Nested Refinements: A Logic for Duck Typing

Ravi Chugh, Pat Rondon, Ranjit Jhala (UCSD)

Terminology

record fields

“dictionary” = finite map from **“keys”** to values

properties

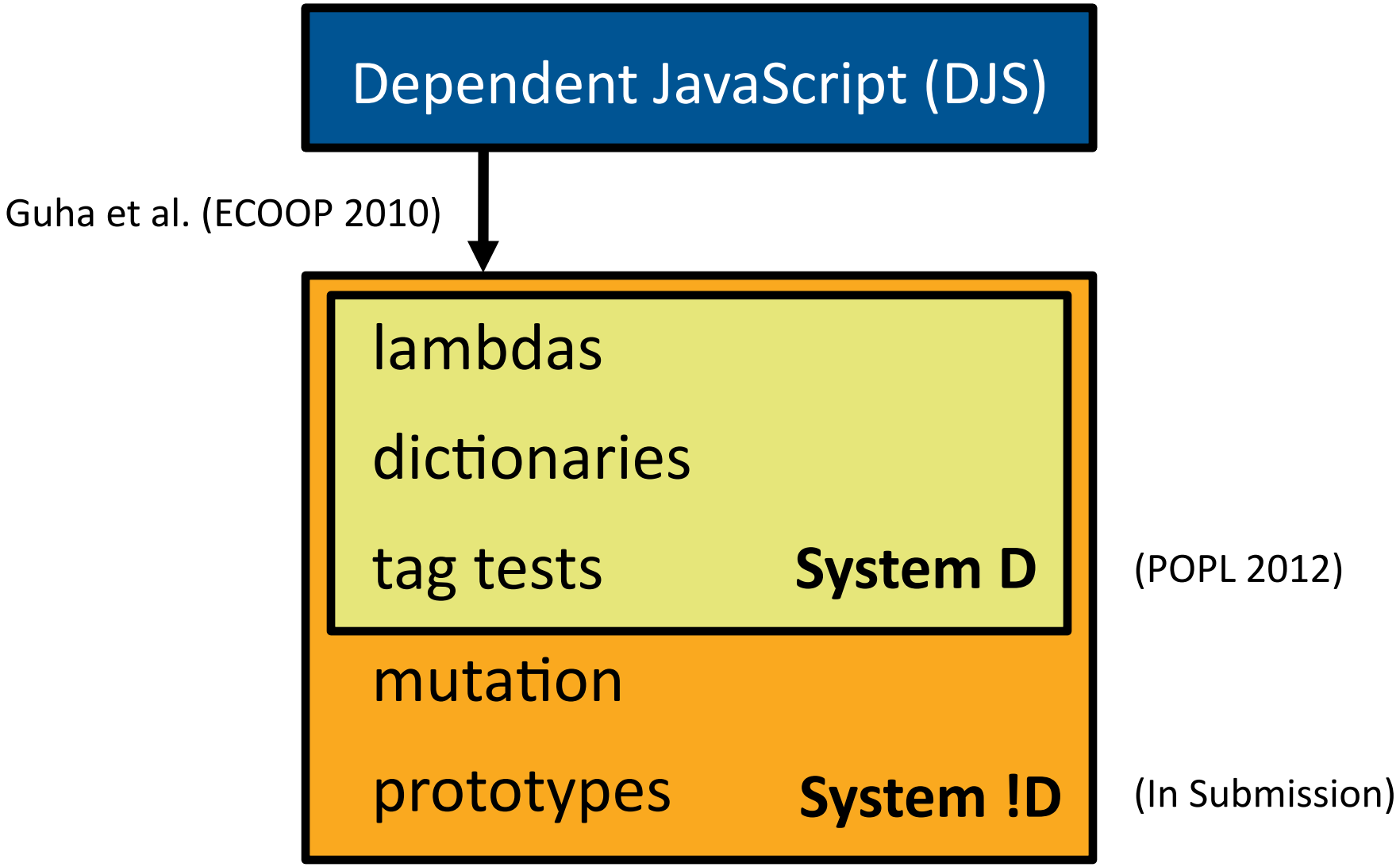
“object” = mutable dictionary

imperative

JavaScript has **“prototype”** inheritance

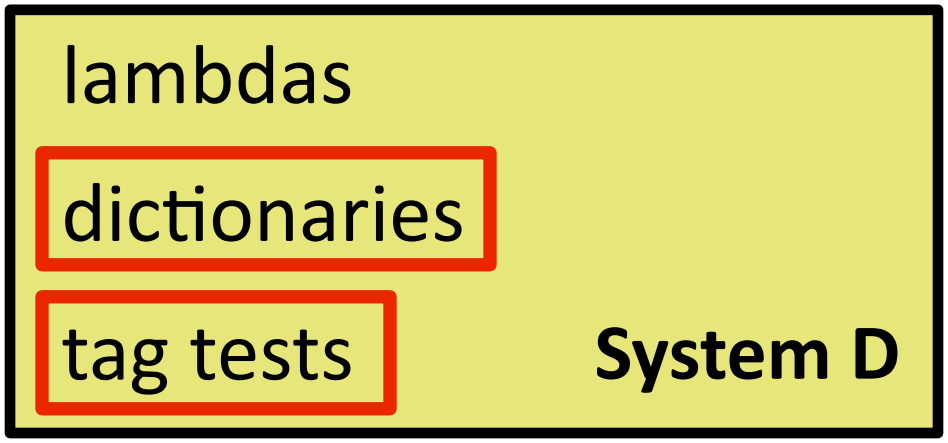
delegation

Types for JavaScript: Approach



x :: { v | tag(v)="Int" ∨ tag(v)="Bool" }

d :: { v | tag(v)="Dict"
  ∧ tag(sel(v,"n"))="Int"
  ∧ tag(sel(v,m))="Int" }

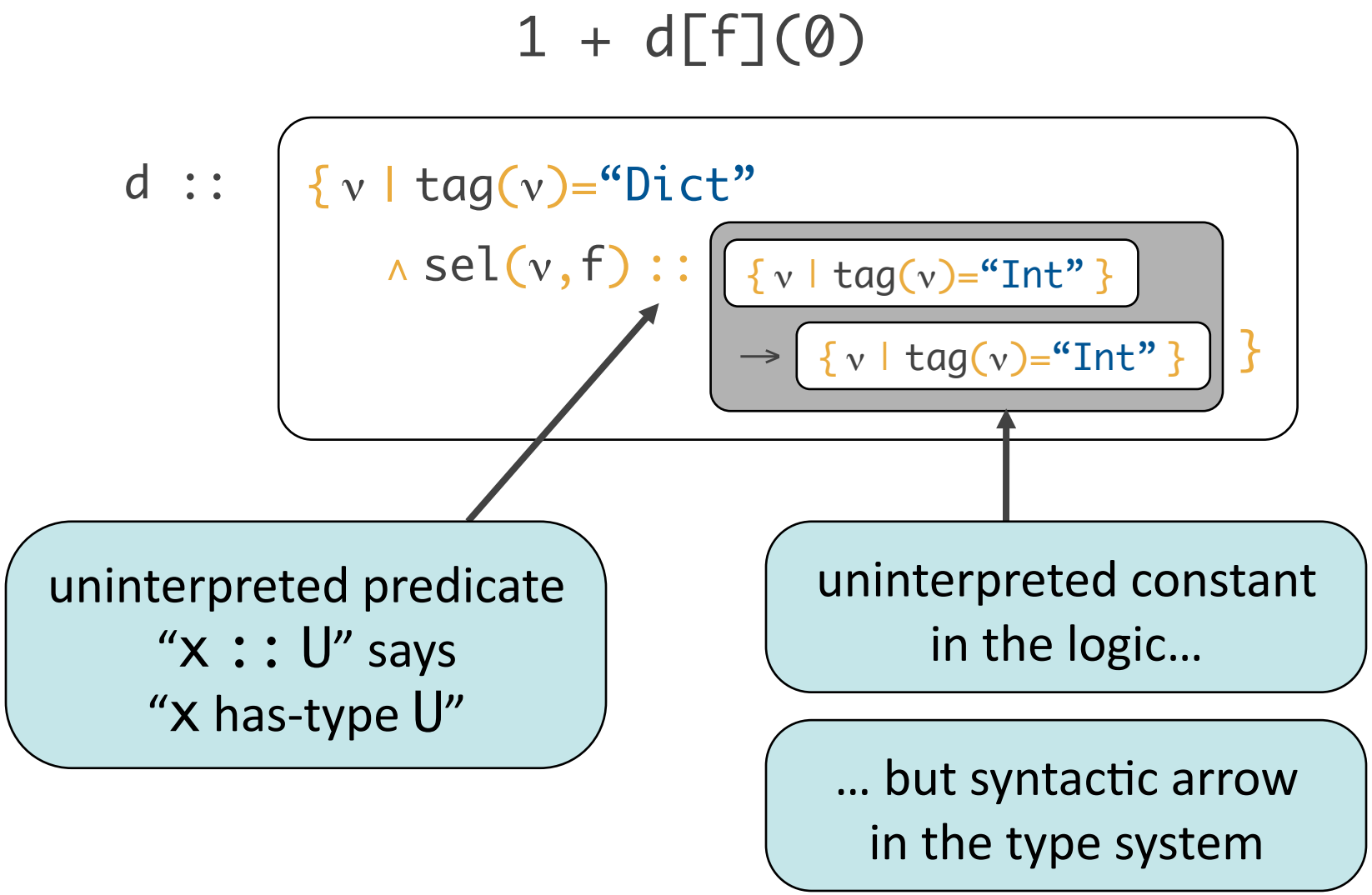


(POPL 2012)

