Tego Type Checker

Brendon Bown

Tego Programming Language

A dynamically-typed functional programming language that unifies lists and tuples into a single-list like data structure

Fibonacci

```
main = fib 5

fib = fib' 0 1

fib' a b n =
    match n to
    | 0 -> b
    | n -> fib' b (a + b) (n - 1)
```

Output:

8

Fibonacci List

```
main = fibList 45

fibList = fibList' 0 1

fibList' a b n =
    match n to
    | 0 -> b
    | n -> b, fibList' b (a + b) (n - 1)
```

Output:

```
(1, 1, 2, 3, 5, 8,
13, 21, 34, 55, 89,
144, 233, 377, 610,
987, ...)
```

Maximum value in a list

```
max list =
    match list to
    | head, () -> head
    | head, tail ->
        let tailMax = max tail in
        if head > tailMax then
            head
        else
            tailMax
```

Note that the first branch matches `()` which indicates that the list is done.

In lists in Tego, `()` is treated as both `unit` and `nil`.

Tego Programming Language

A *dynamically-typed* functional programming language that unifies lists and tuples into a single-list like data structure

Project Goal

Add a **type system** to the dynamically typed language.

Project Goal (Specific)

Write a **type checker** for a **simplified** version of the language that type checks **expressions.**

AST

- Literal: **1**
- Variable: **x**
- Let expression: **let *x* = *e* in *body***
- Function: **fn *param* -> *body***
    - Note that functions only accept one parameter
- Function application: ***f arg***
- If expression: **if *cond* then *t* else *f***
- Match expression:

  match *e* to
  | *pat* -> *branch*
  | *pat* -> *branch*
  | …
- Binary operator expressions: **x *op* y**
- Unary operator expressions: ***op* x**

Types

- Int: **1**
- Char: **'a'**
- Bool: **true**
- Unit: **()**
- Function: **Int -> Bool**

Typing an If Expression

```
if (is_even x) then -- ???
    true            -- Bool
else
    0               -- Int
```

Should this be a valid expression? If so, how should it type?

Types

- Int: **1**
- Char: **'a'**
- Bool: **true**
- Unit: **()**
- Function: **Int -> Bool**
- Union: **Int | Bool**   <-- added a union type

Typing an If Expression

```
if (is_even x) then -- Bool | Int
    true            -- Bool
else
    0               -- Int
```

Introducing a union type allows us to type this if expression

Union Types → Subtypes

```
let x : Int | Bool = true in …
```

`true` is of type `Bool`, while `x` is expected to be of type `Int | Bool`

Subtypes

If a type **T** is a subtype of type **S**, then a value of type **T** can be used in any place where a value of type **S** is required.

*Example: a value of type **Int** can be used anywhere a value of type **Int | Bool** is required. Thus, **Int** is a subtype **Int | Bool**.*

Subtypes

T <: S can be read as “T is a subtype of S”

Example: Int <: Int | Bool

Subtypes

```
Inductive is_subtype : type -> type -> Prop :=
| TST_refl : forall t, t <: t
| TST_union_right : forall t t1 t2,
  t <: t1 \/ t <: t2 ->
  t <: (T_Union t1 t2)
| TST_union_both : forall t1 t2 t3 t4,
  t1 <: t3 \/ t1 <: t4 ->
  t2 <: t3 \/ t2 <: t4 ->
  (T_Union t1 t2) <: (T_Union t3 t4)
where " t '<:' t' " := (is_subtype t t').
```

Subtypes – Functions

A : Int | Bool -> Int can be used wherever **B : Int -> Int** is expected because **A** handles more inputs than **B** requires. Thus, **Int | Bool -> Int :> Int -> Int** (contravariance).

A : Int -> Int can be used wherever **B : Int -> Int | Bool** is expected because **A** produces fewer outputs than code that calls **B** expects. Thus, **Int -> Int :> Int -> Int | Bool** (covariance).

Subtypes – Functions

```
Inductive is_subtype : type -> type -> Prop :=
(* ... *)
| TST_function : forall t1 t2 t3 t4,
  t3 <: t1 -> (* input widening, contravariant *)
  t2 <: t4 -> (* output narrowing, covariant *)
  (T_Function t1 t2) <: (T_Function t3 t4)
(* ... *)
.
```

Subtypes – Functions

```
Theorem TST_function_param : forall t1 t2 t,
  t2 <: t1 ->
  (T_Function t1 t) <: (T_Function t2 t).
Proof.
  intros.
  apply TST_function.
  - apply H.
  - apply TST_refl.
Qed.
```

Subtypes – Functions

```
Theorem TST_function_return : forall t t1 t2,
  t1 <: t2 ->
  (T_Function t t1) <: (T_Function t t2).
Proof.
  intros.
  apply TST_function.
  - apply TST_refl.
  - apply H.
Qed.
```

Subtypes – Transitivity

```
Theorem is_subtype_trans : forall t1 t2 t3,
  t1 <: t2 ->
  t2 <: t3 ->
  t1 <: t3.
```

