Nested Refinements: A Logic for Duck Typing

Ravi Chugh, Pat Rondon, Ranjit Jhala (UCSD)

What are “Dynamic Languages”?

tag tests	affect control flow
dictionary objects	indexed by arbitrary string keys
first-class functions	can appear inside objects

```
let onto callbacks f obj =
  if f = null then
    new List(obj, callbacks)
  else
    let cb = if tagof f = “Str” then obj[f] else f in
    new List(fun () -> cb obj, callbacks)
```

tag tests	affect control flow
dictionary objects	indexed by arbitrary string keys
first-class functions	can appear inside objects

Problem: Lack of static types

… makes rapid prototyping / multi-language applications easy

… makes reliability / performance / maintenance hard

This Talk: System D

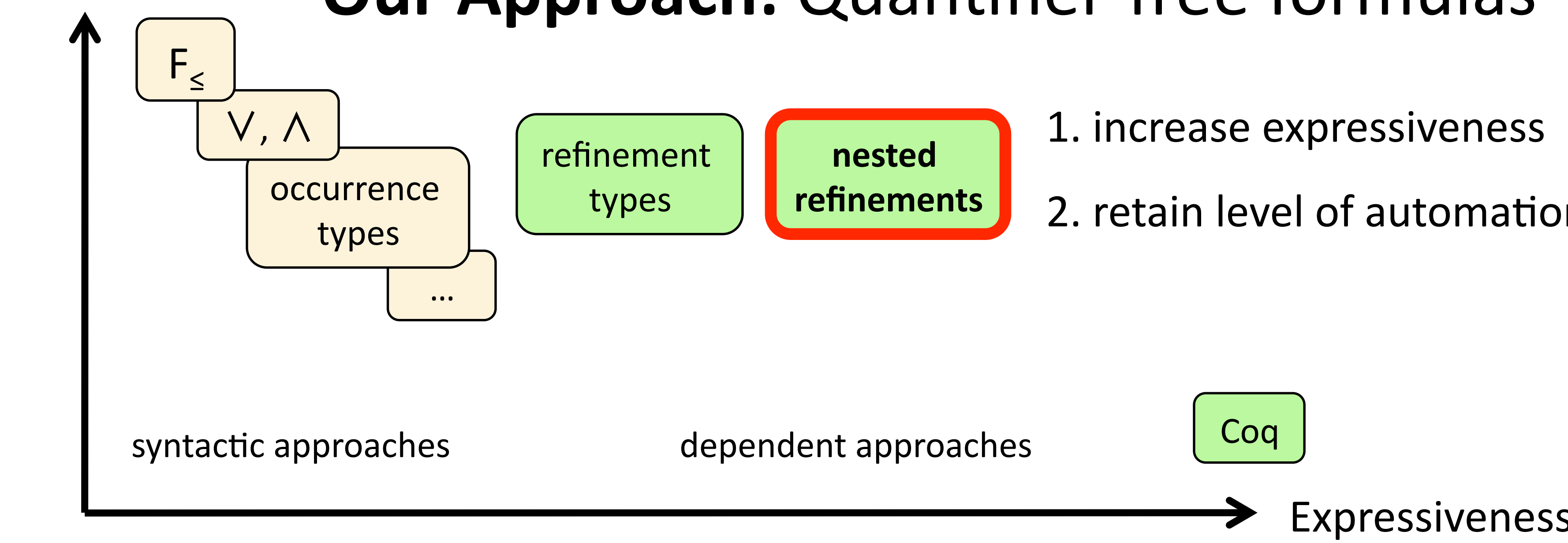
… a type system for these features

tag tests — affect control flow

dictionary objects — indexed by arbitrary string keys

first-class functions — can appear inside objects

Our Approach: Quantifier-free formulas



1. increase expressiveness
2. retain level of automation

tag tests	affect control flow
dictionary objects	indexed by arbitrary string keys
first-class functions	can appear inside objects

$$x :: \{\nu \mid \texttt{tag}(\nu) = \texttt{"Int"} \vee \texttt{tag}(\nu) = \texttt{"Bool"}\}$$

$$d :: \{\nu \mid \texttt{tag}(\nu) = \texttt{"Dict"} \wedge \texttt{tag}(\texttt{sel}(\nu, \texttt{"n"})) = \texttt{"Int"} \wedge \texttt{tag}(\texttt{sel}(\nu, \texttt{m})) = \texttt{"Int"}\}$$