Approach: All types as quantifier-free formulas

Challenge: Functions inside dictionaries

Key Idea: Nested Refinements



Key Idea: Nested Refinements

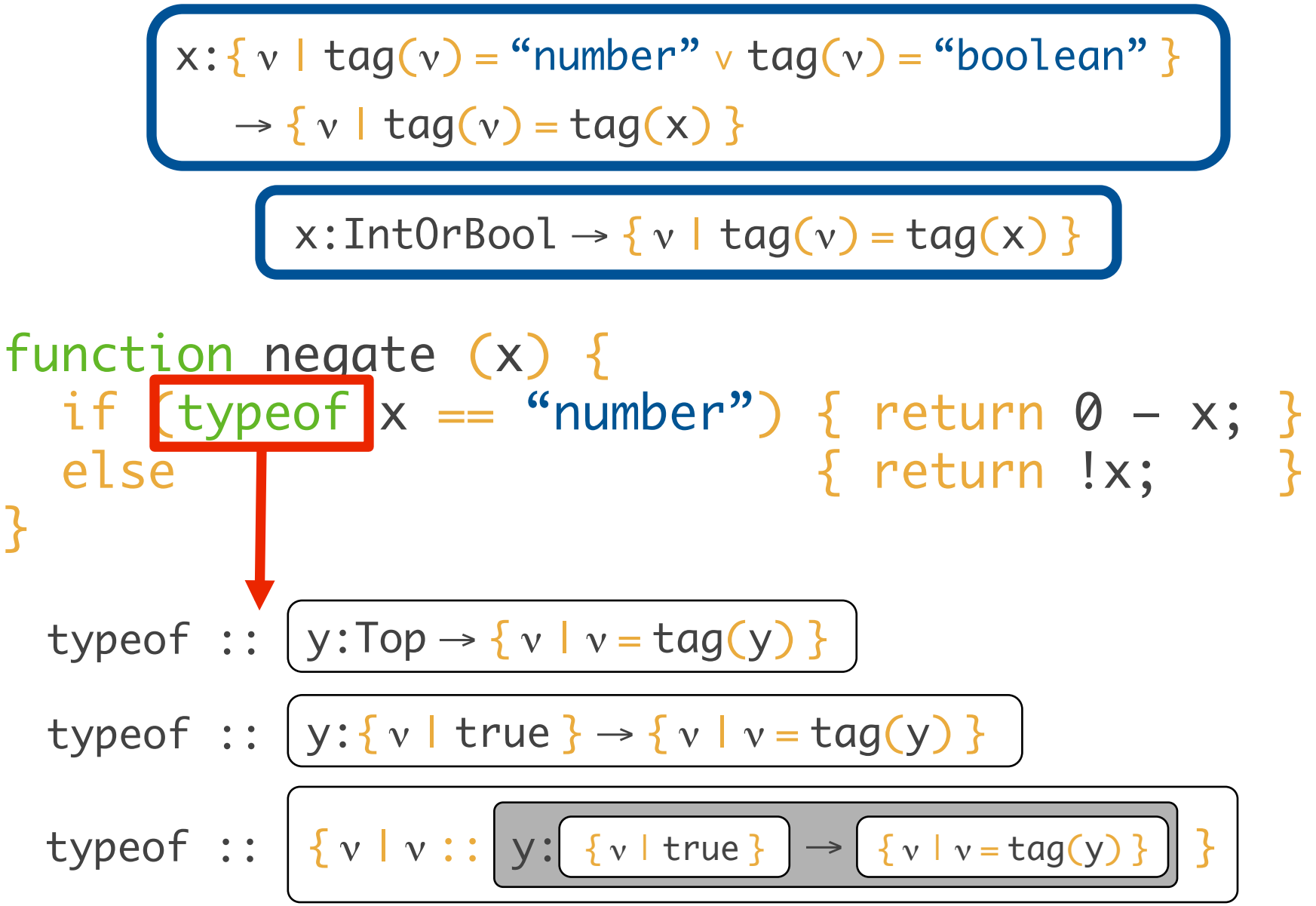
- All values described by refinement formulas

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

- "Has-type" predicate for arrows in formulas

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

- Can express several scripting idioms
- Automatic type checking
  - Decidable refinement logic
  - Subtyping = SMT Validity + Syntactic Subtyping


x:{ ν | tag(ν) = "number" ∨ tag(ν) = "boolean" }
→ { ν | tag(ν) = tag(x) }
x:IntOrBool → { ν | tag(ν) = tag(x) }
function negate (x) {
if (typeof x == "number") { return 0 - x; }
else { return !x; }
}
typeof :: y:Top → { ν | ν = tag(y) }
typeof :: y:{ ν | true } → { ν | ν = tag(y) }
typeof :: { ν | ν :: y:{ ν | true } → { ν | ν = tag(y) } }

$x : \text{IntOrBool} \rightarrow \{ v \mid \text{tag}(v) = \text{tag}(x) \}$ ✓

```
function negate (x) {
  if (typeof x == "number") { return 0 - x; }
  else                      { return !x;    }
}
```

$$x :: \{ v \mid \text{Int}(v) \}$$

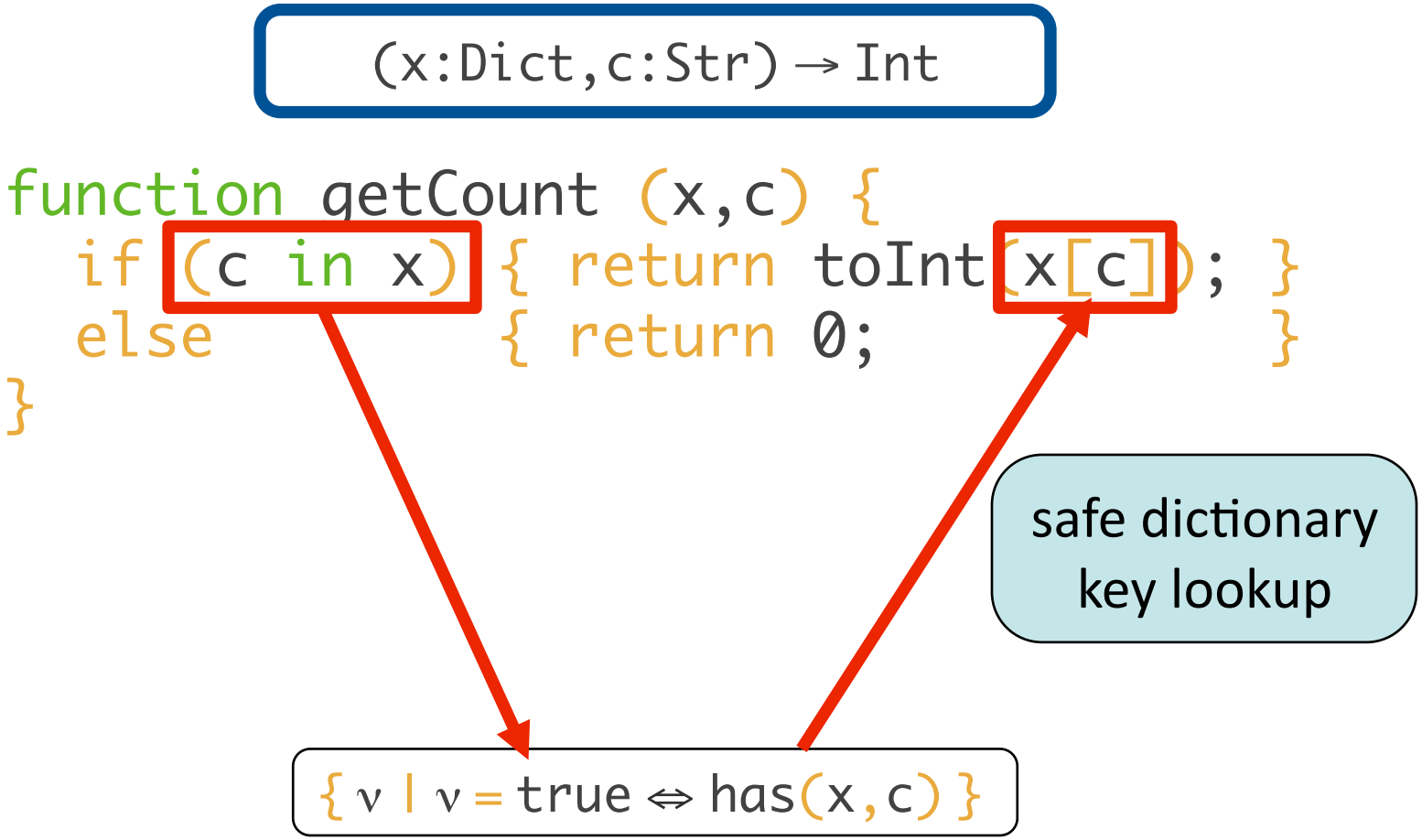
$$0 - x :: \{ v \mid \text{Int}(v) \}$$

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

```
function negate (x) {
  if (typeof x == "number") { return 0 - x; }
  else                      { return !x;    }
}
```

