Constraints vs Structures in Type Systems

Hemant Sai Gouni (& Will Crichton + Jonathan Aldrich)

February 18, 2025

There's a tug-of-war between constraints and structures at the heart of lots of type systems

There's a tug-of-war between constraints and structures at the heart of lots of type systems

I'd like to convince you that...

There's a tug-of-war between constraints and structures at the heart of lots of type systems

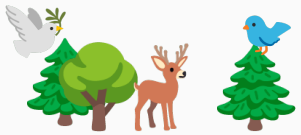
I'd like to convince you that...

1. it exists

There's a tug-of-war between constraints and structures at the heart of lots of type systems

I'd like to convince you that...

1. it exists
2. it's cognitively interesting

Chapter 1: Skirmishes in the Wild 

🫷 Round 1: Rust

Read `'l1: 'l2` as "`'l1` outlives `'l2`."

```
fn first<'a, 'b, 'c, T>(x: &'a T, y: &'b T) ->
&'c T where 'a: 'c {
    let z = &*x; // z : &'d T, implicitly
    z
}
```

🫷 Round 1: Rust

Read `'l1: 'l2` as "`'l1` outlives `'l2`."

```
fn first<'a, 'b, 'c, T>(x: &'a T, y: &'b T) ->
&'c T where 'a: 'c {
    let z = &*x; // z : &'d T, implicitly
    z
}
```

Read `&{x} T` as "a reference to `x` lives inside `T`."

```
fn first<T>(x: &T, y: &T) -> &{x} T {
    let z = &*x; // z : &{x} T here
    z
}
```

🫷 Round 1: Rust

Read `'l1: 'l2` as "`'l1` outlives `'l2`."

```
fn first<'a, 'b, 'c, T>(x: &'a T, y: &'b T) ->
&'c T where 'a: 'c {
    let z = &*x; // z : &'d T, implicitly
    z
}
```

Read `&{x} T` as "a reference to `x` lives inside `T`."

```
fn first<T>(x: &T, y: &T) -> &{x} T {
    let z = &*x; // z : &{x} T here
    z
}
```

👐 Round 1: Rust

Read `'l1: 'l2` as "`'l1` outlives `'l2`."

```
fn first<'a, 'b, 'c, T>(x: &'a T, y: &'b T) ->
& T where 'a: 'c {
    let z = &*x; // z : &'d T, implicitly
    z
}
```

Read `&{x} T` as "a reference to `x` lives inside `T`."

```
fn first<T>(x: &T, y: &T) -> &{x} T {
    let z = &*x; // z : &{x} T here
    z
}
```

👐 Round 2: Information Flow

Read `!l1 < !l2` as ‘information from `!l1` flows into `!l2`.’

```
let p : !password string = "katya"
let check : 'a string -> 'b bool
    with !password < 'b and 'a < 'b =
    (* returning a bool with info from password *)
    function attempt -> p == attempt
```

👐 Round 2: Information Flow

Read `!l1 < !l2` as 'information from `!l1` flows into `!l2`.'

```
let p : !password string = "katya"
let check : 'a string -> 'b bool
    with !password < 'b and 'a < 'b =
    (* returning a bool with info from password *)
    function attempt -> p == attempt
```

Read `[ l1 l2 ] string` as 'this `string` data depends on information from `l1` and `l2`.'

```
let p : [ !password ] string = "katya"
let check : [ 'a ] string -> [ 'a !password ] bool =
    (* returning a bool with info from password *)
    function attempt -> p == attempt
```

👐 Round 2: Information Flow

Read `!l1 < !l2` as 'information from `!l1` flows into `!l2`.'

```
let p : !password string = "katya"
let check : 'a string -> 'b bool
    with !password < 'b and 'a < 'b =
    (* returning a bool with info from password *)
    function attempt -> p == attempt
```

Read `[ l1 l2 ] string` as 'this `string` data depends on information from `l1` and `l2`.'

```
let p : [ !password ] string = "katya"
let check : [ 'a ] string -> [ 'a !password ] bool =
    (* returning a bool with info from password *)
    function attempt -> p == attempt
```

👐 Round 2: Information Flow

```
let inv : !alice int -> !alice int =
    fun x -> -x
let suc : !bob int -> !bob int =
    fun x -> x + 1


val inv_suc : 'a int -> 'b int * 'c int
    with !bob < 'c
    and !alice < 'b
    and 'a < 'b, 'c
    and 'a < !alice, !bob
let inv_suc x = (inv x, suc x)
```

👐 Round 2: Information Flow

```
let inv : !alice int -> !alice int =
    fun x -> -x
let suc : !bob int -> !bob int =
    fun x -> x + 1


val inv_suc : 'a int -> 'b int * 'c int
    with !bob < 'c
    and !alice < 'b
    and 'a < 'b, 'c
    and 'a < !alice, !bob
let inv_suc x = (inv x, suc x)
```

