GADTs for Invariants and Postconditions

InvarGenT: *GADTs*-based **alternative** to *refinement types* provides a constraint-based formulation of type inference and reconstruction of invariants and postconditions.

- Strongest GADTs inference yet.
- Outside GADTs, less expressive than *refinement types*, but many techniques should apply there.
- The same inference process for types and refinements, with unique strengths.

GADTs for Invariants and Postconditions

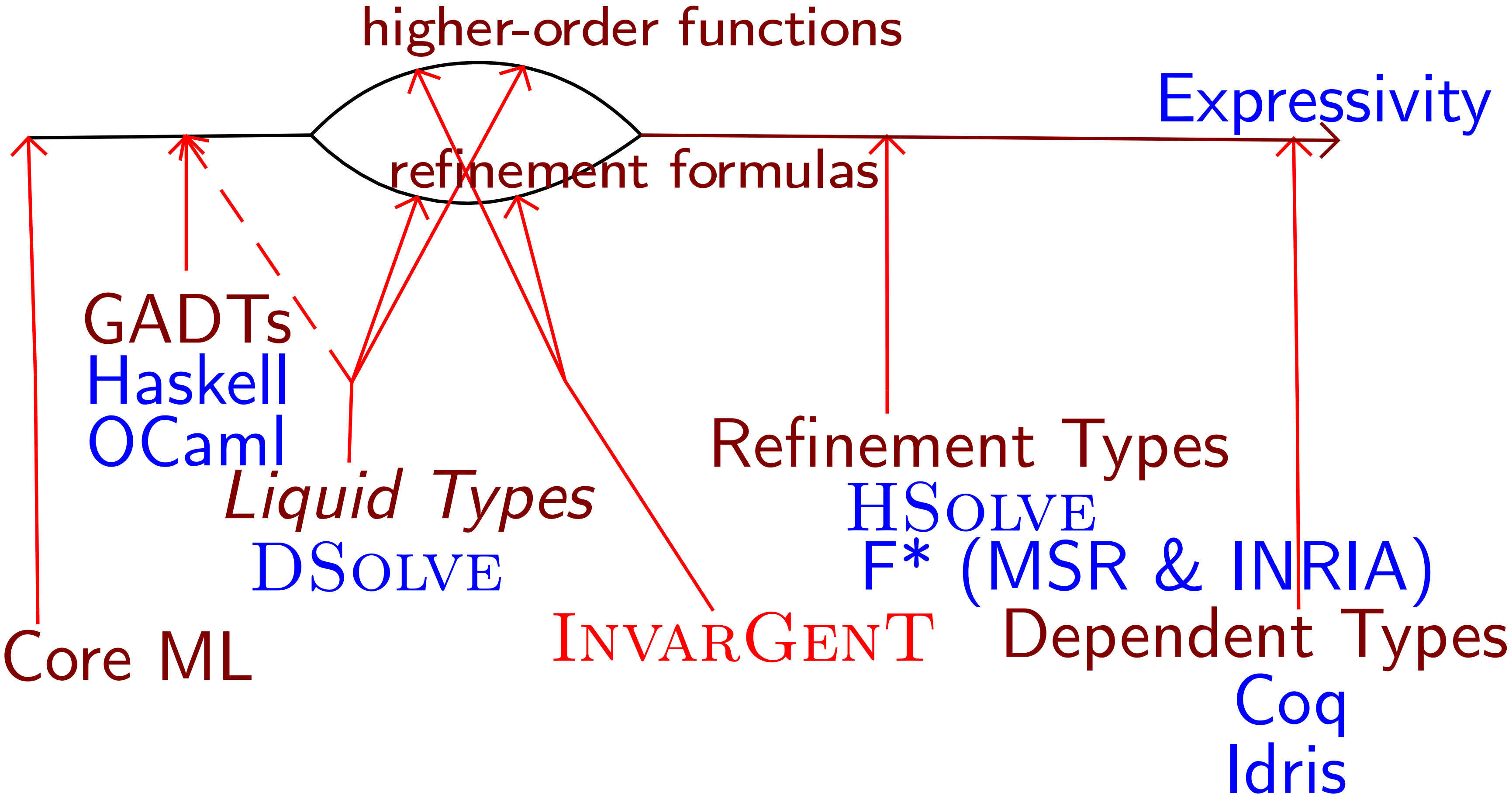
InvarGenT: *GADTs*-based **alternative** to *refinement types* provides a constraint-based formulation of type inference and reconstruction of invariants and postconditions.