x :: { v | Bool(v) }

!x :: { v | Bool(v) }


(x:Dict,c:Str) → Int
function getCount (x,c) {
if (c in x) { return toInt(x[c]); }
else { return 0; }
}
safe dictionary key lookup
{ v | v = true ⇔ has(x,c) }

```
(x:Dict,c:Str) → Int
```

```
function getCount (x,c) {
  if (c in x) { return toInt(x[c]); }
  else        { return 0;           }
}
```

tag(sel(d,c)) = "Int"

```
(x:Dict,c:Str) → {ν | EqMod(ν,x,c) ⇒ Int(d[c])}
```

```
function incCount (x,c) {
  var i = getCount(x,c);
  return {x with c = i + 1};
}
```

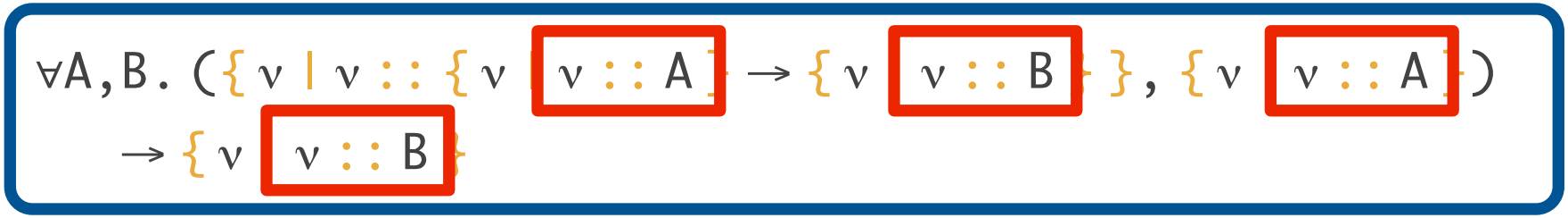
Adding Type Constructors

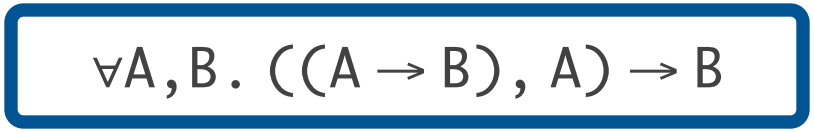
T ::= { v | p }

p ::= … | x :: U

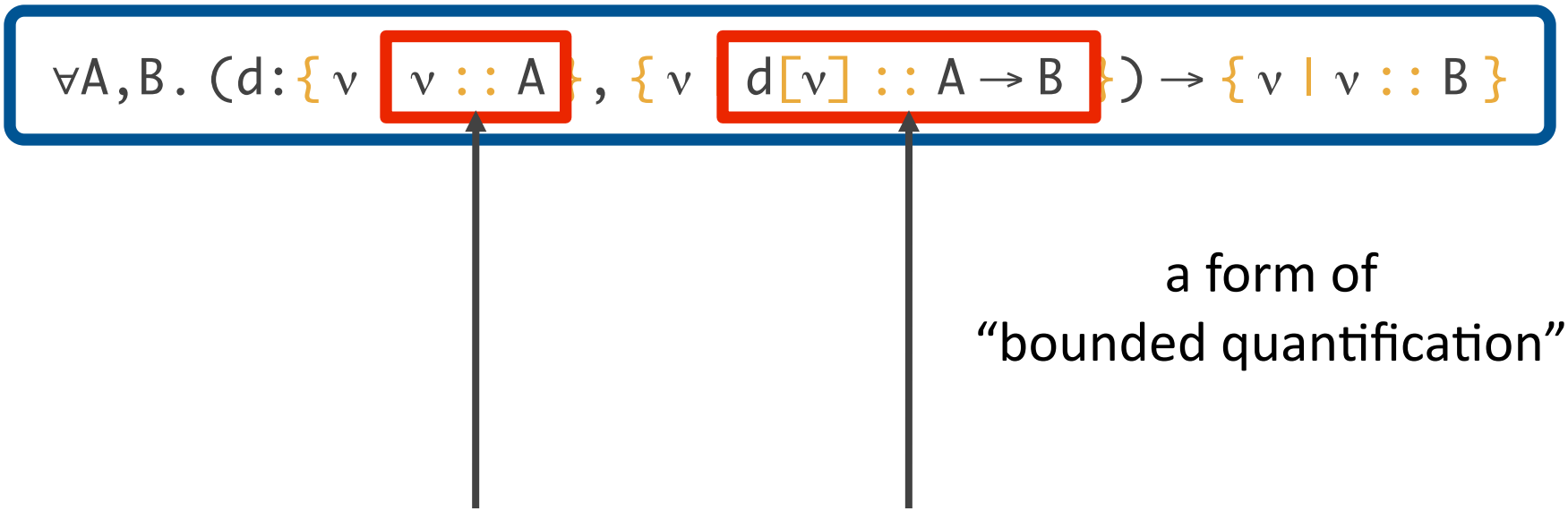
U ::= y:$T_1 \rightarrow T_2$ | A | List T | Null

```
function apply (f,x) { return f(x); }
```





```
function dispatch (d,f) { return d[f](d); }
```



d :: A but additional constraints on A

≈ ∀A <: {f: A → B}. d :: A

```
∀A,B.
 ({ν | ν :: A → B}, {ν | ν :: List[A]}) → {ν | ν :: List[B]}
```

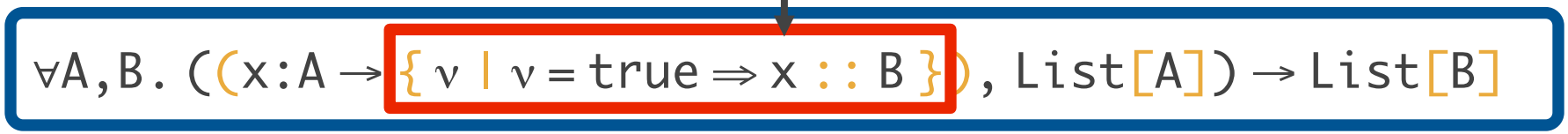
```
∀A,B. ((A → B), List[A]) → List[B]
```

```
function map (f,xs) {
  if (xs == null) { return null; }
  else {
    return f(xs["hd"]) :: map (f,xs["tl"]);
  }
}
```

encode recursive data as dictionaries

```
function filter (f,xs) {
  if (xs == null) { return null; }
  else if (f(xs["hd"])) {
    return xs["hd"] :: filter(f,xs["tl"]);
  } else {
    return filter(f,xs["tl"]);
  }
}
```

