → List (b, n)` *binary_plus.gadt*:

```
datatype Binary : num
datatype Carry : num
datacons Zero : Binary 0
datacons PZero : ∀n [0≤n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n [0≤n]. Binary n ⟶ Binary(2 n + 1)
datacons CZero : Carry 0
datacons COne : Carry 1

let rec plus =
  function CZero ->
    (function
      | Zero ->
        (function Zero -> Zero [...]
    | COne ->
    (function Zero ->
        (function Zero -> POne(Zero)
          | PZero b1 -> POne b1
          | POne b1 -> PZero (plus COne Zero b1)) [...]
```

re. `plus: ∀i, k, n. Carry i → Binary k → Binary n → Binary (n + k + i)`

The Type System – actual `App`

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

$$[\![\Gamma \vdash e_1\, e_2\colon \tau]\!] = \exists\alpha.[\![\Gamma \vdash e_1\colon \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2\colon \alpha]\!] \wedge \not{E}(\alpha), \alpha \# \mathrm{FV}(\Gamma,\tau)$$

Above, $\not{E}(\tau')$ means that τ' is not an existential type. Therefore *slide14.gadt* fails:

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
datacons LCons : ∀n, a [0≤n]. a * List(a, n) ⟶ List(a, n+1)
let rec filter = fun f ->
  efunction LNil -> LNil
    | LCons (x, xs) ->
      ematch f x with
        | True -> LCons (x, filter f xs)
        | False -> filter f xs
```

shell: unfortunately, error not informative in current implementation

```
../invargent slide14.gadt -inform
File "slide14.gadt", line 5, characters 2-134:
No answer in type: term abduction failed
```

filter.gadt:

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
datacons LCons : ∀n, a [0⩽n]. a * List(a, n) ⟶ List(a, n+1)

let rec filter = fun f ->
  efunction LNil -> LNil
    | LCons (x, xs) ->
      ematch f x with
        | True ->
          let ys = filter f xs in
          LCons (x, ys)
        | False ->
          filter f xs
```

We use both `efunction` and `ematch` (a β-redex for `efunction`), because `function` and `match` would require the types of branch bodies to be equal: to be lists of the same length.

```
val filter :
  ∀n, a.
  (a → Bool) → List (a, n) → ∃k[0 ⩽ k ∧ k ⩽ n].List (a, k)
```

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

$$\begin{aligned}[\![\Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 \colon \tau]\!] &= \exists \alpha_0 . [\![\Gamma \vdash e_1 \colon \alpha_0]\!] \wedge \\ &\quad ([\![\Gamma \vdash p.e_2 \colon \alpha_0 \rightarrow \tau]\!] \wedge \not{E}(\alpha_0) \vee_{K \in \mathcal{E}} [\![\Gamma \vdash K p.e_2 \colon \alpha_0 \rightarrow \tau]\!]) \\ &\quad \text{where } \mathcal{E} = \{K | K :: \forall \bar{\alpha} \bar{\beta}[E].\tau \rightarrow \varepsilon_K(\bar{\alpha}) \in \Sigma\}\end{aligned}$$

OCaml code generated for *filter.gadt* – *filter.ml*:

```
type _ list =
  | LNil : (*∀'a.*) ('a (* 0 *)) list
  | LCons : (*∀'n, 'a[0 ⩽ n].*)'a * ('a (* n *)) list ->
    ('a (* n + 1 *)) list
type _ ex2 =
  | Ex2 : (*∀'k, 'n, 'a[0 ⩽ k ∧ k ⩽ n].*)('a (* k *)) list ->
    ((* n,*) 'a) ex2
let rec filter :
  type (*n*) a . (((a -> bool)) -> (a (* n *)) list -> ((* n,*) a) ex2) =
  (fun f ->
    (function LNil -> let xcase = LNil in Ex2 xcase
      | LCons (x, xs) ->
          (if f x then
          let Ex2 ys = filter f xs in let xcase = LCons (x, ys) in Ex2 xcase
          else let Ex2 xcase = filter f xs in Ex2 xcase)))
```

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`.

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
```

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)); ()
```

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 Handles Sensible Non-pointwise Cases

Chuan-kai Lin's system needs a workaround, InvarGenT works with def. settings – *non_pointwise_avl.gadt*:

```
(** Normally we would use [num], but this is a stress test for [type]. *)
datatype Z
datatype S : type
datatype Balance : type * type * type
datacons Less : ∀a. Balance (a, S a, S a)
datacons Same : ∀a. Balance (a, a, a)
datacons More : ∀a. Balance (S a, a, S a)
datatype AVL : type
datacons Leaf : AVL Z
datacons Node :
  ∀a, b, c. Balance (a, b, c) * AVL a * Int * AVL b ⟶ AVL (S c)

datatype Choice : type * type
datacons Left : ∀a, b. a ⟶ Choice (a, b)
datacons Right : ∀a, b. b ⟶ Choice (a, b)

let rotr = fun z d -> function
  | Leaf -> assert false
  | Node (Less, a, x, Leaf) -> assert false
  | Node (Same, a, x, (Node (_,_,_,_) as b)) ->
    Right (Node (Less, a, x, Node (More, b, z, d))) [...]
```

Result: `∀a.Int → AVL a → AVL (S (S a)) → Choice (AVL (S (S a)), AVL (S (S (S a))))`

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.

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

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

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.