Challenge: Functions inside dictionaries

Key Idea: Nested Refinements

```
1 + d[f](0)
```

```
d :: { ν | tag(ν) = “Dict”
        ∧ sel(ν,f) :: { ν | tag(ν)=“Int” }
                      → { ν | tag(ν)=“Int” } }
```

uninterpreted predicate
“x :: U” says
“x has-type U”

syntactic arrow type…

Key Idea: Nested Refinements

```
1 + d[f](0)
```

d :: { ν | tag(ν) = "Dict" ∧ sel(ν,f) :: { ν | tag(ν)="Int" } → { ν | tag(ν)="Int" } }

uninterpreted predicate "x :: U" says "x has-type U"

syntactic arrow type…

… but uninterpreted constant in the logic

Key Idea: Nested Refinements

- All values described by refinement formulas

$$T \; ::= \; \{ \nu \mid p \}$$

- "Has-type" predicate for arrows in formulas

$$p \; ::= \; \ldots \mid x :: y{:}T_1 \rightarrow T_2$$

- Can express idioms of dynamic languages

- Automatic type checking
  - Decidable refinement logic
  - Subtyping = SMT Validity + Syntactic Subtyping

Outline

Intro

Examples

Subtyping

Type Soundness

Conclusion

$x : \{ \nu \mid tag(\nu) = \text{"Int"} \lor tag(\nu) = \text{"Bool"} \} \to \{ \nu \mid tag(\nu) = tag(x) \}$

$x : \text{IntOrBool} \to \{ \nu \mid tag(\nu) = tag(x) \}$

```
let negate x =
  if tagof x = "Int" then 0 - x else not x
```

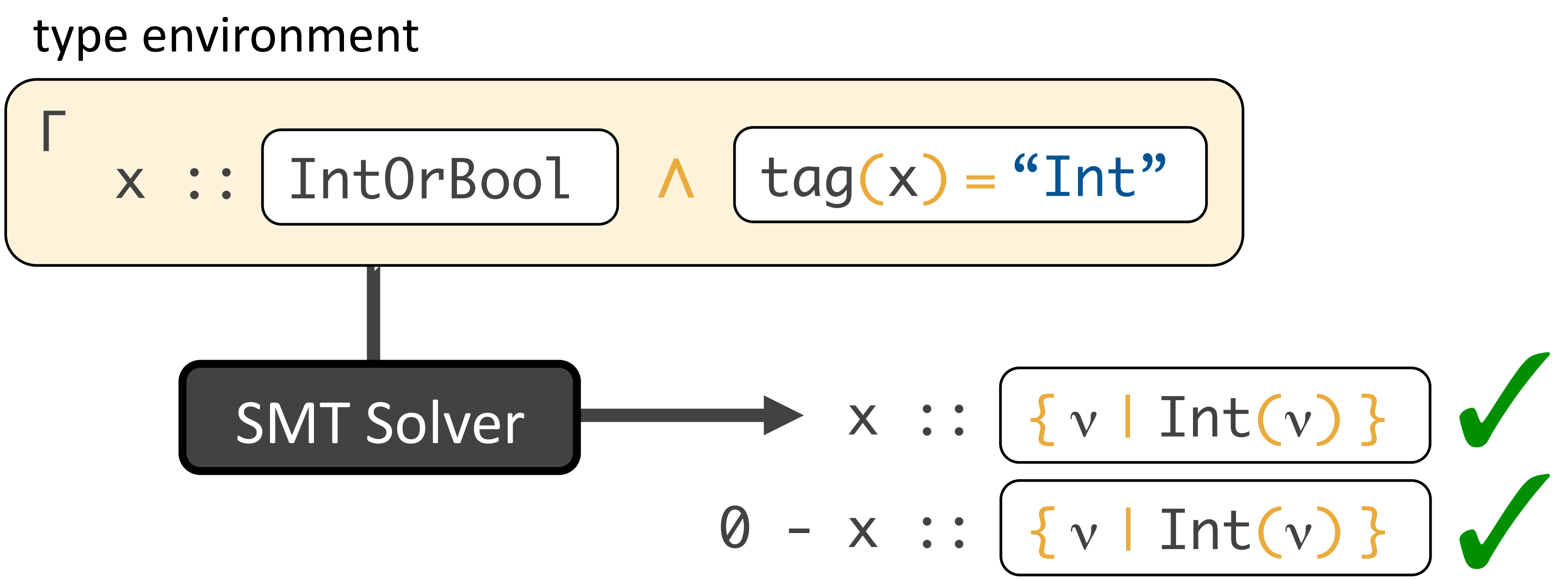
tagof :: $y : \text{Top} \to \{ \nu \mid \nu = tag(y) \}$

$y : \{ \nu \mid \text{true} \}$

```
x:IntOrBool → { ν | tag(ν) = tag(x) }
```

✓

```
let negate x =
  if tagof x = "Int" then 0 - x else not x
```



x:IntOrBool → { ν | tag(ν) = tag(x) } ✓

```
let negate x =
  if tagof x = "Int" then 0 – x else not x
```

type environment

Γ

x :: IntOrBool ∧ not (tag(x) = "Int")