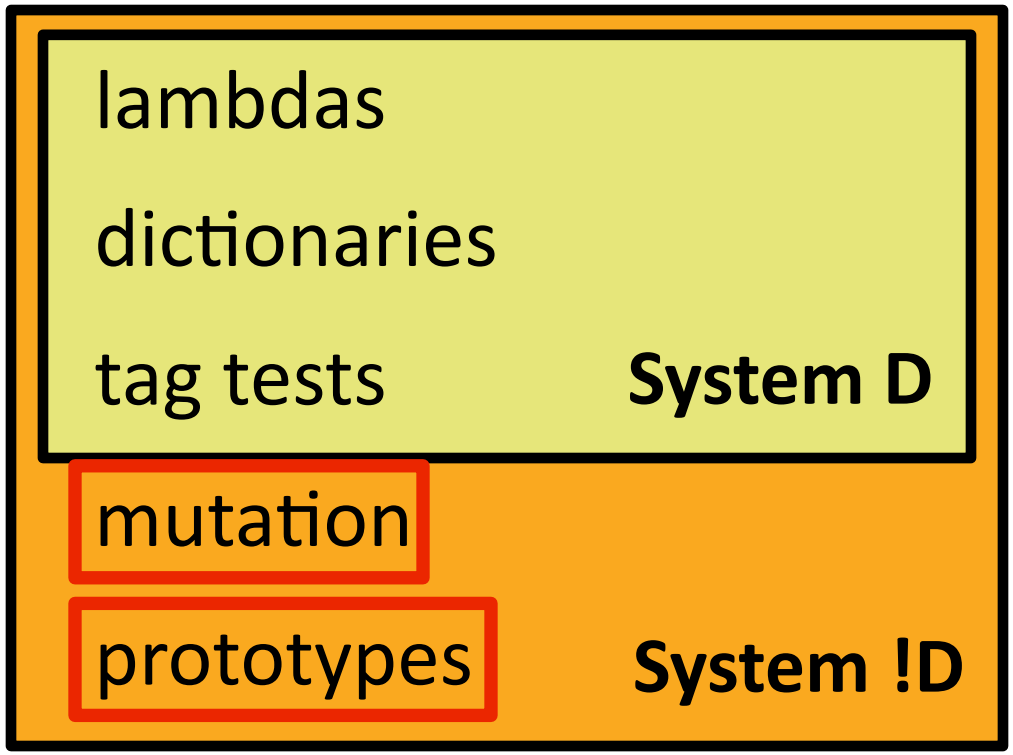
usual definition,
but an interesting type*



*Tobin Hochstadt and Felleisen (POPL 2008)

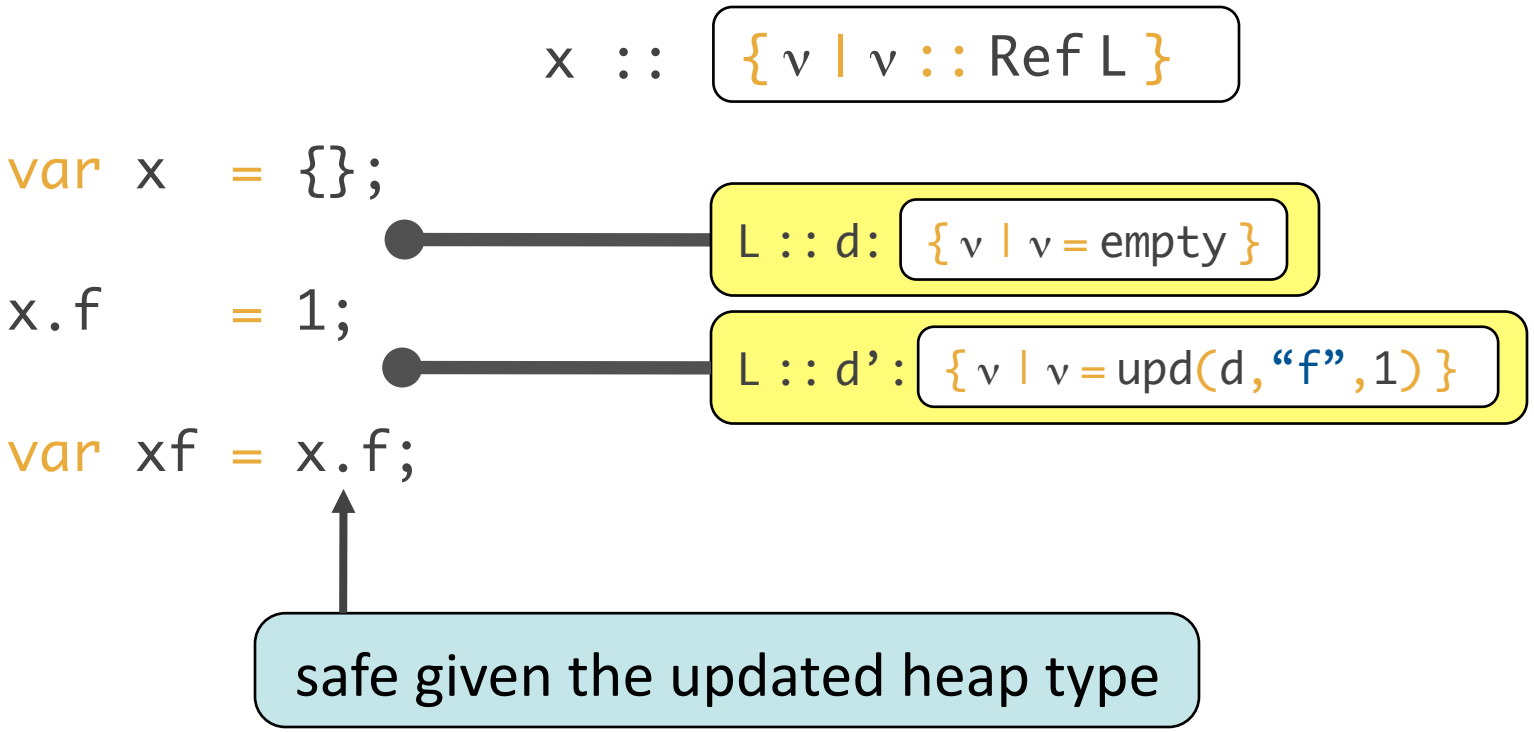
Crockford (JavaScript: The Good Parts)

```
Object.beget = function(o) {
  var F = function(o) { return this; };
  F.prototype = o;
  return new F();
}
```



Strong Update à la Alias Types

- Types are flow-insensitive (as usual)
- But heap types are flow-sensitive
  - Mappings from locations to types can change



Prototype Chain Unrolling

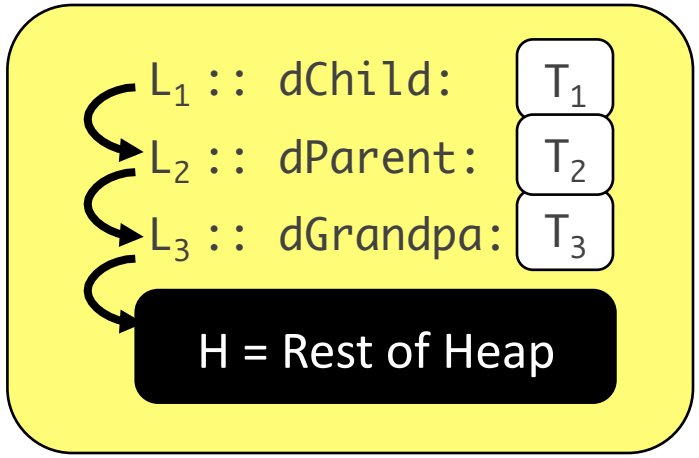
```
var granpda = …;
var parent  = beget(grandpa);
var child   = beget(parent);
```

grandpa :: $\{ \nu \mid \nu :: \text{Ref } L_3 \}$

parent :: $\{ \nu \mid \nu :: \text{Ref } L_2 \}$

child :: $\{ \nu \mid \nu :: \text{Ref } L_1 \}$

(k in child)



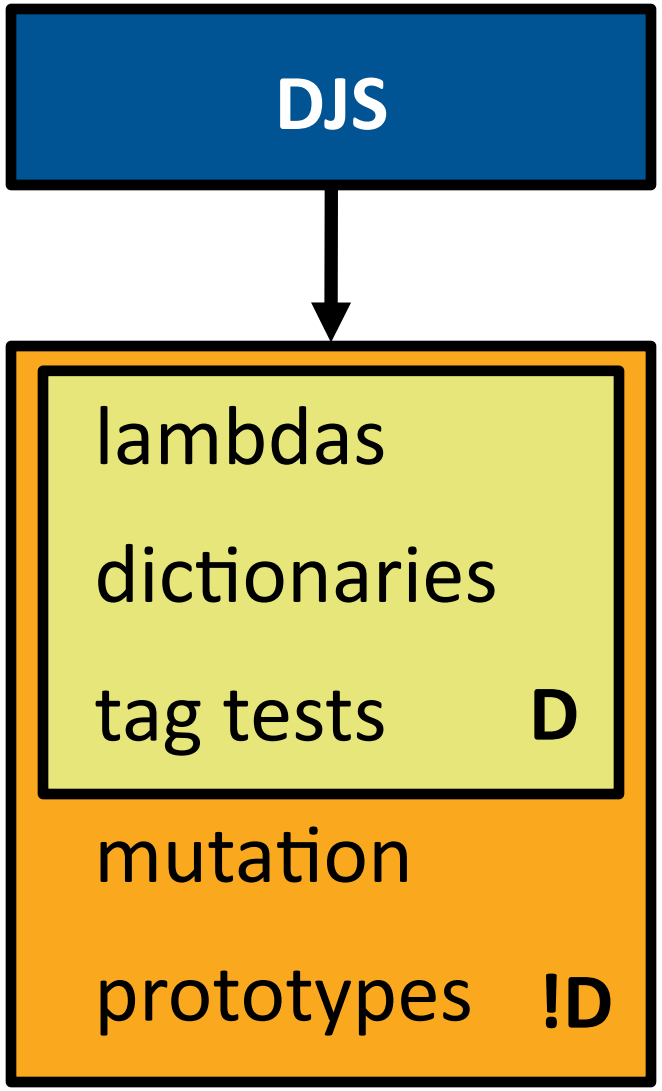
has(dChild,k) ∨

has(dParent,k) ∨

has(dGrandpa,k) ∨

HeapHas(H,L_4,k)

Recap / Future Work



Applications so far:

Inheritance patterns, JS: Good Parts

Future work:

Microbenchmarks (e.g. SunSpider)

Libraries (e.g. Array.prototype)

Application to Browser Extensions

ES5/ES6 Features

Class-based Inheritance

More Local Type Inference

Thanks!