- Instead of SMT solvers, uses:
  - **abduction** to infer maximally general types and maximally weak preconditions,
  - **generalization** to infer strongest postconditions (most specific existential types).

Why focus on full inference?

- Programmers waste time on simple mistakes during rapid prototyping.
- Fully automated specification generation for functions will help them overcome the quick-and-dirty mindset during rapid prototyping.
- Full automation speeds up the evolution of a specification.
- An IDE can paste a generated signature in the source code, perhaps to be further refined by the programmer.

Locating InvarGenT on the map



higher-order functions

Expressivity

refinement formulas

GADTs
Haskell
OCaml

Liquid Types
DSolve

Refinement Types
HSolve
F* (MSR & INRIA)

InvarGenT

Dependent Types
Coq
Idris

Core ML

Generalized Algebraic Data Types

```
datatype Term : type
datacons Lit : Int ⟶ Term Int
datacons Pair :
  ∀a, b. Term a * Term b ⟶ Term (a, b)
```

```
let rec eval = function

  | Lit i -> i

  | Pair (x, y) -> eval x, eval y
```

Generalized Algebraic Data Types

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

where $\Gamma(x) = \forall \beta [\exists \bar{\alpha}. D]. \beta$ renaming

$[\![\Gamma \vdash \mathbf{let\ rec}\ x = e : \tau]\!] =$

$(\forall \beta (\chi(\beta) \Rightarrow [\![\Gamma\{x \mapsto \forall \beta[\chi(\beta)].\beta\} \vdash e : \beta]\!])) \wedge \chi(\tau)$

$[\![\Gamma \vdash \lambda \overline{p_i.e_i} : \tau]\!] =$

$\exists \alpha_1 \alpha_2 . \alpha_1 \rightarrow \alpha_2 \doteq \tau \wedge_i [\![\Gamma \vdash p_i.e_i : \alpha_1 \rightarrow \alpha_2]\!]$

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

where $\exists \bar{\beta}[D]\Gamma'$ is $[\![p \uparrow \tau_1]\!]$

$[\![\vdash K x \uparrow \tau]\!] = \exists \bar{\alpha} \bar{\beta}$ – the specific case $p = K x$

$[\varepsilon(\bar{\alpha}) \doteq \tau \wedge D]\{x \mapsto \tau_1\}$ – modulo variable

where $K :: \forall \bar{\alpha} \bar{\beta}[D].\tau_1 \rightarrow \varepsilon(\bar{\alpha})$ renaming

Generalized Algebraic Data Types

$\forall\beta.\chi(\beta) \Rightarrow \exists\alpha_1, \alpha_2.\alpha_1 \to \alpha_2 \doteq \beta \wedge$ `let rec...function`

$\quad\exists\alpha_3.\mathrm{Term}(\alpha_3) \doteq \alpha_1 \wedge$ `| Lit i -> i`

$\quad(\forall\beta_1.\mathrm{Term}(\beta_1) \doteq \alpha_1 \wedge \mathrm{Int} \doteq \beta_1 \Rightarrow \mathrm{Int} \doteq \alpha_2) \wedge$

$\quad\exists\alpha_4.\mathrm{Term}(\alpha_4) \doteq \alpha_1 \wedge$ `| Pair (x, y) ->`

$\quad\forall\beta_2\beta_3\beta_4.\mathrm{Term}(\beta_4) \doteq \alpha_1 \wedge (\beta_2, \beta_3) \doteq \beta_4 \Rightarrow$

$\quad\exists\alpha_5\alpha_6.\alpha_2 \doteq (\alpha_5, \alpha_6) \wedge$ `eval x, eval y`

$\quad\exists\alpha_7\exists\alpha_8.\alpha_8 \doteq \alpha_7 \to \alpha_5 \wedge \chi(\alpha_8) \wedge \mathrm{Term}(\beta_2) \doteq \alpha_7 \wedge$

$\quad\exists\alpha_9\exists\alpha_0.\alpha_0 \doteq \alpha_9 \to \alpha_6 \wedge \chi(\alpha_0) \wedge \mathrm{Term}(\beta_3) \doteq \alpha_9 \wedge$

$\exists\alpha.\chi(\alpha)$ – Except for χ, same as *Pottier & Simonet*.