SMT Solver →

x :: { ν | Bool(ν) } ✓

not x :: { ν | Bool(ν) } ✓

$$x{:}\texttt{IntOrBool} \rightarrow \{\, \nu \mid \texttt{tag}(\nu) = \texttt{tag}(x) \,\}$$

Nesting structure hidden with syntactic sugar

$$\{\, \nu \mid \nu :: x{:}\texttt{IntOrBool} \rightarrow \{\, \nu \mid \texttt{tag}(\nu) = \texttt{tag}(x) \,\} \,\}$$

Dictionary Operations

Types in terms of McCarthy operators

$$\texttt{mem} :: d{:}\texttt{Dict} \to k{:}\texttt{Str} \to \{\nu \mid \nu = \texttt{true} \Leftrightarrow \texttt{has}(d,k)\}$$

$$\texttt{get} :: d{:}\texttt{Dict} \to k{:}\{\nu \mid \texttt{has}(d,\nu)\} \to \{\nu \mid \nu = \texttt{sel}(d,k)\}$$

$$\texttt{set} :: d{:}\texttt{Dict} \to k{:}\texttt{Str} \to x{:}\texttt{Top} \to \{\nu \mid \nu = \texttt{upd}(d,k,x)\}$$

```
d:Dict → c:Str → Int
```

```
let getCount d c =                  get d c
  if mem d c then toInt (d[c]) else 0
```

$\{\, \nu \mid \nu = \texttt{true} \Leftrightarrow \texttt{has}(d, c) \,\}$

safe dictionary key lookup

$d{:}Dict \rightarrow c{:}Str \rightarrow Int$

```
let getCount d c =
  if mem d c then toInt (d[c]) else 0
```

tag(sel(ν,c)) = "Int"

$d{:}Dict \rightarrow c{:}Str \rightarrow \{\nu \mid EqMod(\nu,d,c) \wedge Int(\nu[c])\}$

```
let incCount d c =
  let i = getCount d c in {d with c = i + 1}
```

set d c (i+1)

Adding Type Constructors

$T ::= \{ \nu \mid p \}$

$p ::= \ldots \mid x :: U$

"type terms" → $U ::= y{:}T_1 \rightarrow T_2$

$\mid$ `A`

$\mid$ `List` T

$\mid$ `Null`

```
let apply f x = f x
```

$$\forall A,B.\ \{\nu \mid \nu :: \{\nu \mid \nu :: A\} \rightarrow \{\nu \mid \nu :: B\}\} \rightarrow \{\nu \mid \nu :: A\} \rightarrow \{\nu \mid \nu :: B\}$$

$$\forall A,B.\ (A \rightarrow B) \rightarrow A \rightarrow B$$

```
let dispatch d f = d[f](d)
```

```
∀A,B. d:{ν | ν :: A} → f:{ν | d[ν] :: A → B} → {ν | ν :: B}
```

a form of
"bounded quantification"

`d :: A` but additional constraints on `A`

`≈ ∀A <: {f: A → B}. d :: A`

$\forall$A,B.
{ν | ν :: A → B} → {ν | ν :: List[A]} → {ν | ν :: List[B]}

$\forall$A,B. (A → B) → List[A] → List[B]

```
let map f xs =
  if xs = null then null
  else new List(f xs["hd"], map f xs["tl"])
```

encode recursive data as dictionaries

```
let filter f xs =
  if xs = null then null
  else if not (f xs["hd"]) then filter f xs["tl"]
  else new List(xs["hd"], filter f xs["tl"])
```

usual definition,
but an interesting type

$\forall A, B.\ (x{:}A \to \{\, \nu \mid \nu = \mathtt{true} \Rightarrow x :: B \,\}) \to \mathtt{List}[A] \to \mathtt{List}[B]$

Outline

Intro

Examples

Subtyping

Type Soundness

Conclusion

type environment

Γ

applyInt :: $(\texttt{Int}, \texttt{Int} \rightarrow \texttt{Int}) \rightarrow \texttt{Int}$

negate :: $x{:}\texttt{IntOrBool} \rightarrow \{ \nu \mid tag(\nu) = tag(x) \}$

applyInt (42, negate)

SMT $\Gamma \wedge \nu = 42 \Rightarrow tag(\nu) = \text{“Int”}$ ✓

$$\frac{\text{SMT} \quad \Gamma \wedge \nu = 42 \Rightarrow tag(\nu) = \text{“Int”}}{\Gamma \vdash \{ \nu \mid \nu = 42 \} < \texttt{Int}}$$

type environment

Γ

applyInt :: (Int, Int → Int) → Int

negate :: x:IntOrBool → { ν | tag(ν) = tag(x) }

applyInt (42, negate)