Thanks to Brown PLT for the LambdaJS tools!

ravichugh.com/nested

Extra Slides

Constants

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

  mem  ::  d:Dict → k:Str → { ν | Bool(ν) ∧ ν = True ⇔ has(d,k) }
  get  ::  d:Dict → k:{ ν | Str(ν) ∧ has(d,ν) } → { ν | ν = sel(d,k) }
  set  ::  d:Dict → k:Str → x:Top → { ν | ν = upd(d,k,x) }
  rem  ::  d:Dict → k:Str → { ν | ν = upd(d,k,bot) }
```

Macros

- Types

$$\texttt{Int} \equiv \{ v \mid \texttt{tag}(v) = \text{“Int”} \}$$

$$x{:}T_1 \to T_2 \equiv \{ v \mid v :: x{:}T_1 \to T_2 \}$$

- Formulas

$$\texttt{Str}(x) \equiv \texttt{tag}(x) = \text{“Str”}$$

$$\texttt{has}(d,k) \equiv \texttt{sel}(d,k) \mathrel{!}= \texttt{bot}$$

$$\texttt{EqMod}(d,d',k) \equiv \forall k'.\ k' \mathrel{!}= k \Rightarrow \texttt{sel}(d,k) \mathrel{!}= \texttt{sel}(d',k)$$

- Logical Values

$$x.k \equiv \texttt{sel}(v, \text{“k”})$$

$$x[k] \equiv \texttt{sel}(v,k)$$

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 ::

$$
\begin{aligned}
&\forall A.\ \mathtt{callbacks}{:}\mathtt{List}[\mathtt{Top} \to \mathtt{Top}] \\
&\to \mathtt{f}{:}\{\nu \mid \mathtt{Str}(\nu) \vee \nu :: A \to \mathtt{Top}\} \\
&\to \mathtt{obj}{:}\{\nu \mid \nu :: A \\
&\qquad \wedge (\mathtt{f} = \mathtt{null} \Rightarrow \nu :: A \to \mathtt{Int}) \\
&\qquad \wedge (\mathtt{Str}(\mathtt{f}) \Rightarrow \nu[\mathtt{f}] :: A \to \mathtt{Int})\} \\
&\to \mathtt{List}[\mathtt{Top} \to \mathtt{Top}]
\end{aligned}
$$

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 ::
  callbacks:List[Top → Top]
  * f:{ g | Str(g) ∨ g :: { x | x = obj } → Top }
  * obj:{ o | (f = null ⇒ o :: { x | x = o } → Int)
              ∧ (Str(f) ⇒ o[f] :: { x | x = o } → Int) }
  → List[Top → Top]
```

Related Work

- TODO

Traditional vs. Nested Refinements

Approach: Refinement Types

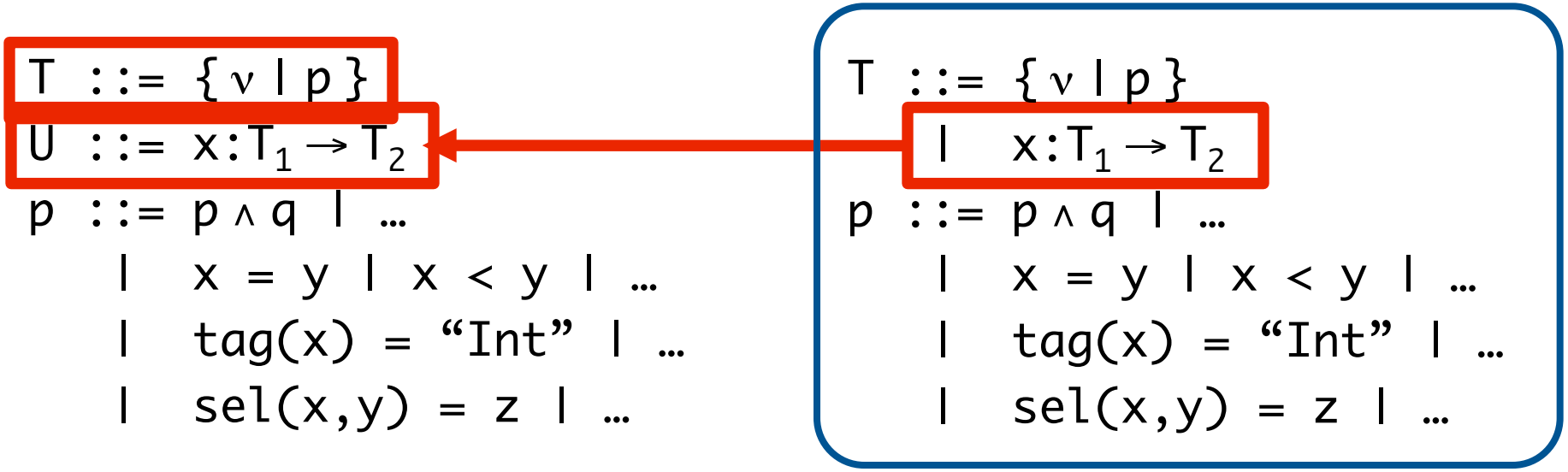
- Reuse refinement type architecture

- 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



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

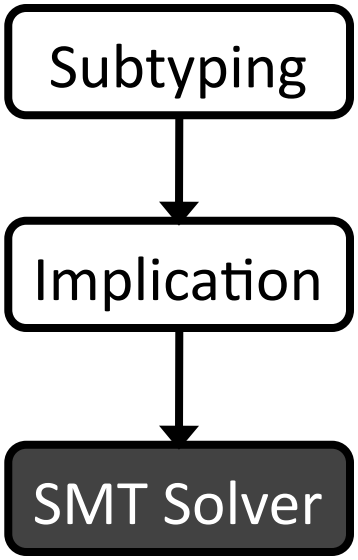
Nested Refinements

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

$$
\begin{array}{l}
T ::= \{ v \mid p \} \\
U ::= x{:}T_1 \rightarrow T_2 \\
p ::= p \wedge q \mid \ldots \\
\quad \mid \; x = y \mid x < y \mid \ldots \\
\quad \mid \; \mathtt{tag}(x) = \text{“Int”} \mid \ldots \\
\quad \mid \; \mathtt{sel}(x,y) = z \mid \ldots \\
\quad \mid \; x :: U
\end{array}
$$

Subtyping

Subtyping



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

$T ::= \{ \nu \mid p \}$
$\quad\quad | \quad x{:}T_1 \rightarrow T_2$

traditional refinements

$\texttt{Int} \equiv \{ \nu \mid \texttt{tag}(\nu)=\text{“Int”} \}$

$\texttt{Top} \equiv \{ \nu \mid \texttt{true} \}$

Subtyping

Subtyping

Implication

SMT Solver

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

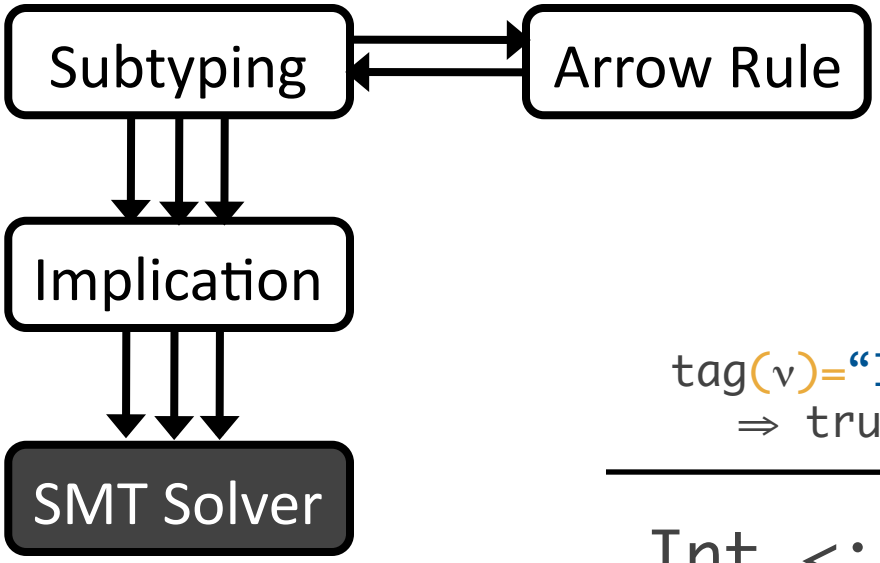
$$T ::= \{ \nu \mid p \} \mid x{:}T_1 \rightarrow T_2$$

traditional refinements

$$\texttt{Int} \equiv \{ \nu \mid \texttt{tag}(\nu)=\text{“Int”} \}$$

$$\texttt{Top} \equiv \{ \nu \mid \texttt{true} \}$$

Subtyping



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

$$\texttt{Top} \rightarrow \texttt{Int} <: \texttt{Int} \rightarrow \texttt{Int}$$

