Semantic Types

Joshua Chen

Computational Logic
Department of Computer Science
University of Innsbruck

Computational Logic Masters Seminar

15 May 2019

Types, generally

In this talk, we'll be concerned with logical and mathematical specification, formalization, and verification.

Types, generally

In this talk, we'll be concerned with logical and mathematical specification, formalization, and verification.

Types...

- distinguish logical classes of objects,
- impose restrictions on formulas,
- help structure specifications and proofs.

Types in type theory

Dependent type theories used in major proof assistants are theoretically elegant, logically expressive, and functionally powerful.

Even "simple" type theory (HOL) has had a lot of success (and some advantages over DTT).

“Rigid” vs. “soft” types

Types as treated by type theory—“rigid” types:

- baked into the logical foundation,
- with accompanying syntactic features.

“Rigid” vs. “soft” types

Types as treated by type theory—“rigid” types:

- baked into the logical foundation,
- with accompanying syntactic features.

“Soft” types:

“Rigid” vs. “soft” types

Types as treated by type theory—“rigid” types:

- baked into the logical foundation,
- with accompanying syntactic features.

“Soft” types:

- *not* part of the underlying logical formalism,

“Rigid” vs. “soft” types

Types as treated by type theory—“rigid” types:

- baked into the logical foundation,
- with accompanying syntactic features.

“Soft” types:

- *not* part of the underlying logical formalism,
- not restricted to *any* particular logical system—can be used generically,

"Rigid" vs. "soft" types

Types as treated by type theory—"rigid" types:

- baked into the logical foundation,
- with accompanying syntactic features.

"Soft" types:

- *not* part of the underlying logical formalism,
- not restricted to *any* particular logical system—can be used generically,
- correspond, essentially, to predicates of the underlying logic.

Types in set theory

Consider axiomatic set theory on top of first-order classical logic.

Only one logical type: *everything is a set*.

Types in set theory

Consider axiomatic set theory on top of first-order classical logic.

Only one logical type: *everything is a set*—effectively untyped!

Types in set theory

Consider axiomatic set theory on top of first-order classical logic.

Only one logical type: *everything is a set*—effectively untyped!

But in practice, single out various kinds of sets by particular properties. . .

Types in set theory

Type		Definition (first-order formula)
A set n is a **natural number**	iff	$n \in \omega$, where ω is the first nonzero limit ordinal.
A set x is an **ordered triple**	iff	x is of the form $(a, (b, c))$, where we use the Kuratowski definition of the ordered pair.
An ordered triple $(G, \cdot, e)$ is a **group**	iff	$(\cdot \colon G \times G \to G) \wedge (\text{associativity of } \cdot) \wedge \ldots.$
A group $(G, \cdot, e)$ is **abelian**	iff	$\forall g, h \in G \ (g \cdot h = h \cdot g)$.

Types in set theory

Type		Definition (first-order formula)
A set n is a **natural number**	iff	$n \in \omega$, where ω is the first nonzero limit ordinal.
A set x is an **ordered triple**	iff	x is of the form $(a, (b, c))$, where we use the Kuratowski definition of the ordered pair.
An ordered triple $(G, \cdot, e)$ is a **group**	iff	$(\cdot\colon G \times G \to G) \wedge (\text{associativity of } \cdot) \wedge \ldots.$
A group $(G, \cdot, e)$ is **abelian**	iff	$\forall g, h \in G\ (g \cdot h = h \cdot g)$.

Here n, x, G, etc. are all sets, but we consider them instances of a particular *type* (of natural numbers, ordered triples, groups, . . .) if they satisfy some defining property.

Semantic types

Basic idea

A **semantic type** T of a logical theory $\mathcal{L}$ is given by a predicate ϕ_T in the language of $\mathcal{L}$. The **inhabitants of** T are the objects of $\mathcal{L}$ that satisfy ϕ_T.

Semantic types

Basic idea

A **semantic type** T of a logical theory $\mathcal{L}$ is given by a predicate ϕ_T in the language of $\mathcal{L}$. The **inhabitants of** T are the objects of $\mathcal{L}$ that satisfy ϕ_T.

The type T may be viewed as a *collection* of particular kinds of objects, or as a *label* applied to objects.