Generalized Algebraic Data Types

We normalize, remember quantifiers separately, and simplify a bit:

$$
\begin{aligned}
&(\top \Rightarrow \chi(\alpha)) \wedge \\
&(\chi(\beta) \Rightarrow \alpha_1 \doteq \mathrm{Term}(\alpha_4) \wedge \beta \doteq \mathrm{Term}(\alpha_4) \to \alpha_2) \wedge \\
&(\chi(\beta) \wedge \mathrm{Term}(\beta_1) \doteq \alpha_1 \wedge \mathrm{Int} \doteq \beta_1 \Rightarrow \alpha_2 \doteq \mathrm{Int}) \wedge \\
&(\chi(\beta) \wedge \mathrm{Term}(\beta_4) \doteq \alpha_1 \wedge (\beta_2, \beta_3) \doteq \beta_4 \Rightarrow \\
&\qquad \chi(\alpha_8) \wedge \chi(\alpha_0) \wedge \alpha_0 \doteq \mathrm{Term}(\beta_3) \to \alpha_6 \wedge \\
&\qquad \alpha_8 \doteq \mathrm{Term}(\beta_2) \to \alpha_5 \wedge \alpha_2 \doteq (\alpha_5, \alpha_6))
\end{aligned}
$$

InvarGenT's approach to Typeability

- The formulas are interpreted in a fixed model $\mathcal{M}$, in particular for any $\bar{\tau}$, $\bar{\tau}'$ and $\varepsilon_1 \neq \varepsilon_2$, $\mathcal{M} \vDash \varepsilon_1(\bar{\tau}) \doteq \varepsilon_2(\bar{\tau}') \Rightarrow \perp$ and $\mathcal{M} \vDash \varepsilon_1(\bar{\tau}) \doteq \varepsilon_1(\bar{\tau}') \Rightarrow \bar{\tau} \doteq \bar{\tau}'$.
- A *solved form formula* $\exists \bar{\alpha}.F$, is $\bar{\alpha} \subseteq \mathrm{FV}(F)$, and a conjunction of atoms F, where equations are a substitution: $x \doteq t_x \wedge y \doteq t_y \wedge \ldots \wedge n \leqslant m \wedge \ldots$
- An *interpretation of predicate variables* is, roughly speaking, $\mathcal{I} = \overline{\chi := \exists \bar{\alpha}_\chi . F_\chi}$ for solved form formulas $\exists \bar{\alpha}_\chi . F_\chi$. $\mathcal{I}, \mathcal{M} \vDash \Phi$ if and only if $\mathcal{M} \vDash \mathcal{I}(\Phi)$.

$\textsc{InvarGenT}$'s approach to Typeability

- Let $[\![\Gamma \vdash e : \tau]\!] \Leftrightarrow \mathcal{Q}.\Phi_N$ where $\Phi_N = \wedge_i(D_i \Rightarrow C_i)$. Solved form formulas $\exists\bar{\alpha}_{\mathrm{res}}.F_{\mathrm{res}}$, $\mathcal{I}$ are a *solution to the type inference problem* $[\![\Gamma \vdash e : \tau]\!]$ when: $\mathcal{I}$, $\mathcal{M} \vDash F_{\mathrm{res}} \Rightarrow \Phi_N$, $\mathcal{M} \vDash \mathcal{Q}.F_{\mathrm{res}}[\bar{\alpha}_{\mathrm{res}} := \bar{t}]$ for some $\bar{t}$, and for every implication in Φ_N, if $\mathcal{M}, \mathcal{I} \vDash \exists \mathrm{FV}(D_i).D_i$ then $\mathcal{M}, \mathcal{I} \vDash \exists \mathrm{FV}(D_i, F_{\mathrm{res}}).D_i \wedge F_{\mathrm{res}}$.
    - The last condition excludes bogus solutions like $\alpha_2 \doteq \mathrm{Int}$ in the earlier example, which a satisfiability solver could return for $[\![\Gamma \vdash e : \tau]\!]$!