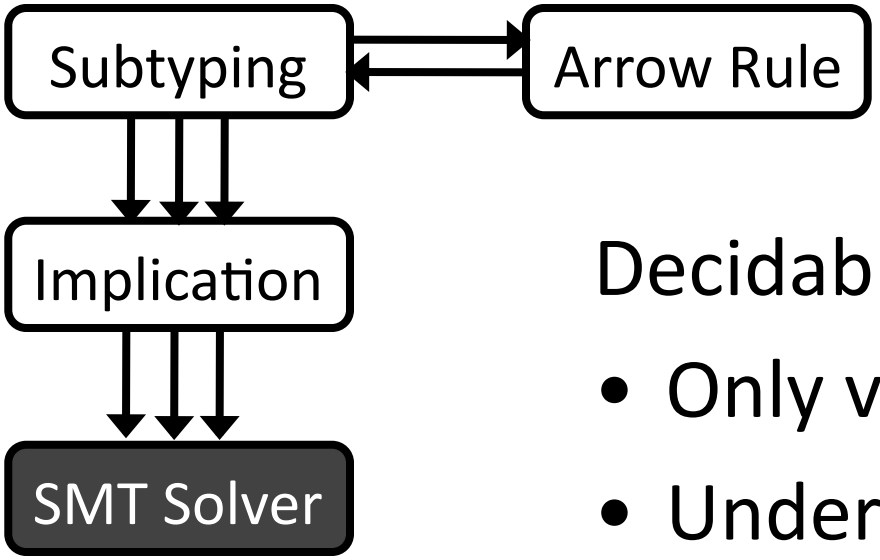
$$T ::= \{ \nu \mid p \} \mid x{:}T_1 \rightarrow T_2$$

traditional refinements

$$\texttt{Int} \equiv \{ \nu \mid \texttt{tag}(\nu)=\text{“Int”} \}$$

$$\texttt{Top} \equiv \{ \nu \mid \texttt{true} \}$$

Subtyping



Decidable if:

- Only values in formulas
- Underlying theories decidable

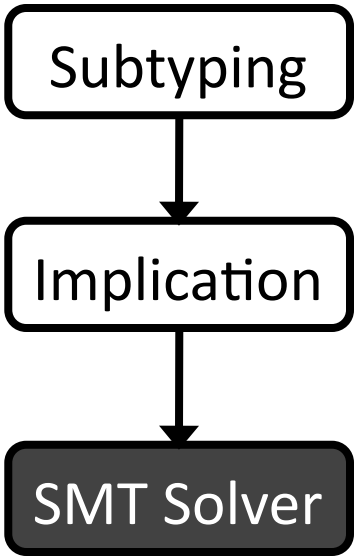
With nested refinements:

- No new theories
- But implication is imprecise!

$$T ::= \{ v \mid p \} \quad | \quad x{:}T_1 \rightarrow T_2$$

traditional refinements

Subtyping with Nesting



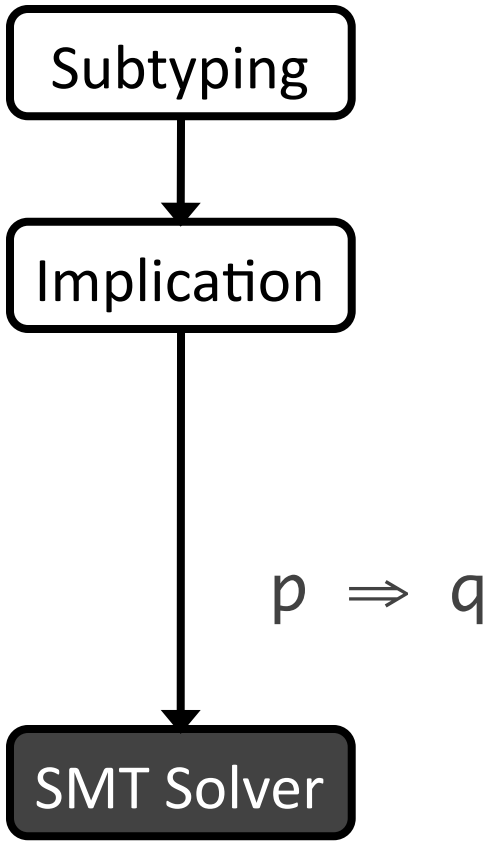
$\nu :: \texttt{Top} \rightarrow \texttt{Int} \not\Rightarrow \nu :: \texttt{Int} \rightarrow \texttt{Int}$

Invalid, as these are distinct uninterpreted constants

Subtyping with Nesting

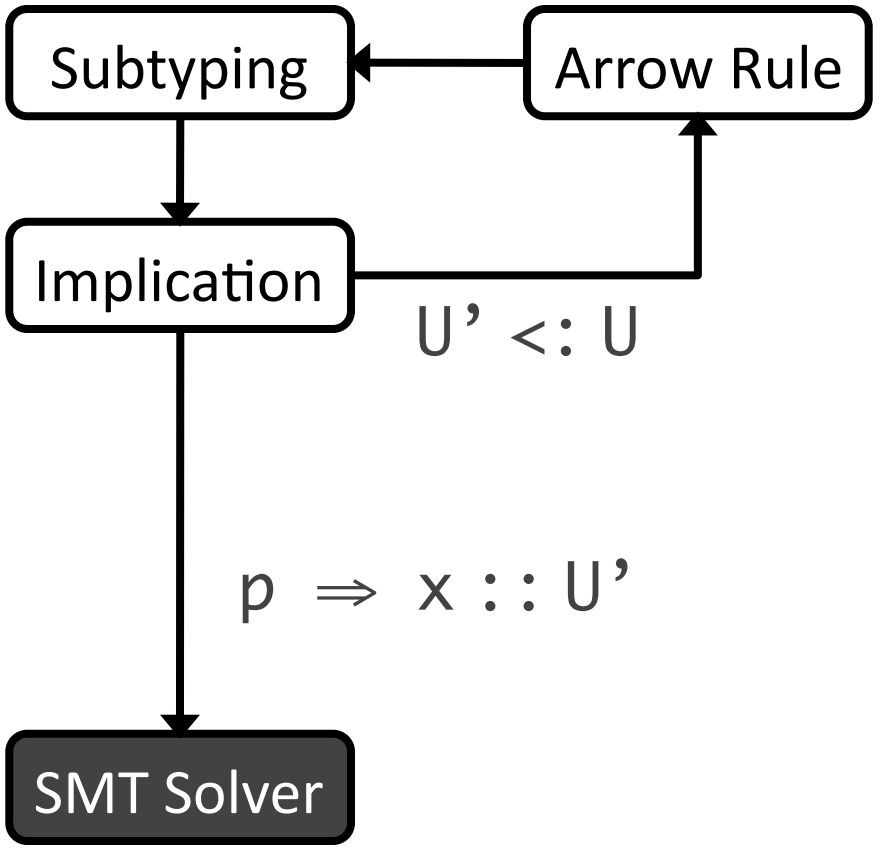
When goal is base predicate:

$p \Rightarrow q$

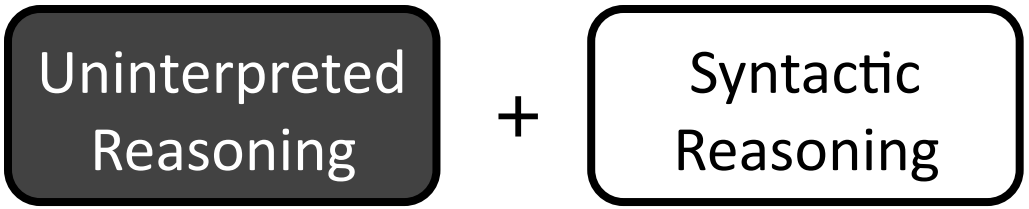


When goal is “has-type” predicate:

$p \Rightarrow x :: U$



Subtyping with Nesting



$$\frac{\texttt{p} \Rightarrow \nu :: \texttt{Top} \rightarrow \texttt{Int} \qquad \texttt{Top} \rightarrow \texttt{Int} <: \texttt{Int} \rightarrow \texttt{Int}}{\texttt{p} \Rightarrow \nu :: \texttt{Int} \rightarrow \texttt{Int}}$$