Semantic types

Basic idea

A **semantic type** T of a logical theory $\mathcal{L}$ is given by a predicate ϕ_T in the language of $\mathcal{L}$. The **inhabitants of** T are the objects of $\mathcal{L}$ that satisfy ϕ_T.

The type T may be viewed as a *collection* of particular kinds of objects, or as a *label* applied to objects.

Objects are allowed multiple semantic types.

Semantic types

Basic idea

A **semantic type** T of a logical theory $\mathcal{L}$ is given by a predicate ϕ_T in the language of $\mathcal{L}$. The **inhabitants of** T are the objects of $\mathcal{L}$ that satisfy ϕ_T.

The type T may be viewed as a *collection* of particular kinds of objects, or as a *label* applied to objects.

Objects are allowed multiple semantic types.

Extensions to this basic idea are possible! (e.g. Mizar's soft type system.)

Syntactic vs. semantic types

Syntactic vs. semantic types

Types. . .

- distinguish logical classes of objects,
- impose restrictions on formulas,
- help structure specifications and proofs.

Syntactic vs. semantic types

Types...

- distinguish logical classes of objects,
- impose restrictions on formulas,
- help structure specifications and proofs.

Syntactic vs. semantic types

Type-theoretic ("syntactic") types...

- distinguish logical classes of objects,
- impose syntactic restrictions on formulas (ill-typed sentences are disallowed),
- help structure specifications and proofs.

Syntactic vs. semantic types

Type-theoretic ("syntactic") types...

- distinguish logical classes of objects,
- impose syntactic restrictions on formulas (ill-typed sentences are disallowed),
- help structure specifications and proofs.

Semantic types...

- distinguish logical classes of objects,
- impose semantic restrictions on formulas (ill-"typed" sentences are always false/logically irrelevant/undefined),
- help structure specifications and proofs.

Semantic types in set theory

Basic idea

A **semantic type** T of a logical theory $\mathcal{L}$ is given by a predicate ϕ_T in the language of $\mathcal{L}$. The **inhabitants of** T are the objects of $\mathcal{L}$ that satisfy ϕ_T.

Semantic types in set theory

Basic idea

A **semantic type** T of a logical theory $\mathcal{L}$ is given by a predicate ϕ_T in the language of $\mathcal{L}$. The **inhabitants of** T are the objects of $\mathcal{L}$ that satisfy ϕ_T.

In set theory,

Semantic types correspond to subclasses

$$T = \{x \mid \phi_T(x)\}$$

of the universe of sets.

Semantic types in set theory

Type construction	Formulation as semantic types
Subtypes $A < B$	$A < B \iff \forall x\,(\phi_A(x) \longrightarrow \phi_B(x))$.
Intersection types $A \cap B$	$A \cap B = \{x \mid \phi_A(x) \wedge \phi_B(x)\}$.
Dependent types $P(x_1 \colon X_1, \ldots, x_n \colon X_n)$	$P = \{x \mid \phi_P(x; x_1, \ldots, x_n)\}$, where ϕ_P has parameters $x_i \in X_i$.
W-types (well-founded inductive types) T	$T = \bigcup\{F(z) \mid z \in Z\}$, where Z is a well-founded set and $F \colon Z \to V$ is defined by well-founded recursion.

Semantic types in set theory

Type construction	Formulation as semantic types
Subtypes $A < B$	$A < B \iff \forall x\,(\phi_A(x) \longrightarrow \phi_B(x))$.
Intersection types $A \cap B$	$A \cap B = \{x \mid \phi_A(x) \wedge \phi_B(x)\}$.
Dependent types $P(x_1 \colon X_1, \ldots, x_n \colon X_n)$	$P = \{x \mid \phi_P(x; x_1, \ldots, x_n)\}$, where ϕ_P has parameters $x_i \in X_i$.
W-types (well-founded inductive types) T	$T = \bigcup\{F(z) \mid z \in Z\}$, where Z is a well-founded set and $F \colon Z \to V$ is defined by well-founded recursion.

