Nested Refinement Types for Dynamic Languages

Ravi Chugh
What are “Dynamic” Languages?

Untyped, and characterized by common features

Reflective type-tests

```
typeof x == ‘number’
```

Dictionary-style objects

```
obj[key] = val
```

“Duck typing”

```
if (duck.quack)
  { duck.quack() }
```

Dynamic code generation

```
eval(‘var x = 1’) 
```

Rich string primitives

```
arr.join(‘,’)
```
Why Study Dynamic Languages?

<table>
<thead>
<tr>
<th>Count</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>235,334</td>
<td>C#</td>
</tr>
<tr>
<td>175,865</td>
<td>Java</td>
</tr>
<tr>
<td>147,557</td>
<td>JavaScript</td>
</tr>
<tr>
<td>95,609</td>
<td>C++</td>
</tr>
<tr>
<td>80,941</td>
<td>Python</td>
</tr>
<tr>
<td>44,447</td>
<td>C</td>
</tr>
<tr>
<td>33,733</td>
<td>Ruby</td>
</tr>
<tr>
<td>4,652</td>
<td>Haskell</td>
</tr>
<tr>
<td>1,265</td>
<td>Scheme</td>
</tr>
<tr>
<td>691</td>
<td>OCaml</td>
</tr>
</tbody>
</table>

http://stackoverflow.com/tags
11-11-11
Why Study Dynamic Languages?

Pervasive, thanks to the web

• No static checking $\Rightarrow$ rapid prototyping
• String libraries + code gen $\Rightarrow$ multi-language systems

Important, thanks to the web

• Large applications written in these languages
• Reliability matters: security, availability, extensibility
• Performance matters: the new Browser Wars
• No static checking $\Rightarrow$ reliability and performance is hard

JavaScript is at the heart of this
Isn’t JavaScript a Terrible Language?

- Undefined value
- Scope manipulation
- Primitive type manipulation
- Implicit coercion
- Weird comparisons:
  - `NaN != NaN`
  - `Infinity == Infinity`
  - `'?,?,?' == new Array(4)`
- Implicit updates to global object
- Misleading syntax
- Implicit coercion
- Var lifting
- Lambda function: `\( \lambda \{
\} \)`
- Super scope manipulation
- `typeof x := 1`
Isn’t JavaScript Awesome?!

Core consists of bread-and-butter features

These features not going away, nor should they*
Found in many languages, typed and untyped
Lambdas in a mainstream language!

Core features are difficult ⇒ good for research

Core features are expressive ⇒ good for practice

JavaScript is evolving
System D lambdas, dictionaries, tag-tests

+ explicit references

+ prototype-based inheritance

Dependent JavaScript translates to D++

Applications in DJS
Research Plan

1. System D lambdas, dictionaries, tag-tests

2. + explicit references
   + prototype-based inheritance
   Dependent JavaScript translates to D++

3. Applications in DJS
1. **System D** lambdas, dictionaries, tag-tests

2. + explicit references
   - + prototype-based inheritance

3. **Applications in DJS**

Dependent JavaScript translates to D++
Motivating Example

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

- **tag-tests**: affect control flow
- **dictionaries**: indexed by string literals or arbitrary values
- **first-class functions**: can appear inside dictionaries
Approach: Refinement Types

tag-tests

Type environment tracks control flow predicates

\[
\begin{align*}
\text{if } \text{tagof } x = \text{“Int”} \text{ then } 0 - x \text{ else not } x
\end{align*}
\]
Approach: Refinement Types

d.n + d[m]

d :: \{ ν | \text{tag}(ν)="Dict" \\
\land \text{tag}(\text{sel}(ν,"n"))="Int" \\
\land \text{tag}(\text{sel}(ν,m))="Int" \}

McCarthy axioms

∀d,k,k’,x. k≠k’ ⇒

\text{sel}(\text{upd}(d,k,x),k) = x

\text{sel}(\text{upd}(d,k,x),k’) = \text{sel}(d,k’)

\text{sel}(\text{empty},k) = \text{bot}
Approach: Refinement Types

tag-tests
dictionaries
first-class functions

\[ 1 + f(0) \]

\[
f :: \quad x: \{ \nu \mid \text{tag}(\nu) = \text{“Int”} \} \rightarrow \{ \nu \mid \text{tag}(\nu) = \text{“Int”} \}
\]

\[
\lambda x.x :: \quad x: \{ \nu \mid \text{tag}(\nu) = \text{“Int”} \} \rightarrow \{ \nu \mid \nu = x \}
\]