SMT

… ∧ negate :: x:IorB → { ν | tag(ν) = tag(x) }

∧ ν = negate

⇒ ν :: Int → Int

Γ ⊢ { ν | ν = negate } < { ν | ν :: Int → Int }

type environment

Γ

$\texttt{applyInt} :: (\texttt{Int}, \texttt{Int} \to \texttt{Int}) \to \texttt{Int}$

$\texttt{negate} :: x{:}\texttt{IntOrBool} \to \{ \nu \mid \texttt{tag}(\nu) = \texttt{tag}(x) \}$

$\texttt{applyInt}\ (42, \texttt{negate})$

SMT

$\ldots \wedge \texttt{negate} :: x{:}\texttt{IorB} \to \{ \nu \mid \texttt{tag}(\nu) = \texttt{tag}(x) \}$

$\wedge\ \nu = \texttt{negate}$

$\not\Rightarrow \nu :: \texttt{Int} \to \texttt{Int}$

distinct uninterpreted constants!

$$\Gamma \vdash \{ \nu \mid \nu = \texttt{negate} \} < \{ \nu \mid \nu :: \texttt{Int} \to \texttt{Int} \}$$

Invalid, since these are uninterpreted constants

$$v :: x{:}\mathtt{IorB} \to \{ v \mid tag(v) = tag(x) \}$$

$$\not\Rightarrow$$

$$v :: \mathtt{Int} \to \mathtt{Int}$$

Want conventional syntactic subtyping

$$\frac{\dfrac{tag(v) = \text{"Int"} \Rightarrow tag(v) = \text{"Int"} \lor tag(v) = \text{"Bool"} \;\checkmark}{\mathtt{Int} <: \mathtt{IorB}} \qquad \dfrac{tag(v) = \text{"Int"} \land tag(v) = tag(x) \Rightarrow tag(v) = \text{"Int"} \;\checkmark}{\{ v \mid tag(v) = tag(x) \} <: \mathtt{Int}}}{\mathtt{IorB} \to \{ v \mid tag(v) = tag(x) \} <: \mathtt{Int} \to \mathtt{Int}}$$

Subtyping with Nesting

To prove $p \Rightarrow q$:

1) Convert q to CNF clauses $(q_{11} \vee \dots) \wedge \dots \wedge (q_{n1} \vee \dots)$

2) For each clause, discharge some literal q_{ij} as follows:

base predicate: $p \Rightarrow q_{ij}$

anything except $x :: U$

e.g. $tag(v) = tag(x)$

$tag(sel(d,k)) =$ "Int"

Subtyping with Nesting

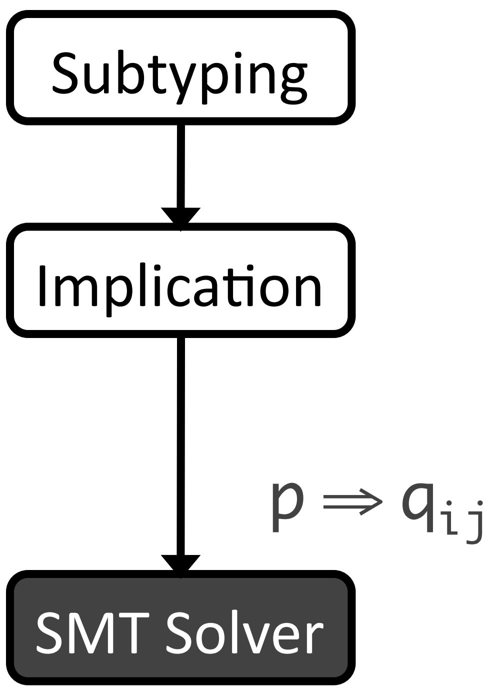
To prove $p \Rightarrow q$:

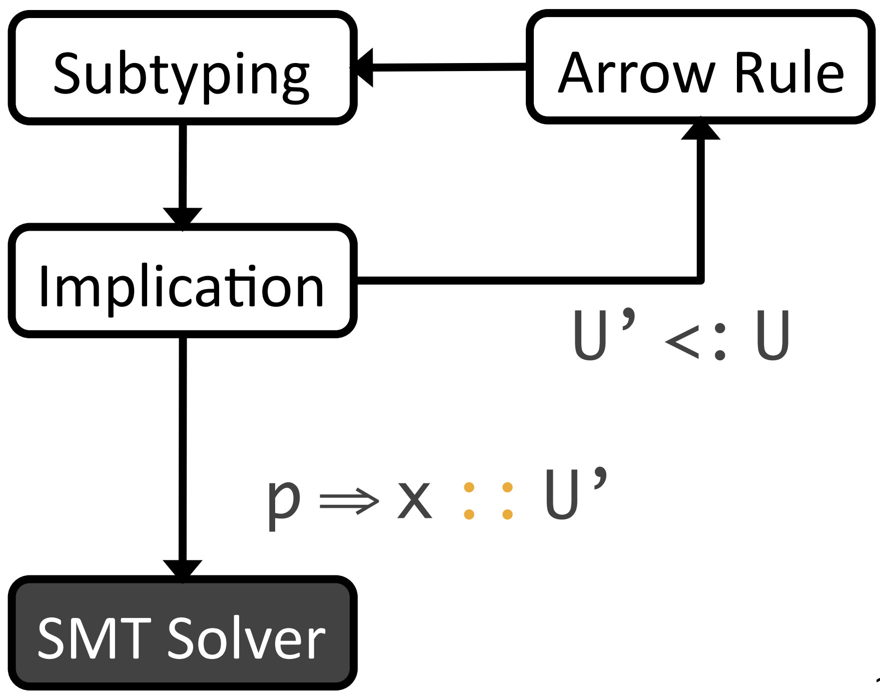
1) Convert q to CNF clauses $(q_{11} \vee \ldots) \wedge \ldots \wedge (q_{n1} \vee \ldots)$

2) For each clause, discharge some literal q_{ij} as follows:

base predicate: $p \Rightarrow q_{ij}$

"has-type" predicate: $p \Rightarrow x :: U$





```
applyInt (42, negate)
```

Uninterpreted Reasoning

$$\ldots \wedge \texttt{negate} :: \boxed{x{:}\texttt{IorB} \rightarrow \{\, \nu \mid tag(\nu) = tag(x) \,\}}$$
$$\wedge\ \nu = \texttt{negate}$$
$$\Rightarrow\ \nu :: \boxed{x{:}\texttt{IorB} \rightarrow \{\, \nu \mid tag(\nu) = tag(x) \,\}}$$

+

Syntactic Reasoning

$$\Gamma \vdash \boxed{x{:}\texttt{IorB} \rightarrow \{\, \nu \mid tag(\nu) = tag(x) \,\}} <: \boxed{\texttt{Int} \rightarrow \texttt{Int}}$$

$$\Gamma \vdash \{\, \nu \mid \nu = \texttt{negate} \,\} < \{\, \nu \mid \nu :: \texttt{Int} \rightarrow \texttt{Int} \,\}$$

Outline

Intro

Examples

Subtyping

Type Soundness

Conclusion

Substitution Lemma		
	If	$x:T_x, \Gamma \vdash e :: T$
	and	$\vdash v :: T_x$
	then	$\Gamma[v/x] \vdash e[v/x] :: T[v/x]$

independent of $\emptyset$, and just echoes the binding from the environment

$$f:\{\nu \mid \nu :: Int \rightarrow Int\} \vdash 0 :: \{\nu \mid f :: Int \rightarrow Int\}$$

$$\vdash \lambda x.x+1 :: \{\nu \mid \nu :: Int \rightarrow Int\}$$

$$\vdash 0 :: \{\nu \mid \lambda x.x+1 :: Int \rightarrow Int\}$$

Substitution Lemma

If $x{:}T_x, \Gamma \vdash e :: T$

and $\vdash v :: T_x$

then $\Gamma[v/x] \vdash e[v/x] :: T[v/x]$

1st attempt

$$\dfrac{0 :: \{\nu \mid \nu = 0\} \qquad \dfrac{\text{SMT} \quad \nu = 0 \not\Rightarrow \lambda x.x{+}1 :: \text{Int} \rightarrow \text{Int}}{\{\nu \mid \nu = 0\} < \{\nu \mid \lambda x.x{+}1 :: \text{Int} \rightarrow \text{Int}\}}}{\vdash 0 :: \{\nu \mid \lambda x.x{+}1 :: \text{Int} \rightarrow \text{Int}\}}$$

Substitution Lemma ✗

If $x{:}T_x, \Gamma \vdash e :: T$

and $\vdash v :: T_x$

then $\Gamma[v/x] \vdash e[v/x] :: T[v/x]$

2nd attempt

SMT: $\nu = 0 \;\not\Rightarrow\; \nu :: U'$

\+

Arrow: $U' <: \text{Int} \to \text{Int}$

$$\frac{0 :: \{\nu \mid \nu = 0\} \qquad \{\nu \mid \nu = 0\} < \{\nu \mid \lambda x.x{+}1 :: \text{Int} \to \text{Int}\}}{\vdash 0 :: \{\nu \mid \lambda x.x{+}1 :: \text{Int} \to \text{Int}\}}$$