Example

The type of groups can be seen as a dependent pair type

$$Gp = \{x \mid \exists G, \cdot, e\ (x = (G, \cdot, e)\ \wedge\ \cdot \colon G \times G \to G\ \ \wedge\ \cdots)\}.$$

Semantic types in set theory

Type construction	Formulation as semantic types
Subtypes $A < B$	$A < B \iff \forall x\,(\phi_A(x) \longrightarrow \phi_B(x))$.
Intersection types $A \cap B$	$A \cap B = \{x \mid \phi_A(x) \wedge \phi_B(x)\}$.
Dependent types $P(x_1 \colon X_1, \ldots, x_n \colon X_n)$	$P = \{x \mid \phi_P(x; x_1, \ldots, x_n)\}$, where ϕ_P has parameters $x_i \in X_i$.
W-types (well-founded inductive types) T	$T = \bigcup\{F(z) \mid z \in Z\}$, where Z is a well-founded set and $F \colon Z \to V$ is defined by well-founded recursion.

Example

The type of groups can be seen as a dependent pair type

$$Gp = \{x \mid \exists G, \cdot, e\ (x = (G, \cdot, e) \wedge \underbrace{\cdot \colon G \times G \to G}_{\text{parameter } G} \wedge \cdots)\}.$$

Semantic types in set theory

Example: Nat List as a semantic inductive type

Semantic types in set theory

Example: Nat List as a semantic inductive type

Only need the following version of the recursion theorem on $(\mathbb{N}, <)$: For any function $G\colon V \to V$ there is a unique function $F\colon \mathbb{N} \to V$ such that

$$F(0) = G(0),$$
$$F(n+1) = G(F(n)).$$

Semantic types in set theory

Example: Nat List as a semantic inductive type

Only need the following version of the recursion theorem on $(\mathbb{N}, <)$: For any function $G \colon V \to V$ there is a unique function $F \colon \mathbb{N} \to V$ such that

$$F(0) = G(0),$$
$$F(n+1) = G(F(n)).$$

Defining $G(0) \coloneqq \{0\}$ and $G(X) \coloneqq \mathbb{N} \times X$, we get a function

$$\begin{aligned} F(0) &= \{0\}, \\ F(1) &= \{(n,0) \mid n \in \mathbb{N}\}, \\ F(2) &= \{(m,(n,0)) \mid m,n \in \mathbb{N}\}, \text{ etc.} \end{aligned}$$

Semantic types in set theory

Example: Nat List as a semantic inductive type

Only need the following version of the recursion theorem on $(\mathbb{N}, <)$: For any function $G\colon V \to V$ there is a unique function $F\colon \mathbb{N} \to V$ such that

$$F(0) = G(0),$$
$$F(n+1) = G(F(n)).$$

Defining $G(0) \coloneqq \{0\}$ and $G(X) \coloneqq \mathbb{N} \times X$, we get a function

$$\begin{aligned} &F(0) = \{0\}, \\ &F(1) = \{(n, 0) \mid n \in \mathbb{N}\}, \\ &F(2) = \{(m, (n, 0)) \mid m, n \in \mathbb{N}\}, \text{ etc.} \end{aligned}$$

Then Nat List $= \bigcup \{F(n) \mid n \in \mathbb{N}\}$.

Semantic types in Mizar

Mizar: proof assistant based on axiomatic set theory on top of first-order logic.

Semantic types in Mizar

Mizar: proof assistant based on axiomatic set theory on top of first-order logic.

Semantic types ("modes" and "attributes") defined via predicates.

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar "modes"

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar "modes" are hierarchical

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar “modes” are hierarchical dependent types

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar "modes" are hierarchical dependent types with implicit arguments.

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar "modes" are hierarchical dependent types with implicit arguments.

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar "modes" are hierarchical dependent types with implicit arguments.
- Mizar "attributes"

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar "modes" are hierarchical dependent types with implicit arguments.
- Mizar "attributes" modify specific types

Semantic types in Mizar

The type of thin subsets of a space X with respect to a measure M on X:

```
definition
  let X be set;
  let S be SigmaField of X;
  let M be sigma_Measure of S;
  mode thin of M -> Subset of X means
    ex B being set st B in S & it c= B & M.B = 0.;
```

The type of first-countable topological spaces:

```
definition
  let T be non empty TopStruct;
  attr T is first-countable means
    for x be Point of T ex B be Basis of x st B is countable;
```

- Mizar "modes" are hierarchical dependent types with implicit arguments.
- Mizar "attributes" modify specific types, extending the basic semantic type system.

Working with semantic types

Semantic type automation

As for type theoretic systems, so for semantic type systems.

Working with semantic types

Semantic type automation

As for type theoretic systems, so for semantic type systems.