Clear separation of base and arrow types

\[
T ::= \{ \nu \mid p \}
\]

\[
| \quad x:T_1 \rightarrow T_2
\]
Approach: Refinement Types

- tag-tests
- dictionaries
- first-class functions

\[ 1 + f(0) \]
Approach: Refinement Types

1 + d[f](∅)

\[ d :: \{ v \mid \text{tag}(v) = \text{"Dict"}, \text{sel}(v, f) \} \]

Key Challenges

1. How to describe arrow inside formula?
2. How to keep type checking decidable?
• Reuse refinement type architecture

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

• Define nested refinement type architecture
Nested Refinements

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

\[
d :: \{ \nu | \text{tag(\nu)} = \text{"Dict"} \}
\land \text{sel(\nu, f)} :: \{ \nu | \text{tag(\nu)} = \text{"Int"} \}
\rightarrow \{ \nu | \text{tag(\nu)} = \text{"Int"} \}
\]

uninterpreted predicate
“\(x :: U\)” says
“\(x\) has-type \(U\)”

uninterpreted constant in the logic...

... but syntactic arrow in the type system!
Nested Refinements

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

\[
\begin{align*}
T & ::= \{ \nu \mid p \} \\
U & ::= x : T_1 \rightarrow T_2 \\
p & ::= p \wedge q \mid \ldots \\
& \quad \mid x = y \mid x < y \mid \ldots \\
& \quad \mid \text{tag}(x) = \text{"Int"} \mid \ldots \\
& \quad \mid \text{sel}(x,y) = z \mid \ldots \\
\end{align*}
\]

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

\[
T ::= \{ \nu \mid p \} \\
U ::= x : T_1 \rightarrow T_2 \\
p ::= p \land q \mid \ldots \\
x = y \mid x < y \mid \ldots \\
tag(x) = \text{“Int”} \mid \ldots \\
sel(x, y) = z \mid \ldots \\
x :: U
\]

\[
T ::= \{ \nu \mid p \} \\
\mid x : T_1 \rightarrow T_2 \\
p ::= p \land q \mid \ldots \\
x = y \mid x < y \mid \ldots \\
tag(x) = \text{“Int”} \mid \ldots \\
sel(x, y) = z \mid \ldots 
\]

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

\[
\begin{align*}
T & ::= \{ \lor \mid p \} \\
U & ::= x : T_1 \rightarrow T_2 \\
p & ::= p \land q \mid \ldots \\
& \mid x = y \mid x < y \mid \ldots \\
& \mid \text{tag}(x) = \text{“Int”} \mid \ldots \\
& \mid \text{sel}(x,y) = z \mid \ldots \\
& \mid x :: U
\end{align*}
\]
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)

foo ::

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:{ ν | Str(ν) } ν :: Int -> Int

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

foo ::

f:{ν | Str(ν) ⇔ ν :: Int → Int}
→ d:{ν | Dict(ν)}

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)

T ::= \{\nu | p\}
U ::= x:T_1 \to T_2
p ::= p \land q \mid \ldots
  \mid x = y \mid x < y \mid \ldots
  \mid \text{tag}(x) = "Int" \mid \ldots
  \mid \text{sel}(x,y) = z \mid \ldots
  \mid x :: U

\[ f : \{\nu | \text{Str}(\nu) \lor \nu :: \text{Int} \to \text{Int}\} \]
\[ \to d : \{\nu | \text{Dict}(\nu) \land \text{Int}(\nu.n) \land \text{Int}(\nu.n) \land \text{Str}(f) \Rightarrow \nu[f] :: \text{Int} \to \text{Int}\} \]
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)

foo ::

\[
\begin{align*}
  f & : \{ \nu \mid \text{Str}(\nu) \lor \nu :: \text{Int} \rightarrow \text{Int} \} \\
  \rightarrow d & : \{ \nu \mid \text{Dict}(\nu) \\
  & \land \text{Int}(\nu.n) \\
  & \land \text{Str}(f) \Rightarrow \nu[f] :: \text{Int} \rightarrow \text{Int} \} \\
  \rightarrow \text{Int}
\end{align*}
\]

\[
\begin{align*}
  T & ::= \{ \nu \mid p \} \\
  U & ::= x : T_1 \rightarrow T_2 \\
  p & ::= p \land q \mid \ldots \\
  x & = y \mid x < y \mid \ldots \\
  \text{tag}(x) & = \text{"Int"} \mid \ldots \\
  \text{sel}(x,y) & = z \mid \ldots \\
  x & :: U
\end{align*}
\]
let foo f d = 
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)
Nested Refinements