Subtypes – Associativity

```
Theorem is_subtype_assoc : forall t1 t2 t,
  (T_Union t1 t2) <: t (* T_Union t3 t4 *) <->
  (T_Union t2 t1) <: t (* T_Union t3 t4 *).
Proof.
  split;
  intros.
  - inversion H; subst.
    (* ... and so on ... *)
Abort.
```

Subtypes – Associativity

```
Inductive is_subtype : type -> type -> Prop :=
(* ... *)
| TST_union_assoc : forall t1 t2 t,
  (T_Union t1 t2) <: t ->
  (T_Union t2 t1) <: t
(* ... *)
.
```

Subtypes – Factoring

```
Theorem is_subtype_union__union_left : forall t1 t2 t3 t4,
  (T_Union t1 t2) <: (T_Union t3 t4) ->
  t1 <: (T_Union t3 t4).
Proof.
  intros.
  inversion H; subst.
  - left. constructor.
    (* ... and so on ... *)
Abort.
```

Subtypes – Factoring

```
Inductive is_subtype : type -> type -> Prop :=
(* ... *)
| TST_union_factor : forall t1 t2 t3 t4,
  (T_Union t1 t2) <: (T_Union t3 t4) ->
  t1 <: (T_Union t3 t4)
(* ... *)
.
```

Subtypes – Unions Revisited

```
Inductive is_subtype : type -> type -> Prop :=
(* ... *)
| TST_union_right : forall t t1 t2,
  t <: t1 \/ t <: t2 ->
  t <: (T_Union t1 t2)
| TST_union_both : forall t1 t2 t3 t4,
  t1 <: t3 \/ t1 <: t4 ->
  t2 <: t3 \/ t2 <: t4 ->
  (T_Union t1 t2) <: (T_Union t3 t4)
(* ... *)
.
```

Subtypes – Unions Revisited

```
Inductive is_subtype : type -> type -> Prop :=
(* ... *)
| TST_union_left : forall t t1 t2,
  t <: t1 ->
  t <: (T_Union t1 t2)
| TST_union_right : forall t t1 t2,
  t <: t2 ->
  t <: (T_Union t1 t2)
| TST_union_union : forall t1 t2 t3 t4,
  t1 <: (T_Union t3 t4) ->
  t2 <: (T_Union t3 t4) ->
  (T_Union t1 t2) <: (T_Union t3 t4)
(* ... *)
.
```

Subtypes – Unions Revisited

Is this valid?

Int | Int :> Int

```
| TST_union_union : forall t1 t2 t3 t4,
  t1 <: (T_Union t3 t4) ->
  t2 <: (T_Union t3 t4) ->
  (T_Union t1 t2) <: (T_Union t3 t4)
```

Subtypes – Unions Revisited

```
| TST_union_union : forall t1 t2 t3 t4,
  t1 <: (T_Union t3 t4) ->
  t2 <: (T_Union t3 t4) ->
  (T_Union t1 t2) <: (T_Union t3 t4)
```

Subtypes – Unions Revisited

```
| TST_union_union : forall t1 t2 t,
  t1 <: t ->
  t2 <: t ->
  (T_Union t1 t2) <: t
```

Subtypes – Unions Revisited

```
Theorem TST_union_union' : forall t1 t2 t3 t4,
  t1 <: (T_Union t3 t4) ->
  t2 <: (T_Union t3 t4) ->
  (T_Union t1 t2) <: (T_Union t3 t4).
Proof.
  intros.
  apply TST_union_union; assumption.
Qed.
```

Subtypes – Unions Revisited

```
| TST_union_factor : forall t1 t2 t3 t4,
  (T_Union t1 t2) <: (T_Union t3 t4) ->
  t1 <: (T_Union t3 t4)
```

Subtypes – Unions Revisited

```
| TST_union_factor : forall t1 t2 t,
  (T_Union t1 t2) <: t ->
  t1 <: t
```

Subtypes – Unions Revisited

```
Theorem TST_union_factor' : forall t1 t2 t3 t4,
  (T_Union t1 t2) <: (T_Union t3 t4) ->
  t1 <: (T_Union t3 t4).
Proof.
  intros.
  eapply TST_union_factor.
  apply H.
Qed.
```

Type Equivalence

In general, type equivalence is trivial. In most cases, a type is only equivalent to itself. The only interesting rules are those relating to the **union** type.

Note that **~=** is the chosen notation to indicate type equivalence.