- Implicit argument inference

Working with semantic types

Semantic type automation

As for type theoretic systems, so for semantic type systems.

- Implicit argument inference
- Type inference—possibility of multiple types means having to disambiguate in specific instances.

Working with semantic types

Semantic type automation

As for type theoretic systems, so for semantic type systems.

- Implicit argument inference
- Type inference—possibility of multiple types means having to disambiguate in specific instances.
- Type derivation—may need to widen, narrow, or derive other types for a term.

Working with semantic types

Semantic type automation

As for type theoretic systems, so for semantic type systems.

- Implicit argument inference
- Type inference—possibility of multiple types means having to disambiguate in specific instances.
- Type derivation—may need to widen, narrow, or derive other types for a term.

Type derivation

Sometimes easy:

$$x \text{ is infinite set} \implies x \text{ is nonempty set.}$$

Type derivation

Sometimes easy:

$$x \text{ is infinite set} \implies x \text{ is nonempty set.}$$

Sometimes less so:

$$G \text{ is a finite integral domain} \implies G \text{ is a field. (Wedderburn)}$$

$$f \text{ is a bounded entire function} \implies f \text{ is a constant function. (Liouville)}$$

Type derivation

Sometimes easy:

$$x \text{ is infinite set} \implies x \text{ is nonempty set.}$$

Sometimes less so:

$$G \text{ is a finite integral domain} \implies G \text{ is a field. (Wedderburn)}$$

$$f \text{ is a bounded entire function} \implies f \text{ is a constant function. (Liouville)}$$

Type derivation is needed everywhere in proof automation (when applying functions, instantiating premises of theorems, etc).

Type derivation, generally

Assume an ambient logical system $\mathcal{L}$ with implication $\Longrightarrow$.

Type derivation, generally

Assume an ambient logical system $\mathcal{L}$ with implication $\Longrightarrow$.

Definition

A **type rule** of a semantic type system in $\mathcal{L}$ is a theorem of the form

$$\psi_1 \implies \cdots \implies \psi_n \implies t\colon T,$$

where the ψ_i are formulas, t is a term, and T is a semantic type.

Type derivation, generally

Assume an ambient logical system $\mathcal{L}$ with implication $\implies$.

Definition

A **type rule** of a semantic type system in $\mathcal{L}$ is a theorem of the form

$$\psi_1 \implies \cdots \implies \psi_n \implies t \colon T,$$

where the ψ_i are formulas, t is a term, and T is a semantic type.

General type derivation task

Given collections $\mathcal{S}$ of terms, $\mathcal{T}$ of valid typing judgments, and $\mathcal{R}$ of type rules, return all typing judgments for terms in $\mathcal{S}$ that are derivable from $\mathcal{T}$ using the rules in $\mathcal{R}$.

Type derivation, generally

Assume an ambient logical system $\mathcal{L}$ with implication $\implies$.

Definition

A **type rule** of a semantic type system in $\mathcal{L}$ is a theorem of the form

$$\psi_1 \implies \cdots \implies \psi_n \implies t\colon T,$$

where the ψ_i are formulas, t is a term, and T is a semantic type.

General type derivation task

Given collections $\mathcal{S}$ of terms, $\mathcal{T}$ of valid typing judgments, and $\mathcal{R}$ of type rules, return all typing judgments for terms in $\mathcal{S}$ that are derivable from $\mathcal{T}$ using the rules in $\mathcal{R}$.

To be clear, we are not deriving new type *rules*, but using existing rules to derive new typing *judgments*.

Type derivation, generally

General type derivation task

Given collections $\mathcal{S}$ of terms, $\mathcal{T}$ of valid typing judgments, and $\mathcal{R}$ of type rules, return all typing judgments for terms in $\mathcal{S}$ that are derivable from $\mathcal{T}$ using the rules in $\mathcal{R}$.

Type derivation, generally

General type derivation task

Given collections $\mathcal{S}$ of terms, $\mathcal{T}$ of valid typing judgments, and $\mathcal{R}$ of type rules, return all typing judgments for terms in $\mathcal{S}$ that are derivable from $\mathcal{T}$ using the rules in $\mathcal{R}$.

Need not have a terminating algorithm in general.