• Type Language

• Subtyping

• Extensions

• Recap
Subtyping

\[ T ::= \{ \nu \mid p \} \]

\[ \mid x : T_1 \rightarrow T_2 \]

\[ \text{traditional refinements} \]

\[ \text{tag(\nu)} = \text{“Int”} \Rightarrow \text{true} \]

\[ \text{Int <: Top} \]

\[ \text{Int} = \{ \nu \mid \text{tag(\nu)} = \text{“Int”} \} \]

\[ \text{Top} = \{ \nu \mid \text{true} \} \]
Subtyping

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

traditional refinements

Implication

SMT Solver

Subtyping

\[ \{ \nu \} \]
\[ \equiv \{ \nu \mid \text{tag}(\nu) = \text{"Int"} \} \]
\[ \text{Int} \equiv \text{Top} \rightarrow \text{Int} \]
\[ \text{Int} <: \text{Int} \rightarrow \text{Int} \]
\[ \text{tag}(\nu) = \text{"Int"} \Rightarrow \text{true} \]
\[ \Rightarrow \text{tag}(\nu) = \text{"Int"} \]

\[ \text{Int} <: \text{Top} \]

\[ \text{Top} \rightarrow \text{Int} <: \text{Int} \rightarrow \text{Int} \]
Subtyping

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

traditional refinements

\[ \text{Int} = \{ \nu \mid \text{tag}(\nu) = \text{"Int"} \} \]

\[ \text{Top} = \{ \nu \mid \text{true} \} \]

\[ \text{tag}(\nu) = \text{"Int"} \Rightarrow \text{true} \]

\[ \text{tag}(\nu) = \text{"Int"} \Rightarrow \text{tag}(\nu) = \text{"Int"} \]

\[ \text{Int} <: \text{Top} \]

\[ \text{Int} <: \text{Int} \]

\[ \text{Top} \rightarrow \text{Int} <: \text{Int} \rightarrow \text{Int} \]
Subtyping

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

Decidable if:
- Only values in formulas
- Underlying theories decidable

With nested refinements:
- No new theories
- But implication is imprecise!

traditional refinements
Subtyping with Nesting

Invalid, as these are distinct uninterpreted constants
Subtyping with Nesting

When goal is base predicate:
\[ p \Rightarrow q \]

Subtyping

Implication

\[ p \Rightarrow q \]

SMT Solver

When goal is “has-type” predicate:
\[ p \Rightarrow x :: U \]

Subtyping

Arrow Rule

Implication

\[ U' <: U \]

\[ p \Rightarrow x :: U' \]

SMT Solver
Subtyping with Nesting

Uninterpreted Reasoning + Syntactic Reasoning

\[ p \Rightarrow \nu :: \text{Top} \rightarrow \text{Int} \]

\[ \text{Top} \rightarrow \text{Int} \prec \text{Int} \rightarrow \text{Int} \]

\[ p \Rightarrow \nu :: \text{Int} \rightarrow \text{Int} \]

Normalize formulas to subdivide obligations appropriately
Nested Refinements

- Type Language
- Subtyping
- Extensions
- Recap
Extensions

- Simple to add additional type constructors
- Extend the grammar of type terms

\[ U ::= x:T_1 \rightarrow T_2 \mid A \mid \text{List}[T] \mid \text{Null} \]

- Add additional syntactic subtyping rules

Syntactic Rules
- Arrow Rule
- Covariant List Rule
- Null List Rule
∀A,B.
{ν | ν :: A → B} → {ν | ν :: List[A]} → {ν | ν :: List[B]}

let map f xs =
  if xs = null then null
  else f xs[“hd”] :: map f xs[“tl”]

encode recursive data as dictionaries
let filter f xs = 
  if xs = null then 
    null 
  else if f xs["hd"] then 
    xs["hd"] :: filter f xs["tl"] 
  else 
    filter f xs["tl"] 

usual definition, but an interesting type

∀A,B. \{ \nu \mid \nu :: x:A \rightarrow \{ \nu \mid \nu = \text{True} \Rightarrow x :: B \} \} 