Type Equivalence

```
Inductive tequiv : type -> type -> Prop :=
| TE_refl : forall t, t ~= t
| TE_union_assoc : forall t1 t2 t,
  T_Union t1 t2 ~= t ->
  T_Union t2 t1 ~= t
| TE_union_comm : forall t1 t2 t3 t,
  T_Union (T_Union t1 t2) t3 ~= t ->
  T_Union t1 (T_Union t2 t3) ~= t
where " t '~=' t' " := (tequiv t t')..
```

Type Equivalence

```
Inductive tequiv : type -> type -> Prop :=
(* ... *)
| TE_union : forall t1 t1' t2 t2' t,
  t1 ~= t1' ->
  t2 ~= t2' ->
  (T_Union t1 t2) ~= t ->
  (T_Union t1' t2') ~= t
(* ... *)
.
```

Type Equivalence

```
Inductive tequiv : type -> type -> Prop :=
(* ... *)
| TE_function : forall t1 t1' t2 t2' t,
  t1 ~= t1' ->
  t2 ~= t2' ->
  T_Function t1 t2 ~= t ->
  T_Function t1' t2' ~= t
(* ... *)
.
```

Type Equivalence

```
Inductive tequiv : type -> type -> Prop :=
(* ... *)
| TE_symm : forall t1 t2,
  t1 ~= t2 ->
  t2 ~= t1
| TE_trans : forall t1 t2 t3,
  t1 ~= t2 ->
  t2 ~= t3 ->
  t1 ~= t3
(* ... *)
.
```

Type Equivalence

```
Inductive tequiv : type -> type -> Prop :=
(* ... *)
| TE_union_merge : forall t1 t2 t,
  t1 ~= t ->
  t2 ~= t ->
  T_Union t1 t2 ~= t
(* ... *)
.
```

Type Equivalence in Subtypes

```
Inductive is_subtype : type -> type -> Prop :=
(* ... *)
| TST_refl : forall t1 t2,
  t1 ~= t2 ->
  t1 <: t2
(* ... *)
.
```

Type Equivalence in Subtypes

```
Inductive is_subtype : type -> type -> Prop :=
(* ... *)
| TST_refl : forall t, t <: t
| TST_equiv : forall t1 t2 t3 t4,
  t1 ~= t2 ->
  t3 ~= t4 ->
  t1 <: t3 ->
  t2 <: t4
(* ... *)
.
```

Type Equivalence in Subtypes

```
Theorem TST_equiv_left : forall t1 t2 t,
  t1 ~= t2 ->
  t1 <: t ->
  t2 <: t.
Proof.
  intros t1 t2 t H_equiv H_sub.
  eapply TST_equiv.
  - apply H_equiv.
  - apply TE_refl.
  - apply H_sub.
Qed.
```

Type Equivalence in Subtypes

```
Theorem TST_equiv_right : forall t t1 t2,
  t1 ~= t2 ->
  t <: t1 ->
  t <: t2.
Proof.
  intros t t1 t2 H_equiv H_sub.
  eapply TST_equiv.
  - apply TE_refl.
  - apply H_equiv.
  - apply H_sub.
Qed.
```

Subtypes – Unproved Theorems

```
Unproved theorems:
  Theorem TST_union_either : forall t t1 t2,
    t <: (T_Union t1 t2) ->
    t <: t1 \/ t <: t2.
  Theorem is_subtype_trans : forall t1 t2 t3,
    t1 <: t2 ->
    t2 <: t3 ->
    t1 <: t3.
```

Subtypes – Question

```
| TST_union_union : forall t1 t2 t3 t4,
  t1 <: (T_Union t3 t4) ->
  t2 <: (T_Union t3 t4) ->
    (T_Union t1 t2) <: (T_Union t3 t4)
| TST_union_factor : forall t1 t2 t3 t4,
  (T_Union t1 t2) <: (T_Union t3 t4) ->
  t1 <: (T_Union t3 t4)
| TST_union_assoc : forall t1 t2 t,
  (T_Union t1 t2) <: t ->
  (T_Union t2 t1) <: t
```

How do I decide which axioms should be included in the base relation and which ones should be proved? How do I decide when a given axiom is unprovable?

Takeaway

It is important to keep your relation axioms as small as possible.

Takeaway

```
| TST_equiv : forall t1 t2 t3 t4,
  t1 ~= t2 ->
  t3 ~= t4 ->
  t1 <: t3 ->
  t2 <: t4
```

```
Theorem TST_equiv_left : forall t1 t2 t,
  t1 ~= t2 ->
  t1 <: t ->
  t2 <: t.

Theorem TST_equiv_right : forall t t1 t2,
  t1 ~= t2 ->
  t <: t1 ->
  t <: t2.
```

Takeaway

```
| TST_union_union : forall t1 t2 t,
  t1 <: t ->
  t2 <: t ->
  (T_Union t1 t2) <: t
```

```
Theorem TST_union_union' : forall t1 t2 t3 t4,
  t1 <: (T_Union t3 t4) ->
  t2 <: (T_Union t3 t4) ->
    (T_Union t1 t2) <: (T_Union t3 t4).
```

Takeaway

However, it is also important to include every axiom required.

Takeaway

```
| TE_union_merge : forall t1 t2 t,
  t1 ~= t ->
  t2 ~= t ->
  T_Union t1 t2 ~= t

(* Int | Int ~= Int *)
```

Questions?