[S-Valid-Uninterpreted]

$$\frac{\text{SMT}\ \Gamma \wedge p \Rightarrow q}{\Gamma \vdash \{\nu \mid \nu = p\} < \{\nu \mid \nu = q\}}$$

[S-Valid-Interpreted]

$$\frac{\mathcal{I}_n \vDash \Gamma \wedge p \Rightarrow q}{\Gamma \vdash_n \{\nu \mid \nu = p\} < \{\nu \mid \nu = q\}}$$

- Rule not closed under substitution
- Interpret formulas by "hooking back" into type system
- Stratification to create ordering for induction

$$\mathcal{I}_n \vDash \lambda x.x+1 :: \text{Int} \rightarrow \text{Int}$$

iff

$$\vdash_{n-1} \lambda x.x+1 :: \{\nu \mid \nu :: \text{Int} \rightarrow \text{Int}\}$$

Type Soundness

Stratified Substitution Lemma	If	$x{:}T_x, \Gamma \vdash_n e :: T$
	and	$\vdash_n v :: T_x$
	then	$\Gamma[v/x] \vdash_{n+1} e[v/x] :: T[v/x]$

"Level 0" for type checking source programs, using only [S-Valid-Uninterpreted]

Stratified Preservation	If	$\vdash_0 e :: T$	and $e \rightarrow v$
	then	$\vdash_m v :: T$	for some m

artifact of the metatheory

Recap

- Dynamic languages make heavy use of:
  - run-time tag tests, dictionary objects, lambdas
- Nested refinements
  - generalizes refinement type architecture
  - enables combination of dictionaries and lambdas
- Decidable refinement logic
  - all proof obligations discharged algorithmically
  - novel subtyping decomposition to retain precision
- Syntactic type soundness

Future Work

- Imperative Updates
- Inheritance (prototypes in JS, classes in Python)
- Applications
- More local type inference / syntactic sugar
- Dictionaries in statically-typed languages

Thanks!

ravichugh.com/nested

Extra Slides

Constants

$$\text{tagof} :: x{:}\text{Top} \rightarrow \{ \nu \mid \nu = \text{tag}(x) \}$$

$$\text{mem} :: d{:}\text{Dict} \rightarrow k{:}\text{Str} \rightarrow \{ \nu \mid \text{Bool}(\nu) \wedge \nu = \text{True} \Leftrightarrow \text{has}(d,k) \}$$

$$\text{get} :: d{:}\text{Dict} \rightarrow k{:}\{ \nu \mid \text{Str}(\nu) \wedge \text{has}(d,\nu) \} \rightarrow \{ \nu \mid \nu = \text{sel}(d,k) \}$$

$$\text{set} :: d{:}\text{Dict} \rightarrow k{:}\text{Str} \rightarrow x{:}\text{Top} \rightarrow \{ \nu \mid \nu = \text{upd}(d,k,x) \}$$

$$\text{rem} :: d{:}\text{Dict} \rightarrow k{:}\text{Str} \rightarrow \{ \nu \mid \nu = \text{upd}(d,k,\text{bot}) \}$$

Macros

- Types

```
Int ≡ { v | tag(v)="Int" }

x:T1 → T2 ≡ { v | v :: x:T1 → T2 }
```

- Formulas

```
Str(x) ≡ tag(x) = "Str"
has(d,k) ≡ sel(d,k) != bot
EqMod(d,d',k) ≡ ∀k'. k' != k ⇒
                    sel(d,k) != sel(d',k)
```

- Logical Values

```
x.k ≡ sel(v,"k")
x[k] ≡ sel(v,k)
```

Onto

functional version of Dojo function

```
let onto callbacks f obj =
  if f = null then
    new List(obj,callbacks)
  else
    let cb = if tagof f = “Str” then obj[f] else f in
    new List(fun () -> cb obj, callbacks)
```

onto ::

$$\forall A.\ \texttt{callbacks}:\texttt{List}[\texttt{Top} \rightarrow \texttt{Top}]$$
$$\rightarrow \texttt{f}:\{\nu \mid \nu = \texttt{null} \vee \texttt{Str}(\nu) \vee \nu :: A \rightarrow \texttt{Top}\}$$
$$\rightarrow \texttt{obj}:\{\nu \mid \nu :: A$$
$$\wedge (\texttt{f} = \texttt{null} \Rightarrow \nu :: A \rightarrow \texttt{Int})$$
$$\wedge (\texttt{Str}(\texttt{f}) \Rightarrow \nu[\texttt{f}] :: A \rightarrow \texttt{Int})\}$$
$$\rightarrow \texttt{List}[\texttt{Top} \rightarrow \texttt{Top}]$$