\rightarrow \{ \nu \mid \nu :: \text{List}[A] \} 

\rightarrow \{ \nu \mid \nu :: \text{List}[B] \}
let dispatch d f = d[f] d

∀A,B. d:{ν | Dict(ν) ∧ ν :: A} → {ν | Str(ν) ∧ d[ν] :: A → B } → {ν | ν :: B }

a form of “bounded quantification” since d :: A but additional constraints on A
Recap

• Refinement types are a compelling approach
  – Dynamic dictionaries require dependency
  – Tag-tests require path sensitivity
• But, not enough for lambdas in dictionaries
• Nested refinement types are a clean solution
  – Natural way to describe dynamic idioms
  – Novel subtyping remains decidable and automatic
• Interesting soundness proof
Research Plan

1. System D lambdas, dictionaries, tag-tests

2. + explicit references
   + prototype-based inheritance

Dependent JavaScript translates to D++

3. Applications in DJS

D + references + prototypes
Research Plan

1. **System D** lambdas, dictionaries, tag-tests

2. + explicit references
   - + prototype-based inheritance

3. Applications in DJS

Dependent JavaScript translates to D++
System D

let x = {} in
let x' = x with "f" = 1 in
x'.f

JavaScript

var x = {};
x.f = 1;
x.f

functional update

x :: \{ v | v = empty \}
x' :: \{ v | v = upd(x, "f", 1) \}

mutation!
System !D = D + Explicit References

allocate cell  \text{ref } x
dereference  \text{!}x
update cell  \text{} := y

\(X\) stores value of type \(\{ \nu \mid \text{Dict}(\nu) \}\)
Update to reference cell doesn’t affect type
So this dictionary read does not type check!

JavaScript

```
var x = {}
x.f = 1
```

System !D

```
let x = ref {} in
let _ = x := (!x with "f" = 1) in
let _ = (!x)["f"]
```

need to strongly update reference type
• In JS, **every** dictionary is stored in a reference

• No strong update ⇒ cannot extend dictionary!

• So, we must support flow-sensitive invariants

```python
var x = {};  
x.f = 1;  
x.f
```

```plaintext
let x = ref {} in
let _ = x := (!x with "f" = 1) in
let _ = (!x)["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

\[
\begin{align*}
U & ::= \ldots \mid \text{Ref } L \\
L &: d: \{ \nu \mid \nu = \text{empty} \} \\
L &: d': \{ \nu \mid \nu = \text{upd}(d, "f", 1) \}
\end{align*}
\]

```plaintext
let x = ref {} in
let _ = x := (!x with "f" = 1) in
let _ = (!x)["f"]
```

safe given the updated heap type
One Last Feature: Prototypes

• Each object is backed by a prototype object
  – Can be any ordinary object
  – Immutable link set at construction time*

• Object read  \( x[k] \)
  – If \( x \) has key \( k \), return the binding
  – Else, recurse on \( x.__proto__[k] \)

• Object write  \( x[k] = y \)
  – does not affect prototype chain
```javascript
var grandpa = {
  "surname" = "Quackenboss",
  "saying"  = "When I was your age…",
  "age"     = 77
}
var parent  = { "age" = 44 }
var child   = { "age" = 22 }

Assume proto chain is:

```
var a = child.age
```

1. does child have age?