🫷🫸 Round 2: Information Flow

```
let inv : [ alice ] int -> [ alice ] int =
    fun x -> -x
let suc : [ bob ] int -> [ bob ] int =
    fun x -> x + 1


val inv_suc : [ ] int -> [ alice ] int * [ bob ]
int let inv_suc x = (inv x, suc x)
```

👐 Round 3: Subtyping

Read `'b <: 'a` as ‘`'b` is a subtype of `'a`.’

```
val twice : 'a -> ('a -> 'b) -> 'b where 'b <: 'a
let twice = fun x -> fun f -> f (f x)
```

Read `'b ∨ 'a` as ‘this type is either `'b` or `'a`.’

```
val twice : 'a -> ('b ∨ 'a -> 'b) -> 'b
let twice = fun x -> fun f -> f (f x)
```

👐 Round 3: Subtyping

Read `'b <: 'a` as ‘`'b` is a subtype of `'a`.’

```
val choose : bool -> 'a -> 'c
    where 'a <: 'c and bool <: 'c
let choose = fun b -> fun x ->
    if b then x else false
```

Read `'b ∨ 'a` as ‘this type is either `'b` or `'a`.’

```
val choose : bool -> 'a -> 'a ∨ bool
let choose = fun b -> fun x ->
    if b then x else false
```

Chapter 2: A Smoldering Debate 🔥

Generality 🌌 vs Specificity 🌠

Generality 🌌 vs Specificity 🌠

Few constructs that can capture *many* problems *adequately*

Generality 🌌 vs Specificity 🌠

Few constructs that can capture *many* problems *adequately*

Many constructs that capture *specific* problems *extraordinarily well*

Generality 🌌 vs Specificity 🌠

Few constructs that can capture *many* problems *adequately*

✅ Pro: Less disparate pieces to keep in memory, better compositional reasoning

Many constructs that capture *specific* problems *extraordinarily well*

Generality 🌌 vs Specificity 🌠

Few constructs that can capture *many* problems *adequately*

- ✅ Pro: Less disparate pieces to keep in memory, better compositional reasoning
- ❌ Con: More difficult to tell what any particular program fragment does at-a-glance.

Many constructs that capture *specific* problems *extraordinarily well*

Generality 🌌 vs Specificity 🌠

Few constructs that can capture *many* problems *adequately*

- ✅ Pro: Less disparate pieces to keep in memory, better compositional reasoning
- ❌ Con: More difficult to tell what any particular program fragment does at-a-glance.

Many constructs that capture *specific* problems *extraordinarily well*

- ✅ Pro: Closer to mental model of programming problem

Generality 🌌 vs Specificity 🌠

Few constructs that can capture *many* problems *adequately*

- ✅ Pro: Less disparate pieces to keep in memory, better compositional reasoning
- ❌ Con: More difficult to tell what any particular program fragment does at-a-glance.

Many constructs that capture *specific* problems *extraordinarily well*

- ✅ Pro: Closer to mental model of programming problem
- ❌ Con: More difficult to identify and transfer intuition between problems

Generality 🌌 vs Specificity 🌠

Generality 🌌 vs Specificity 🌠

`'a: 'c` `{x}`

Generality 🌌 vs Specificity 🌠

```
'a: 'c
```

```
{x}
```

```
!password < 'b, 'a < 'b
```

```
[ 'a !password ]
```

Generality 🌌 vs Specificity 🌠

`'a: 'c`	`{x}`
`!password < 'b, 'a < 'b`	`[ 'a !password ]`
`a <: c, b <: c`	`a ∨ b`

Generality 🌌 vs Specificity 🌠

```
'a: 'c
```

```
!password < 'b, 'a < 'b
```

```
a <: c, b <: c
```

Inequalities

```
{x}
```

```
[ 'a !password ]
```

```
a ∨ b
```

Domain-Dependent

Implicitness 🤯 vs Explicitness 🤬

Implicitness 🤯 vs Explicitness 🤬

Structure of problem is *hidden*
from view, hidden parts
automated away

Implicitness 🤯 vs Explicitness 🤬

Structure of problem is *hidden* from view, hidden parts *automated* away

Structure of problem *reified* in programming interface, *explicit intervention* required

Implicitness 🤯 vs Explicitness 🤬

Structure of problem is *hidden* from view, hidden parts *automated* away

Structure of problem *reified* in programming interface, *explicit intervention* required

✅ Pro: Don't have to think about type information where not needed for problem at hand

Implicitness 🤯 vs Explicitness 🤬

Structure of problem is *hidden* from view, hidden parts *automated* away

Structure of problem *reified* in programming interface, *explicit intervention* required