Normalize formulas to

subdivide obligations appropriately

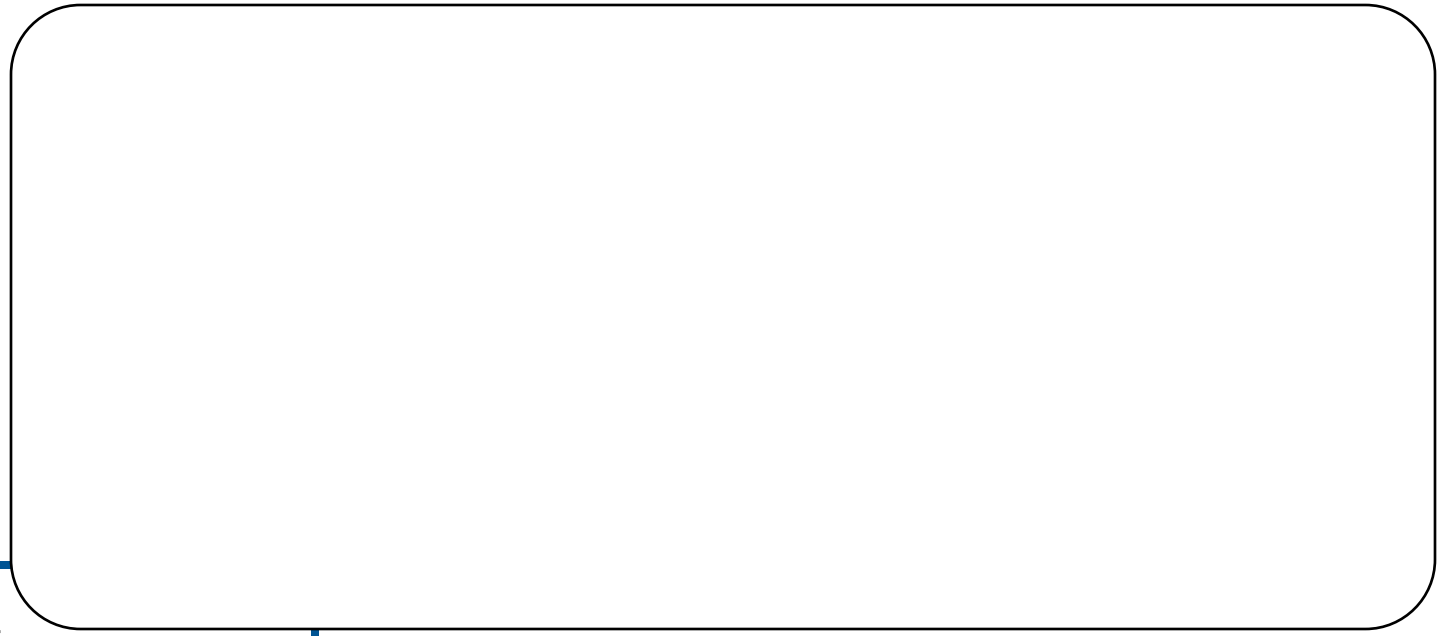
Normalization

- TODO

foo example

```
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
```

foo ::



$T ::= \{ v \mid p \}$
$U ::= x{:}T_1 \rightarrow T_2$
$p ::= p \wedge q \mid \ldots$
$\quad \mid \; x = y \mid x < y \mid \ldots$
$\quad \mid \; tag(x) = \text{“Int”} \mid \ldots$
$\quad \mid \; sel(x,y) = z \mid \ldots$
$\quad \mid \; x :: U$

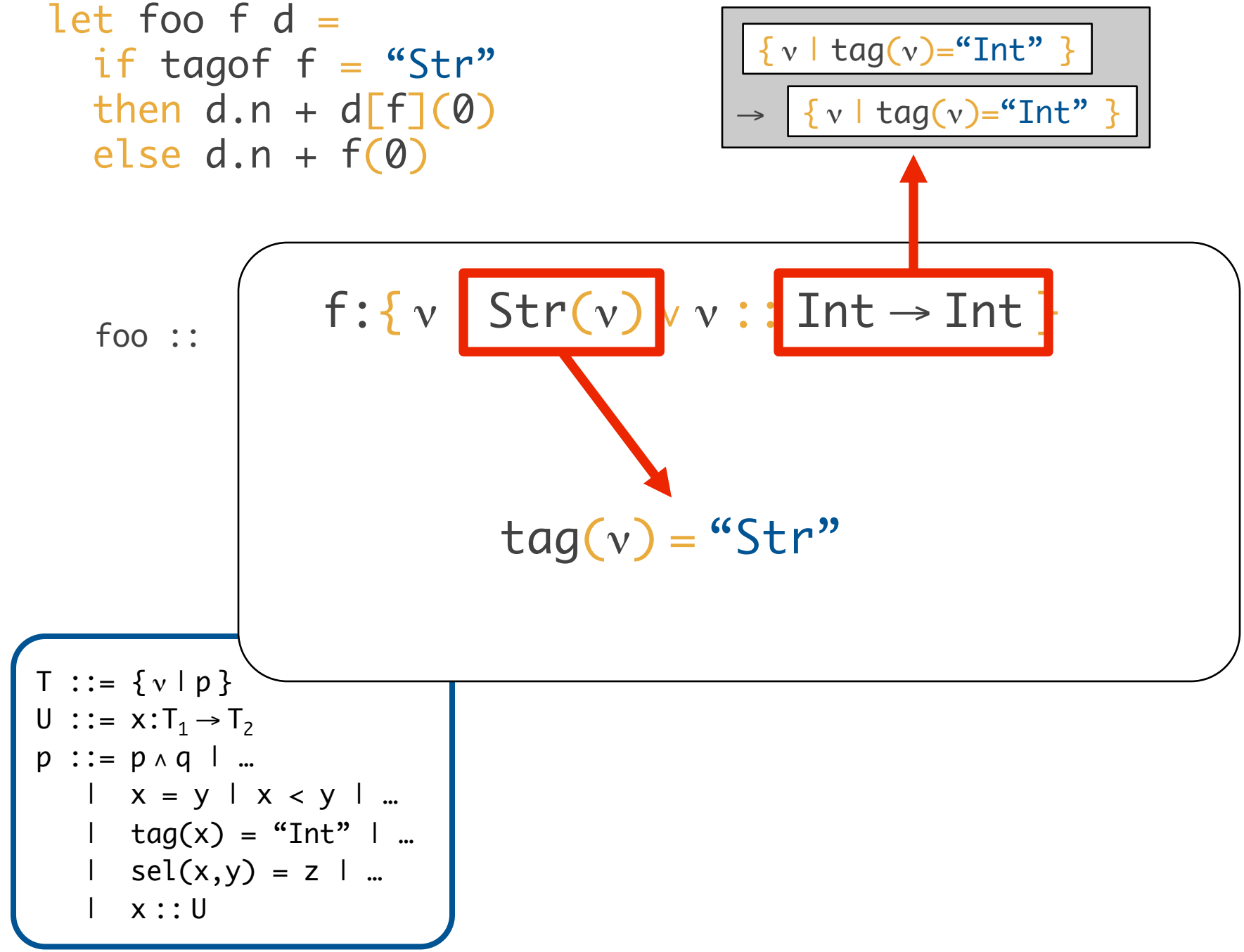
```
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
```

foo ::

$$f:\{\nu \mid \mathrm{Str}(\nu)\} \wedge \nu :: \mathrm{Int} \to \mathrm{Int}$$

$$
\begin{aligned}
T &::= \{\nu \mid p\} \\
U &::= x{:}T_1 \to T_2 \\
p &::= p \wedge q \mid \ldots \\
&\quad \mid x = y \mid x < y \mid \ldots \\
&\quad \mid \mathrm{tag}(x) = \text{“Int”} \mid \ldots \\
&\quad \mid \mathrm{sel}(x,y) = z \mid \ldots \\
&\quad \mid x :: U
\end{aligned}
$$

```
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
```