Joint Constraint Abduction

- Not incidentally, the inference technique we need has been introduced independently: abduction!
- Abduction is inference to the best explanation: for a problem $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$ over $\mathcal{M}$, the *abduction answer* A defined by the following conditions explains the observation C_i given the context D_i (and background knowledge $\mathcal{M}$):
  - relevance: $\mathcal{M} \vDash A \wedge D_i \Rightarrow C_i$ for all i,
  - consistency: $\mathcal{M} \vDash \exists \mathrm{FV}(A, D_i).A \wedge D_i$ for all i,
  - validity: $\mathcal{M} \vDash \mathcal{Q}.A[\bar{\alpha} := \bar{t}]$ for some $\bar{t}$.

Joint Constraint Abduction

Continuing the example, we start with $\chi(\cdot) = \top$:

$$\alpha_1 \doteq \mathrm{Term}(\alpha_4) \quad \beta \doteq \mathrm{Term}(\alpha_4) \rightarrow \alpha_2$$

$$(\top \Rightarrow \alpha_1 \doteq \mathrm{Term}(\alpha_4) \wedge \beta \doteq \mathrm{Term}(\alpha_4) \rightarrow \alpha_2) \wedge$$

$$\alpha_2 \doteq \alpha_4$$

$$(\mathrm{Term}(\beta_1) \doteq \alpha_1 \wedge \mathrm{Int} \doteq \beta_1 \Rightarrow \alpha_2 \doteq \mathrm{Int}) \wedge$$

$$(\mathrm{Term}(\beta_4) \doteq \alpha_1 \wedge (\beta_2, \beta_3) \doteq \beta_4 \Rightarrow$$

$$\alpha_0 \doteq \mathrm{Term}(\alpha_6)$$

$$\alpha_0 \doteq \mathrm{Term}(\beta_3) \rightarrow \alpha_6 \wedge$$

$$\alpha_8 \doteq \mathrm{Term}(\alpha_5) \qquad \alpha_2 \doteq \alpha_4$$

$$\alpha_8 \doteq \mathrm{Term}(\beta_2) \rightarrow \alpha_5 \wedge \alpha_2 \doteq (\alpha_5, \alpha_6))$$

Joint Constraint Abduction

Finally: $\chi(\beta) = \exists \beta_0 . \beta \doteq \mathrm{Term}(\beta_0) \rightarrow \beta_0$

and $F_{\mathrm{res}} = \alpha_1 \doteq \mathrm{Term}(\beta_0) \wedge \alpha_2 \doteq \beta_0 \wedge \ldots$

Polymorphic Recursion

- We solve for the types and invariants (i.e. type schemes) of recursive functions – and ensure the correctness of solutions – by iteration.
- On the last iteration, the whole abduction answer is contained in F_{res} – no update to the invariants $\chi(\cdot)$ and thus a fixpoint to the iteration.

Existential Types and Postconditions

- Lists with length:

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

Existential Types and Postconditions

- Often exact types are too tight:

```
let rec filter f =
  function LNil -> LNil
    | LCons (x, xs) ->
      if f x then
          LCons (x, filter f xs)
      else filter f xs
```

"No answer in num: numerical abduction failed"

Existential Types and Postconditions

- Explicitly introducing existential types to capture postconditions:

```
let rec filter f =
  efunction LNil -> LNil
    | LCons (x, xs) ->
      eif f x then
          let ys = filter f xs in
          LCons (x, ys)
      else filter f xs
```

Existential Types and Postconditions

- We get:

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

- We do not allow existential types for function arguments – the values need to be `let`-bound before use.