2. what is type of child.age?

```
a :: { ν | Int(ν) }
```

```
a :: { ν | ν = 22 v v = 44 v v = 77 }
```

```
a :: { ν | ν = 22 }
```
```
• Track prototype links in heap type
  – immutable and acyclic

• New uninterpreted predicate $\text{ObjHas}(H, L, k)$
  – analog to $\text{has}(d, k)$ (macro for $\text{sel}(d, k) \neq \text{bot}$)

• New uninterpreted function $\text{ObjSel}(H, L, k)$
  – analog to $\text{sel}(d, k)$
Checking if child has “age”

\[
\text{ObjHas}(H, L_1, “age”) \vee \text{ObjHas}(H, L_2, “age”) \vee \text{ObjHas}(H, L_3, “age”) \vee \text{ObjHas}(H, L_4, “age”)
\]
System $\text{ID} \mathbin{\text{+=}} \text{Prototypes}$

grandpa :: $\{ \nu | \nu :: \text{Ref} \ L_3 \}$
parent :: $\{ \nu | \nu :: \text{Ref} \ L_2 \}$
child :: $\{ \nu | \nu :: \text{Ref} \ L_1 \}$

H =

$\text{L}_3 :: \text{dGrandpa:} \ T_3$
$\text{L}_2 :: \text{dParent:} \ T_2$
$\text{L}_1 :: \text{dChild:} \ T_1$

$\text{var } a = \text{child}[k]$

$\begin{align*}
a &= \text{ObjSel}(H, L_1, k) \\
\text{has(dChild,k)} &\Rightarrow a \in \text{sel(dChild,k)} \quad \land \quad \text{!has(dChild,k)} \Rightarrow a = \text{ObjHas}(H, L_2, k) \\
\text{has(dParent,k)} &\Rightarrow a \in \text{sel(dParent,k)} \quad \land \quad \text{!has(dParent,k)} \Rightarrow a = \text{ObjHas}(H, L_3, k) \\
\text{has(dGrandpa,k)} &\Rightarrow a \in \text{sel(dGrandpa,k)} \quad \land \quad \text{true}
\end{align*}$

unroll the known part of heap
Recap

• System D is functional, but JS is imperative

• Mutation
  – Track heap flow-sensitively to allow strong updates
  – Precise dictionary types live on the heap

• Prototype-based inheritance
  – Model read semantics by traversing proto chain

• System !D = D + Refs + Protos
Dependent JavaScript

• Translation due to Guha et al.
• System !D is type system for LC + Refs + Protos
• Add !D types to source-level JS syntax
• Update translation with !D types
Research Plan

1. System D lambdas, dictionaries, tag-tests

2. + explicit references
   + prototype-based inheritance
   Dependent JavaScript translates to D++

3. Applications in DJS

DJS

\[
\lambda \{ \}
\text{typeof } x := 1
\text{super}
\]
Research Plan

1. System D lambdas, dictionaries, tag-tests
2. + explicit references
   + prototype-based inheritance
Dependent JavaScript translates to D++
3. Applications in DJS

D + references + prototypes
Applications

• “JavaScript: The Good Parts”
  – patterns identified as good practice
• Microbenchmarks
  – ~14 KLOC in SunSpider
• Extensions and widgets
  – secure subsets to allow 3rd party apps
• Frameworks
  – large development cost by few expert programmers
• Type inference in Firefox JIT
  – perhaps add slower, offline, more precise analysis
1. System D
2. DJS = D + refs + protos
3. Applications
Additional Directions

- Better Inference
  - Untyped programmers allergic to annotations
  - Perhaps run-time techniques

- Class-based inheritance

- Contracts
  - Defer obligations not discharged statically

- Study of aliasing in practice
  - How often is mutation “necessary”?  
  - Pipe dream: mainstream language with explicit refs
Thanks!

Collaborators: Pat Rondon, Ranjit Jhala

http://cseweb.ucsd.edu/~rchugh/research/nested/
Extra Slides
tagof :: x:Top → { v | v = tag(x) }

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

• Types

\[ \text{Int} = \{ \nu \mid \text{tag}(\nu) = \text{"Int"} \} \]
\[ x : T_1 \rightarrow T_2 = \{ \nu \mid \nu :: x : T_1 \rightarrow T_2 \} \]

• Formulas