Type derivation, generally

General type derivation task

Given collections $\mathcal{S}$ of terms, $\mathcal{T}$ of valid typing judgments, and $\mathcal{R}$ of type rules, return "enough" typing judgments for terms in $\mathcal{S}$ that are derivable from $\mathcal{T}$ using the rules in $\mathcal{R}$.

Need not have a terminating algorithm in general.

Type derivation, generally

General type derivation task

Given collections $\mathcal{S}$ of terms, $\mathcal{T}$ of valid typing judgments, and $\mathcal{R}$ of type rules, return "enough" typing judgments for terms in $\mathcal{S}$ that are derivable from $\mathcal{T}$ using the rules in $\mathcal{R}$.

Need not have a terminating algorithm in general.

Restricted versions of the problem do.

Type derivation in Mizar

Type rules ("cluster registrations") in Mizar have the abstract form

$$
\begin{aligned}
\llbracket x_1 \colon A_1, x_2 \colon A_2(x_1), \ldots, x_k \colon A_k(x_1, \ldots, x_{k-1}),& \\
t_1(x_1, \ldots, x_k) \colon B_1(x_1, \ldots, x_k), \ldots,& \\
t_n(x_1, \ldots, x_k) \colon B_n(x_1, \ldots, x_k)\rrbracket& \\
\Longrightarrow\ t(x_1, \ldots, x_k) \colon B(x_1, \ldots, x_k)&
\end{aligned}
$$

Type derivation in Mizar

Type rules ("cluster registrations") in Mizar have the abstract form

$$
\begin{aligned}
\llbracket x_1 : A_1, x_2 : A_2(x_1), \ldots, x_k : A_k(x_1, \ldots, x_{k-1}),&\\
t_1(x_1, \ldots, x_k) : B_1(x_1, \ldots, x_k), \ldots,&\\
t_n(x_1, \ldots, x_k) : B_n(x_1, \ldots, x_k)\rrbracket&\\
\implies t(x_1, \ldots, x_k) : B(x_1, \ldots, x_k)&
\end{aligned}
$$

x_i are variables

Type derivation in Mizar

Type rules ("cluster registrations") in Mizar have the abstract form

$$\llbracket x_1 \colon A_1, x_2 \colon A_2(x_1), \ldots, x_k \colon A_k(x_1, \ldots, x_{k-1}),\\ t_1(x_1, \ldots, x_k) \colon B_1(x_1, \ldots, x_k), \ldots,\\ t_n(x_1, \ldots, x_k) \colon B_n(x_1, \ldots, x_k)\rrbracket\\ \Longrightarrow\ t(x_1, \ldots, x_k) \colon B(x_1, \ldots, x_k)$$

x_i are variables, t_i, t are term constructors.

Type derivation in Mizar

Type rules ("cluster registrations") in Mizar have the abstract form

$$\llbracket x_1 \colon A_1, x_2 \colon A_2(x_1), \ldots, x_k \colon A_k(x_1, \ldots, x_{k-1}),$$
$$t_1(x_1, \ldots, x_k) \colon B_1(x_1, \ldots, x_k), \ldots,$$
$$t_n(x_1, \ldots, x_k) \colon B_n(x_1, \ldots, x_k)\rrbracket$$
$$\implies t(x_1, \ldots, x_k) \colon B(x_1, \ldots, x_k)$$

x_i are variables, t_i, t are term constructors.

Example

```
registration
  let T be TopSpace;
  let X, Y be open Subset of T;
  cluster X \/ Y -> open Subset of T;
```

Type derivation in Mizar

Type rules ("cluster registrations") in Mizar have the abstract form

$$
\begin{aligned}
[\![x_1 \colon A_1, x_2 \colon A_2(x_1), \ldots, x_k \colon A_k(x_1, \ldots, x_{k-1}),& \\
t_1(x_1, \ldots, x_k) \colon B_1(x_1, \ldots, x_k), \ldots,& \\
t_n(x_1, \ldots, x_k) \colon B_n(x_1, \ldots, x_k)]\!]& \\
\Longrightarrow\ t(x_1, \ldots, x_k) \colon B(x_1, \ldots, x_k)&
\end{aligned}
$$

x_i are variables, t_i, t are term constructors.