- ✅ Pro: Don't have to think about type information where not needed for problem at hand
- ❌ Con: Might have to keep track of implicit invariants in working memory

Implicitness 🤯 vs Explicitness 🤬

Structure of problem is *hidden* from view, hidden parts *automated* away

- ✅ Pro: Don't have to think about type information where not needed for problem at hand
- ❌ Con: Might have to keep track of implicit invariants in working memory

Structure of problem *reified* in programming interface, *explicit intervention* required

- ✅ Pro: Invariants fully represented in source, don't need to mentally recompute

Implicitness 🤯 vs Explicitness 🤬

Structure of problem is *hidden* from view, hidden parts *automated* away

- ✅ Pro: Don't have to think about type information where not needed for problem at hand
- ❌ Con: Might have to keep track of implicit invariants in working memory

Structure of problem *reified* in programming interface, *explicit intervention* required

- ✅ Pro: Invariants fully represented in source, don't need to mentally recompute
- ❌ Con: Can't easily forget unimportant type structure, more 'fluff' to read and discard

Implicitness 🤯 vs Explicitness 🤬

```
fn id<'a>(x: &'a T) -> &'a T
```

Implicitness 🤯 vs Explicitness 🤬

```
fn id<'a>(x: &'a T) -> &'a T
```

```
fn id(x: &T) -> &T
```

Implicitness 🤯 vs Explicitness 🤬

```
fn id<'a>(x: &'a T) -> &'a T
```

```
fn id(x: &T) -> &T
```

```
val both : 'a int -> 'b int -> 'c int
    where 'a < 'c and 'b < 'c
```

Implicitness 🌞 vs Explicitness 😡

```
fn id<'a>(x: &'a T) -> &'a T
```

```
fn id(x: &T) -> &T
```

```
val both : 'a int -> 'b int -> 'c int
    where 'a < 'c and 'b < 'c
```

```
val both : 'a int -> 'a int -> 'a int
```

Constraints vs Structures Lies at a Crossroads

Constraints vs Structures Lies at a Crossroads

Goal: Show that structures offer a better pathway to usability for complex type system features than constraints.

Constraints vs Structures Lies at a Crossroads

Goal: Show that structures offer a better pathway to usability for complex type system features than constraints.

- The constraint approach encodes information about the program via systems of inequalities (constraints)

Constraints vs Structures Lies at a Crossroads

Goal: Show that structures offer a better pathway to usability for complex type system features than constraints.

- The constraint approach encodes information about the program via systems of inequalities (constraints)

constraints = general + implicit

Constraints vs Structures Lies at a Crossroads

Goal: Show that **structures** offer a better pathway to usability for complex type system features than **constraints**.

- The constraint approach encodes information about the program via systems of inequalities (constraints)

  constraints $=$ **general** $+$ **implicit**

- The structural approach reifies these properties directly in the syntax via first-class algebraic structures

Constraints vs Structures Lies at a Crossroads

Goal: Show that structures offer a better pathway to usability for complex type system features than constraints.

- The constraint approach encodes information about the program via systems of inequalities (constraints)

  constraints = general + implicit

- The structural approach reifies these properties directly in the syntax via first-class algebraic structures

  structures = specific + explicit

Constraints vs Structures Lies at a Crossroads

Goal: Show that **structures** offer a better pathway to usability for complex type system features than **constraints**.

- The constraint approach encodes information about the program via systems of inequalities (constraints)

  constraints = **general** + **implicit**

- The structural approach reifies these properties directly in the syntax via first-class algebraic structures

  structures = **specific** + **explicit**

Note: *Not* a rigorous or prescriptive definition— still descriptive at this point!

Chapter 3: Poking and Prodding 

Experiments

Hypothesis: The structural approach will be easier to use...

Experiments

Hypothesis: The structural approach will be easier to use...

Syntactically

For tasks that extensively leverage/require the additional type information.

Experiments

Hypothesis: The structural approach will be easier to use...

Syntactically

For tasks that extensively leverage/require the additional type information.

- Structural approach simplifies the presentation of types.
- Constraint approach allows you to separate out and defer more complex type information

Experiments

Hypothesis: The structural approach will be easier to use...

Syntactically

For tasks that extensively leverage/require the additional type information.

Semantically

For tasks concerned with *distinguished elements* in the problem domain.

- Structural approach simplifies the presentation of types.
- Constraint approach allows you to separate out and defer more complex type information

Experiments

Hypothesis: The structural approach will be easier to use...

Syntactically

For tasks that extensively leverage/require the additional type information.

- Structural approach simplifies the presentation of types.
- Constraint approach allows you to separate out and defer more complex type information

Semantically

For tasks concerned with *distinguished elements* in the problem domain.