Abduction for Linear Inequalities

To find the abduction answers to $d \Rightarrow c$ for two linear inequalities d, c, pick a common variable $\alpha \in \mathrm{FV}(d) \cap \mathrm{FV}(c)$ or the constant $\alpha = 1$. 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 $c_\alpha \leqslant d_\alpha$.

Constraint Generalization

- Our postconditions are the strongest conditions G_{def} that can be derived from the contexts of all cases of a definition of an existential type $\varepsilon_{\mathrm{def}}$ introduced by `efunction`, `ematch`, `eif`.

  $$\mathcal{M} \vDash \mathcal{I}_k(D_i) \Rightarrow G_{\mathrm{def}} \text{ for all } i \text{ defining } \varepsilon_{\mathrm{def}}$$

  where $\mathcal{I}_k$ is the solution of both invariants-preconditions and postconditions from the previous iteration.

Constraint Generalization

- We call this algorithmic task, symbolically $\bigvee_{i \in \mathrm{def}} \mathcal{I}_k(D_i)$, *constraint generalization*. It is simpler than abduction.
    - We use modified *anti-unification* algorithm to find existential type shapes (i.e. for generalization in the term domain),
    - and simplified *generalized convex hull* algorithm to find numerical postconditions (i.e. for generalization in the numerical domain).

Finding Invariants and Postconditions

1. Start with trivial preconditions (no properties).
    - Too weak, will get strengthened.
2. Use constraint generalization to find **strongest postconditions** – on initial iterations, from base cases only.
    - May get weakened once all cases considered.
3. Use joint constraint abduction to update **maximally weak preconditions**.
4. Go to (2) if either invariants or postconditions change.

InvarGenT vs. Pointwise GADTs

- Examples from Chuan-kai Lin's PhD thesis within the scope of his algorithm:

`rotate: ∀a.Dir→Int→RoB (Black, a)→ Dir→Int→ RoB (Black, a)→RoB (Red,a) →RoB (Black, S a)`	0.02s
`zip2: ∀a, b.Zip2 (B, a)→b→a`	0.16s
`rotl: ∀a.AVL a→Int→AVL (S (S a))→ Choice (AVL (S (S a)), AVL (S(S(S a))))`	0.03s
`ins: ∀a.Int→AVL a→ Choice (AVL a, AVL (S a))`	0.41s
`extract: ∀a, b.Path b→Tree (b, a)→a`	0.06s
`run_state: ∀a, b.b→State (b, a)→(b, a)`	0.01s
`head: ∀a, b.List (a, S b)→a`	∞

INVARGENT vs. Pointwise GADTs

- and outside the scope of Chuan-kai Lin algorithm:

`joint: ∀a.Split (a, a)→a`	<0.01s
`rotr: ∀a.Int→AVL a→AVL (S (S a))→ Choice (AVL (S (S a)), AVL (S(S(S a))))`	0.09s
`delmin: ∀a.AVL (S a)→ (Int, Choice (AVL a, AVL (S a)))`	0.31s
`fd_comp: ∀a, b, c.FunDesc (c, b)→ FunDesc (b, a)→ FunDesc (c, a)`	0.2s*, 0.1s*
`zip1: ∀a, b.Zip1 (List b, a)→b→a`	0.08s
`leq: ∀a.Nat a→NatLeq (a, a)`	<0.01s**
`run_state: ∀a, b.b→State (b, a)→(b, a)`	0.03s

* Slight meaning-preserving modification

** Needs a non-default option `-prefer_guess`

InvarGenT vs. DSolve (*Liquid Types*)

Program	InvarGenT	DSolve
`dotprod`	0.05s	0.31s
`bcopy`	0.03s	0.15s
`bsearch`	0.07s	0.46s
`queen`	0.42s	0.7s
`isort`	0.3s, 0.37s	0.88s
`tower` no assertions	0.84s	∞
`tower` with assertion	3.93s	3.33s
`matmult`	0.34s	1.79s
`heapsort`	2.34s	0.53s
`fft` no assertions	36.4s*	?
`fft` with assertion	37.5s*,**	9.13s
`simplex`	8.1s*, 31.4s	7.73s
`gauss` no assertions	2.66s*, 1.02s*,***	?
`gauss` with assertion	2.72s	3.17s