Example

```
registration
  let T be TopSpace;
  let X, Y be open Subset of T;
  cluster X \/ Y -> open Subset of T;
```

Abstract form:

$$
[\![T \colon \text{TopSpace}, X \colon \text{open Subset of T}, Y \colon \text{open Subset of T}]\!] \quad \Longrightarrow\ X \cup Y \colon \text{open Subset of T}
$$

Restricted type derivation

Type rules ("cluster registrations") in Mizar have the abstract form

$$\begin{aligned}
&[\![x_1 : A_1, x_2 : A_2(x_1), \ldots, x_k : A_k(x_1, \ldots, x_{k-1}),\\
&\qquad t_1(x_1, \ldots, x_k) : B_1(x_1, \ldots, x_k), \ldots,\\
&\qquad\quad t_n(x_1, \ldots, x_k) : B_n(x_1, \ldots, x_k)]\!]\\
&\qquad\qquad \Longrightarrow\; t(x_1, \ldots, x_k) : B(x_1, \ldots, x_k) \quad (*)
\end{aligned}$$

Restricted type derivation

Type rules ("cluster registrations") in Mizar have the abstract form

$$
\begin{aligned}
\llbracket x_1 \colon A_1, x_2 \colon A_2(x_1), \ldots, x_k \colon A_k(x_1, \ldots, x_{k-1}),& \\
t_1(x_1, \ldots, x_k) \colon B_1(x_1, \ldots, x_k), \ldots,& \\
t_n(x_1, \ldots, x_k) \colon B_n(x_1, \ldots, x_k)\rrbracket& \\
\Longrightarrow\ t(x_1, \ldots, x_k) \colon B(x_1, \ldots, x_k)& \quad (*)
\end{aligned}
$$

Restricted type derivation task

Given *finite* collections $\mathcal{S}$ of *closed* terms, $\mathcal{T}$ of valid typing judgments for terms in $\mathcal{S}$, and $\mathcal{R}$ of type rules of the form $(*)$, return all typing judgments for terms in $\mathcal{S}$ that are derivable from $\mathcal{T}$ using the rules in $\mathcal{R}$.

The type derivation algorithm

Algorithm: Type derivation

Input: Finite sets of closed terms $\mathcal{S}$, valid typings $\mathcal{T}$ for $\mathcal{S}$, and type rules $\mathcal{R}$ of the form $(*)$.

Output: A set $\mathcal{T}'$ of typing judgments for terms in $\mathcal{S}$.

1 Set $i := 0$, $T_i := \mathcal{T}$, and $T_{i+1} := \{\}$
2 **foreach** *term* s *in* $\mathcal{S}$ **do**
3   **foreach** *rule* r *in* $\mathcal{R}$ **do**
4   **if** s *unifies with* $t(x_1, \ldots, x_k)$ *via* σ **then**
5   Let n be the number of premises of $r\sigma$
6   **foreach** *n-tuple tys of typings (with repetition) in* T_i **do**
7   Try to discharge the premises of $r\sigma$ with the entries of *tys* in order, instantiating the variables x_i
8   **if** *all premises are discharged to yield a typing ty* **then** $T_{i+1} := T_{i+1} \cup \{ty\}$
9   **end**
10   **end**
11   **end**
12 **end**
13 **if** $T_i \neq T_{i+1}$ **then** set $i := i + 1$ and go to Line 2
14 **else** return T_i

The type derivation algorithm

Proposition

The type derivation algorithm is correct and terminates.

The type derivation algorithm

Proposition

The type derivation algorithm is correct and terminates.

Proof.

The type derivation algorithm

Proposition

The type derivation algorithm is correct and terminates.

Proof.

- Correctness is an invariant of the sets T_i—the typings in $T_0 = \mathcal{T}$ are valid by definition, and every typing in T_{i+1} is the result of discharging a type rule with valid typings (by the induction hypothesis), and is thus valid.

The type derivation algorithm

Proposition

The type derivation algorithm is correct and terminates.

Proof.

- Correctness is an invariant of the sets T_i—the typings in $T_0 = \mathcal{T}$ are valid by definition, and every typing in T_{i+1} is the result of discharging a type rule with valid typings (by the induction hypothesis), and is thus valid.
- Since $\mathcal{S}$ is finite, the only way to indefinitely generate new typing judgments is to generate infinitely many types $B = B(s_1, \ldots, s_k)$ such that $s\colon B$ for some $s \in \mathcal{S}$.

The type derivation algorithm

Proposition

The type derivation algorithm is correct and terminates.