- `a <: c, b <: c`
  `a ∨ b`
- `'a -> 'b`
  `[ 'a ] -> [ ]`
- Making *internally represented* information *external* (explicit).

Experiments

Focusing on Rust and information flow. Why?

Experiments

Focusing on Rust and information flow. Why?

- Easier to come up with isomorphic examples.

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
  - ❤️ Good: isomorphic examples between constraints and structures *within the same problem domain*.

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
  - ❤️ Good: isomorphic examples between constraints and structures *within the same problem domain*.
  - ❤️‍🔥 Better: isomorphic examples *across problem domains*—between aliasing and information flow here.

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
  - ❤️ Good: isomorphic examples between constraints and structures *within the same problem domain*.
  - ❤️‍🔥 Better: isomorphic examples *across problem domains*—between aliasing and information flow here.
- Better **generalizability**.

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
  - ❤️ Good: isomorphic examples between constraints and structures *within the same problem domain*.
  - ❤️‍🔥 Better: isomorphic examples *across problem domains*—between aliasing and information flow here.
- Better **generalizability**.
  - Want to come up with design principles for type systems *beyond* Rust or information flow.

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments:

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments:

🌱 Natural Typing

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments:

🌱 Natural Typing

🧪 Synthesizing Types

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments:

🌱 Natural Typing

🧪 Synthesizing Types

🔪🍖 Program Slicing

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments:

- 🌱 Natural Typing
- 🧪 Synthesizing Types
- 🔪🍖 Program Slicing
- 👾 Attack/Defense Game

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments:

- 🌱 Natural Typing
- 🧪 Synthesizing Types
- 🔪🍖 Program Slicing
- 👾 Attack/Defense Game
- 👨‍💻 Refactor With Me

Experiments

Focusing on **Rust** and **information flow**. Why?

- Easier to come up with **isomorphic examples**.
- Better **generalizability**.

Experiments:

- 🌱 Natural Typing
- 🧪 **Synthesizing Types**
- 🔪🍖 Program Slicing
- 👾 Attack/Defense Game
- 👩‍💻 Refactor With Me

Synthesizing Types

Question

What is the influence of the constraints-structures distinction on how people **induct mental models of type systems** and **apply them to examples**?

Synthesizing Types

Question

What is the influence of the constraints-structures distinction on how people **induct mental models of type systems** and **apply them to examples**?

```
val send : 'a bool -> 'b bool
```

Synthesizing Types

Question

What is the influence of the constraints-structures distinction on how people **induct mental models of type systems** and **apply them to examples**?

```
val send : 'a bool -> 'b bool
```

```
val send : [ 'a ] bool -> [ ] bool
```

Synthesizing Types

Question

What is the influence of the constraints-structures distinction on how people **induct mental models of type systems** and **apply them to examples**?

```
val send : 'a bool -> 'b bool

val send : [ 'a ] bool -> [ ] bool

let result : ? bool = send (check p)
```

Synthesizing Types

```
val send : 'a bool -> 'b bool

val send : [ 'a ] bool -> [ ] bool

let result : ? bool = send (check p)
```

Synthesizing Types

```
val send : 'a bool -> 'b bool
```

```
val send : [ 'a ] bool -> [ ] bool
```

```
let result : ? bool = send (check p)
```

Question: what's the type of `result`?

Synthesizing Types

```
val send : 'a bool -> 'b bool
```

```
val send : [ 'a ] bool -> [ ] bool
```

```
let result : ? bool = send (check p)
```

Question: what's the type of `result`?

- `?` = `'c`

Synthesizing Types

```
val send : 'a bool -> 'b bool
```

```
val send : [ 'a ] bool -> [ ] bool
```

```
let result : ? bool = send (check p)
```

Question: what's the type of `result`?

- `?` = `'c` 😵

Synthesizing Types

```
val send : 'a bool -> 'b bool
```

```
val send : [ 'a ] bool -> [ ] bool
```

```
let result : ? bool = send (check p)
```

Question: what's the type of `result`?

- `?` = `'c` 😵‍💫
- `?` = `[ ]`

Synthesizing Types

```
val send : 'a bool -> 'b bool
```

```
val send : [ 'a ] bool -> [ ] bool
```

```
let result : ? bool = send (check p)
```

Question: what's the type of `result`?

- `?` = `'c` 😵‍💫
- `?` = `[ ]`

Goal: figure out fuzzy semi-formal *mental models* of type systems that model *misconceptions*.

Chapter 4: A Cliffhanger 🧗

The constraints-structures distinction hints at a rich framework for analyzing the usability of programming languages.

We want to derive general, far-sighted design principles for type systems that place human interface concerns beside mathematical considerations at the foundations of programming languages.

hsgouni@cs.cmu.edu / hgouni@hci.social