Onto (2)

functional version of Dojo function

```
let onto (callbacks,f,obj) =
  if f = null then
    new List(obj,callbacks)
  else
    let cb = if tagof f = "Str" then obj[f] else f in
    new List(fun () -> cb obj, callbacks)
```

onto ::

$$\begin{aligned}
&\texttt{callbacks}:\texttt{List}[\texttt{Top} \rightarrow \texttt{Top}] \\
&* \ \texttt{f}:\{\, g \mid g = \texttt{null} \vee \texttt{Str}(g) \vee g :: \{\, x \mid x = \texttt{obj} \,\} \rightarrow \texttt{Top} \,\} \\
&* \ \texttt{obj}:\{\, o \mid (f = \texttt{null} \Rightarrow o :: \{\, x \mid x = o \,\} \rightarrow \texttt{Int}) \\
&\qquad\qquad \wedge (\texttt{Str}(f) \Rightarrow o[f] :: \{\, x \mid x = o \,\} \rightarrow \texttt{Int}) \,\} \\
&\rightarrow \texttt{List}[\texttt{Top} \rightarrow \texttt{Top}]
\end{aligned}$$

Traditional vs. Nested Refinements

Approach: Refinement Types

- Reuse refinement type architecture

- Find a decidable refinement logic for
  - Tag-tests ✓
  - Dictionaries ✓
  - Lambdas ✗ ✓*

- Define **nested** refinement type architecture

Nested Refinements

- Refinement formulas over a decidable logic
  - uninterpreted functions, McCarthy arrays, linear arithmetic
- **All values** refined by formulas

```
T  ::= { ν | p }
U  ::= x:T1 → T2
p  ::= p ∧ q  |  …
    |  x = y | x < y |  …
    |  tag(x) = “Int” |  …
    |  sel(x,y) = z |  …
```

```
T  ::= { ν | p }
    |  x:T1 → T2
p  ::= p ∧ q  |  …
    |  x = y | x < y |  …
    |  tag(x) = “Int” |  …
    |  sel(x,y) = z |  …
```

traditional refinements

Nested Refinements

- Refinement formulas over a decidable logic
  - uninterpreted functions, McCarthy arrays, linear arithmetic
- **All values** refined by formulas
- "has-type" allows "type terms" in formulas

```
T ::= {ν | p}
U ::= x:T₁ → T₂
p ::= p ∧ q  |  …
   |  x = y | x < y | …
   |  tag(x) = "Int"  | …
   |  sel(x,y) = z | …
   |  x :: U
```

```
T ::= {ν | p}
   |  x:T₁ → T₂
p ::= p ∧ q  |  …
   |  x = y | x < y | …
   |  tag(x) = "Int"  | …
   |  sel(x,y) = z | …
```

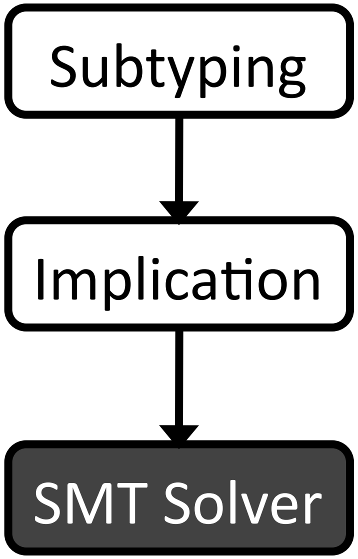
traditional refinements

Nested Refinements

- Refinement formulas over a decidable logic
  - uninterpreted functions, McCarthy arrays, linear arithmetic
- **All values** refined by formulas
- "has-type" allows "type terms" in formulas

```
T ::= {ν | p}
U ::= x:T1 → T2
p ::= p ∧ q | …
   |  x = y | x < y | …
   |  tag(x) = "Int" | …
   |  sel(x,y) = z | …
   |  x :: U
```

Subtyping (Traditional Refinements)



$$\frac{\texttt{tag}(\nu) = \text{“}\texttt{Int}\text{”} \;\Rightarrow\; \texttt{true}}{\texttt{Int} \;<:\; \texttt{Top}}$$

$$\begin{array}{lll} T & ::= & \{\, \nu \mid p \,\} \\ & \mid & x{:}T_1 \rightarrow T_2 \end{array}$$

traditional refinements

Subtyping (Traditional Refinements)