Proof.

- Correctness is an invariant of the sets T_i—the typings in $T_0 = \mathcal{T}$ are valid by definition, and every typing in T_{i+1} is the result of discharging a type rule with valid typings (by the induction hypothesis), and is thus valid.
- Since $\mathcal{S}$ is finite, the only way to indefinitely generate new typing judgments is to generate infinitely many types $B = B(s_1, \ldots, s_k)$ such that $s \colon B$ for some $s \in \mathcal{S}$. Furthermore each s_j is a term that appears in a typing judgment of some T_i.

The type derivation algorithm

Proposition

The type derivation algorithm is correct and terminates.

Proof.

- Correctness is an invariant of the sets T_i—the typings in $T_0 = \mathcal{T}$ are valid by definition, and every typing in T_{i+1} is the result of discharging a type rule with valid typings (by the induction hypothesis), and is thus valid.
- Since $\mathcal{S}$ is finite, the only way to indefinitely generate new typing judgments is to generate infinitely many types $B = B(s_1, \ldots, s_k)$ such that $s\colon B$ for some $s \in \mathcal{S}$. Furthermore each s_j is a term that appears in a typing judgment of some T_i.
- But there are only finitely many choices for $B(x_1, \ldots, x_k)$ since $\mathcal{R}$ is finite, and there are also finitely many choices for the s_j since, by induction, only terms from $\mathcal{S}$ appear in typing judgments of the T_i.

$\square$

The type derivation algorithm, concretely

Recasts the difficulty of deriving "enough" type information as the problem of choosing a good initial set of terms $\mathcal{S}$.

The type derivation algorithm, concretely

Recasts the difficulty of deriving "enough" type information as the problem of choosing a good initial set of terms $\mathcal{S}$.

Implemented in Isabelle/Mizar, where $\mathcal{S}$ is chosen to be the set of all subterms appearing in the goal statement, or any assumptions visible in the proof context.

The type derivation algorithm, concretely

Recasts the difficulty of deriving “enough” type information as the problem of choosing a good initial set of terms $\mathcal{S}$.

Implemented in Isabelle/Mizar, where $\mathcal{S}$ is chosen to be the set of all subterms appearing in the goal statement, or any assumptions visible in the proof context.

Works well, mostly. In rare cases need to explicitly instantiate some hypotheses in order to add specific terms to $\mathcal{S}$.

Why semantic types?

Why semantic types?

Power and flexibility

Why semantic types?

Power and flexibility

- May use any logic of choice—not bound to Curry-Howard.

Why semantic types?

Power and flexibility

- May use any logic of choice—not bound to Curry-Howard.
- More flexible type discipline—multiple types allowed, as many ways to form types as there are to form predicates.

Why semantic types?

Power and flexibility

- May use any logic of choice—not bound to Curry-Howard.
- More flexible type discipline—multiple types allowed, as many ways to form types as there are to form predicates.

Simplicity and naturalness

Why semantic types?

Power and flexibility

- May use any logic of choice—not bound to Curry-Howard.
- More flexible type discipline—multiple types allowed, as many ways to form types as there are to form predicates.

Simplicity and naturalness

- Nontrivial constructions in type theory can be trivial in semantic types (e.g. subtypes, intersection types).

Why semantic types?

Power and flexibility

- May use any logic of choice—not bound to Curry-Howard.
- More flexible type discipline—multiple types allowed, as many ways to form types as there are to form predicates.

Simplicity and naturalness

- Nontrivial constructions in type theory can be trivial in semantic types (e.g. subtypes, intersection types).
- Closer to the practice of working mathematicians.

Why semantic types?

Power and flexibility

- May use any logic of choice—not bound to Curry-Howard.
- More flexible type discipline—multiple types allowed, as many ways to form types as there are to form predicates.

Simplicity and naturalness

- Nontrivial constructions in type theory can be trivial in semantic types (e.g. subtypes, intersection types).
- Closer to the practice of working mathematicians.
- May motivate greater adoption of proof assistants?

Semantic types in type theory. . . ?

Can add dependent types to HOL (approach taken by Isabelle object logics based on MLTT).

Semantic types in type theory. . . ?

Can add dependent types to HOL (approach taken by Isabelle object logics based on MLTT).

Current discussion about integrating support for semantic types into Isabelle. . .

Thanks!