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

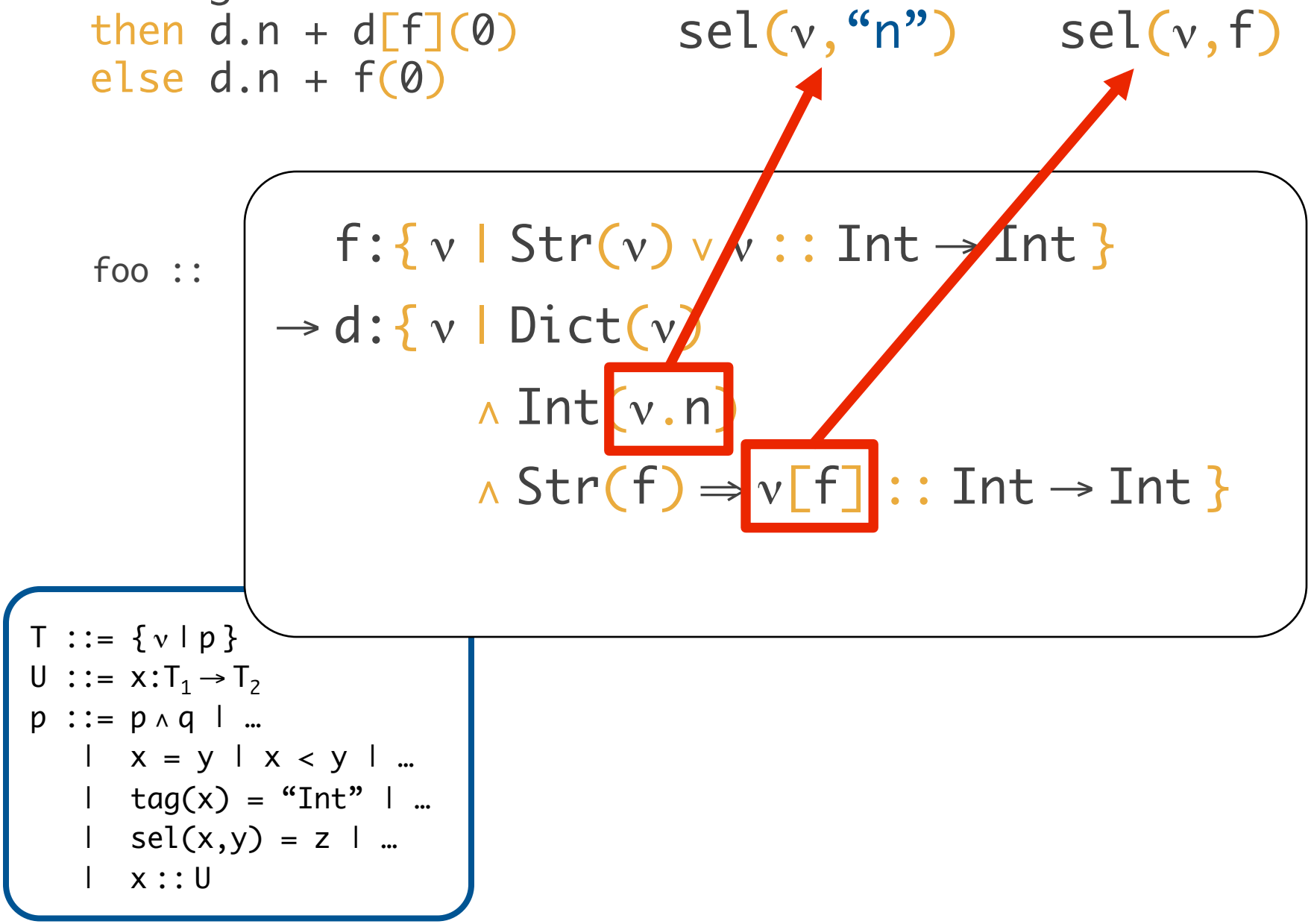
```
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
```

foo ::

$$f:\{\nu \mid Str(\nu) \vee \nu :: Int \to Int\}$$
$$\to d:\{\nu \mid Dict(\nu)$$
$$\wedge\ Int(\nu.n)$$
$$\wedge\ Str(f) \Rightarrow \nu[f] :: Int \to Int\}$$

$T ::= \{\nu \mid p\}$
$U ::= x{:}T_1 \to T_2$
$p ::= p \wedge q \mid \ldots$
$\quad \mid\ x = y \mid x < y \mid \ldots$
$\quad \mid\ tag(x) = \text{“Int”} \mid \ldots$
$\quad \mid\ sel(x,y) = z \mid \ldots$
$\quad \mid\ x :: U$

```
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
```




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

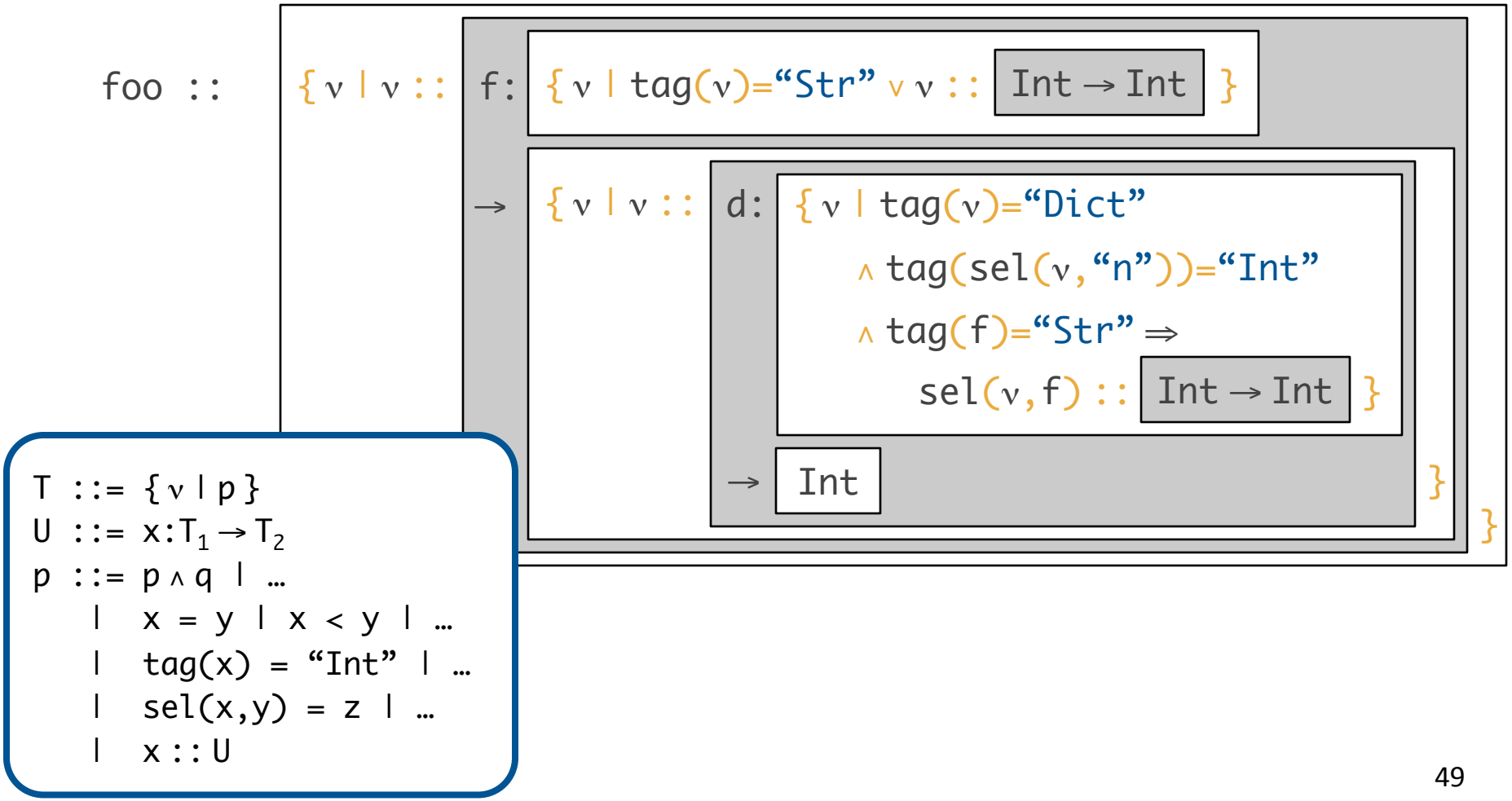
```
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
```

foo ::

$$
\begin{aligned}
& f:\{\nu \mid \mathrm{Str}(\nu) \vee \nu :: \mathrm{Int} \to \mathrm{Int}\} \\
\to\ & d:\{\nu \mid \mathrm{Dict}(\nu) \\
& \qquad \wedge\ \mathrm{Int}(\nu.n) \\
& \qquad \wedge\ \mathrm{Str}(f) \Rightarrow \nu[f] :: \mathrm{Int} \to \mathrm{Int}\} \\
\to\ & \boxed{\mathrm{Int}}
\end{aligned}
$$

$$
\begin{aligned}
T &::= \{\nu \mid p\} \\
U &::= x{:}T_1 \to T_2 \\
p &::= p \wedge q \mid \ldots \\
&\quad \mid\ x = y \mid x < y \mid \ldots \\
&\quad \mid\ \mathrm{tag}(x) = \text{“Int”} \mid \ldots \\
&\quad \mid\ \mathrm{sel}(x,y) = z \mid \ldots \\
&\quad \mid\ x :: U
\end{aligned}
$$

```
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
```




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