Subtyping

Implication

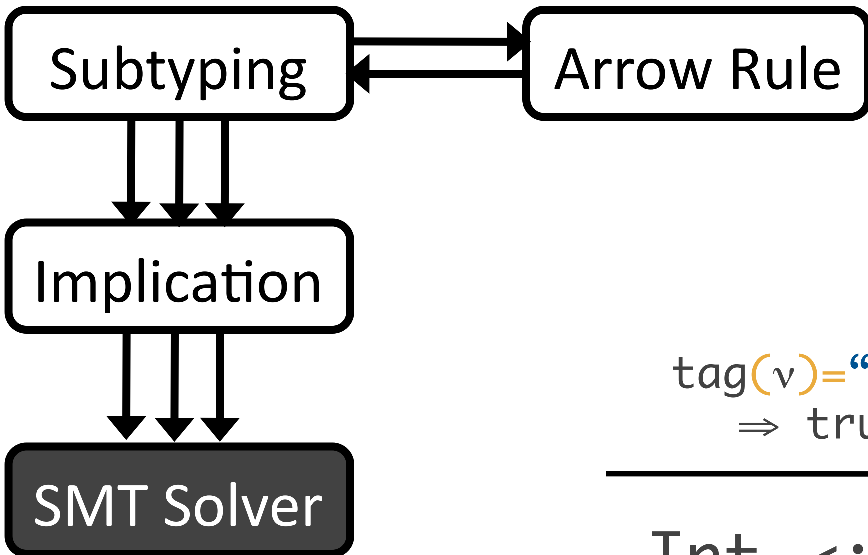
SMT Solver

$$\dfrac{\dfrac{\texttt{tag}(\nu)=\text{“Int”} \Rightarrow \texttt{true}}{\texttt{Int} <: \texttt{Top}} \qquad \dfrac{\texttt{tag}(\nu)=\text{“Int”} \Rightarrow \texttt{tag}(\nu)=\text{“Int”}}{\texttt{Int} <: \texttt{Int}}}{\texttt{Top} \rightarrow \texttt{Int} <: \texttt{Int} \rightarrow \texttt{Int}}$$

$$\begin{array}{lll} T & ::= & \{\, \nu \mid p \,\} \\ & \mid & x{:}T_1 \rightarrow T_2 \end{array}$$

traditional refinements

Subtyping (Traditional Refinements)

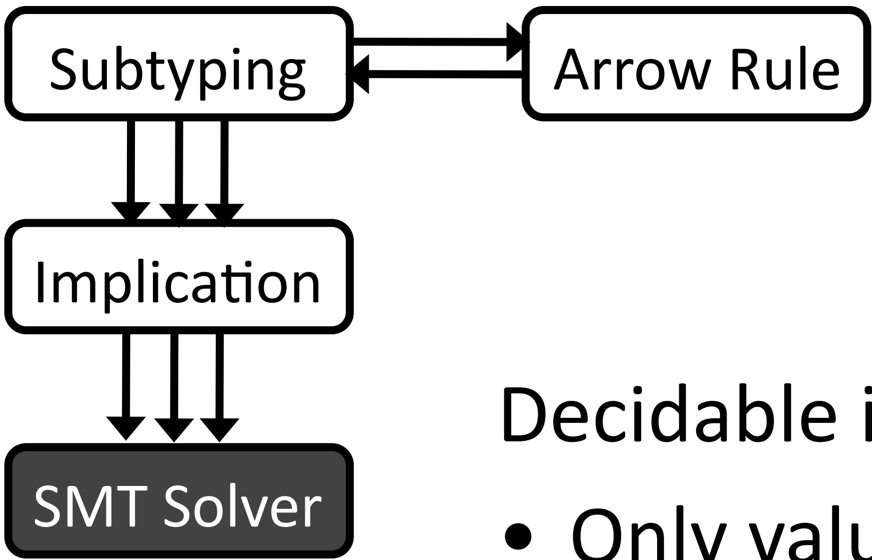


$$\frac{\dfrac{\texttt{tag}(\nu)=\text{"Int"} \Rightarrow \texttt{true}}{\texttt{Int} <: \texttt{Top}} \qquad \dfrac{\texttt{tag}(\nu)=\text{"Int"} \Rightarrow \texttt{tag}(\nu)=\text{"Int"}}{\texttt{Int} <: \texttt{Int}}}{\texttt{Top} \rightarrow \texttt{Int} <: \texttt{Int} \rightarrow \texttt{Int}}$$

$$\begin{array}{lll} T & ::= & \{\nu \mid p\} \\ & \mid & x{:}T_1 \rightarrow T_2 \end{array}$$

traditional refinements

Subtyping (Traditional Refinements)



Decidable if:

- Only values in formulas
- Underlying theories decidable

$$
\begin{aligned}
T \;::=&\; \{\nu \mid p\} \\
\mid&\;\; x{:}T_1 \rightarrow T_2
\end{aligned}
$$

traditional refinements