Options: ** `-same_with_assertions` *** `-prefer_bound_to_local`

InvarGenT Original Examples

- Lists with length:

`head: ∀n,a[1⩽n].List (a, n)→a`	<0.01s
`append: ∀a,n,k.List (a, n)→List (a, k)→ List(a, n + k)`	0.02s
`flatten_pairs: ∀n, a. List ((a, a), n) → List (a, 2 n)`	0.01s
`flatten_quadrs: ∀n, a. List ((a,a,a,a), n) → List (a, 4 n)`	0.06s
`filter: ∀n, a. (a→Bool) → List (a, n) → ∃k[0 ⩽ k ∧ k ⩽ n].List (a, k)`	0.18s
`zip: ∀a,b,n,k.(List (a,n), List (b,k))→ ∃i[i=min(n, k)].List ((a,b),i)`	0.38s

InvarGenT Original Examples

- Binary numbers:

`plus: ∀n,k,i.Carry i→Binary k→Binary n→Binary (n+k+i)`	0.66s
`increment: ∀n.Binary n→Binary (n + 1)`	0.01s
`bitwise_or: ∀k, n.Binary k→Binary n → ∃i[k ⩽ i ∧ n ⩽ i ∧ i ⩽ n + k].Binary i`	1.21s

- AVL trees with imbalance of 2: [next slide]

create: ∀k, n, a[0⩽n ∧ 0⩽k ∧ n⩽k+2 ∧ k⩽n+2]. Avl (a, k) → a → Avl (a, n) → ∃i[i=max (k + 1, n + 1)].Avl (a, i)	0.09s
rotr: ∀k, n, a[0⩽n ∧ n+2⩽k ∧ k⩽n+3]. Avl (a, k) → a → Avl (a, n) → ∃n[k ⩽ n ∧ n ⩽ k + 1].Avl (a, n)	1.07s
add: ∀n, a. a → Avl (a, n) → ∃k[1⩽k ∧ n⩽k ∧ k⩽n+1].Avl (a, k)	0.69s
remove_min_binding: ∀n, a[1⩽n].Avl (a, n) → ∃k[n⩽k+1 ∧ k⩽n ∧ k+2⩽2 n].Avl (a, k)	0.59s
merge: ∀k, n, a[n⩽k+2 ∧ k⩽n+2]. (Avl (a, n), Avl (a, k)) → ∃i[n⩽i ∧ k⩽i ∧ i⩽n+k ∧ i⩽max (k+1, n+1)].Avl (a, i)	0.93s
remove: ∀n, a. a → Avl (a, n) → ∃k[n⩽k+1 ∧ 0⩽k ∧ k⩽n].Avl (a, k)	0.38s
Total time for add, remove and helper functions:	4.92s

Response to prof. Sulzmann

- **Connection to earlier work**. Use and development of abduction in a GADTs framework, is indeed the main contribution, but also important is stressing the need for both maximally weak preconditions and strongest postconditions.
- **Problem statement**. The thesis attempts to develop fully automated derivation of specifications in the context of functional programming. The thesis statement is that this goal can be productively stated and achieved as type inference in a GADTs-based type system.

Response to prof. Szubert

- **Location of proofs**. Both the proofs and experiments with the test cases contributed to the development of the thesis. The contribution of the proofs is cashed out in the type system and the algorithmic components described in the main text.

Response to prof. Szubert

- **Obscure presentation of semantics**. The operational semantics of the mini-language complies with the standard call-by-value variant of semantics used by statically typed functional programming languages with pattern matching. It is presented in Section 2.2.1. Section 3.6 merely aims to demonstrate that the calculus does not deviate from this intuitive semantics.

Response to prof. Szubert

- **The role of specifications**. The comparative advantage of the thesis lies in the ability of InvarGenT to automatically generate specifications for cases which are beyond the capability of other systems. Multiple research groups are working on facilitating formal verification from specifications provided upfront.