\[ \text{Str}(x) = \text{tag}(x) = \text{"Str"} \]
\[ \text{has}(d,k) = \text{sel}(d,k) \neq \text{bot} \]
\[ \text{EqMod}(d,d',k) = \forall k'. k' \neq k \Rightarrow \text{sel}(d,k) \neq \text{sel}(d',k) \]

• Logical Values

\[ x . k = \text{sel}(\nu, \text{"k"}) \]
\[ x[k] = \text{sel}(\nu, k) \]
Onto

• TODO
Related Work

- TODO
Subtyping with Nesting

Uninterpreted Reasoning + Syntactic Reasoning

\[ \nu :: \text{Top} \to \text{Int} \]
\[ \Rightarrow \nu :: \text{Top} \to \text{Int} \]
\[ \frac{\nu :: \text{Top} \to \text{Int}}{\nu :: \text{Top} \to \text{Int} \Rightarrow \nu :: \text{Int} \to \text{Int}} \]

Normalize formulas to subdivide obligations appropriately
Classic to Nested Refinements in Three Steps
Refinement Types

\[ B ::= \text{Int} \mid \text{Bool} \mid \ldots \]
\[ T ::= \{ \nu : B \mid p \} \]
\[ \mid x : T_1 \rightarrow T_2 \]
\[ p ::= p \land q \mid \ldots \]
\[ \mid x = y \mid x < y \mid \ldots \]

\[ \lambda x . x :: \boxed{x : \{ \nu : \text{Int} \mid \text{true} \} \rightarrow \{ \nu : \text{Int} \mid \text{true} \}} \]
\[ \lambda x . x :: \boxed{x : \{ \nu : \text{Int} \mid \text{true} \} \rightarrow \{ \nu : \text{Int} \mid \nu = x \}} \]

5 :: \boxed{\{ \nu : \text{Int} \mid \text{true} \}}
5 :: \boxed{\{ \nu : \text{Int} \mid 0 < \nu < 10 \}}
5 :: \boxed{\{ \nu : \text{Int} \mid \nu = 10 \}}
Refinement Types

\[ B ::= \text{Int} \mid \text{Bool} \mid \ldots \]
\[ T ::= \{ \nu : B \mid p \} \]
\[ \mid x : T_1 \to T_2 \]
\[ p ::= p \land q \mid \ldots \]
\[ \mid x = y \mid x < y \mid \ldots \]

Type checking is decidable if:

- Underlying refinement logic is decidable
- Only program values appear in formulas
Tag-tests

\[ B ::= \text{Int} | \text{Bool} | \ldots \]
\[ T ::= \{ \nu : B | p \} \]
\[ p ::= p \land q | \ldots \]
\[ x : T_1 \rightarrow T_2 \]
\[ \text{if } \text{tagof } x = \text{"Int"} \text{ then } 0 - x \text{ else } \text{not } x \]

\[ x ::= \{ \nu : ???? \} \quad \text{can’t pick a single base type} \]
\[ x ::= \{ \nu : \text{Any} | \text{tag}(\nu) = \text{"Int"} \lor \text{tag}(\nu) = \text{"Bool"} \} \]
\[ x ::= \{ \nu : \text{Any} | \text{tag}(\nu) = \text{"Int"} \lor \text{tag}(\nu) = \text{"Bool"} \} \]
Tag-tests

Type environment tracks control flow predicates

\[ T ::= \{ \nu \mid p \} \]
\[ \mid x:T_1 \to T_2 \]
\[ p ::= p \land q \mid \ldots \]
\[ \mid x = y \mid x < y \mid \ldots \]
\[ \mid \text{tag}(x) = \text{"Int"} \mid \ldots \]

\[
\begin{align*}
\text{if} \ \text{tagof} \ x &= \text{"Int"} \ \text{then} \ 0 - x \ \text{else} \ \text{not} \ x \\
\end{align*}
\]

\[
\begin{align*}
x &: \{ \nu \mid \text{tag}(\nu) = \text{"Int"} \} \\
x &: \{ \nu \mid \text{tag}(\nu) = \text{"Bool"} \} \\
x &: \{ \nu \mid \text{tag}(\nu) = \text{"Int"} \ \vee \ \text{tag}(\nu) = \text{"Bool"} \}
\end{align*}
\]
Tag-tests

\[ T ::= \{ \nu \mid p \} \]
\[ \mid x : T_1 \rightarrow T_2 \]
\[ p ::= p \land q \mid \ldots \]
\[ \mid x = y \mid x < y \mid \ldots \]
\[ \mid \text{tag}(x) = "Int" \mid \ldots \]
DicEonaries

McCarthy axioms

\[ T ::= \{ \nu \mid p \}
\ |
\ x: T_1 \to T_2
\]

\[ p ::= p \land q \ |
\ x = y \ |
\ x < y \ |
\ ...\]

\[ \forall d, k, k', x. \ k \neq k' \Rightarrow
\]

\[ \text{sel}(\text{upd}(d, k, x), k) = x \]

\[ \text{sel}(\text{upd}(d, k, x), k') = \text{sel}(d, k') \]

\[ \text{sel}(\text{empty}, k) = \text{bot} \]

\[ d.n + d[m] \]

\[ d ::= \{ \nu \mid \text{tag}(\nu) = \text{"Dict"} \]

\[ \land \text{tag}(\text{sel}(\nu, \text{"n"})) = \text{"Int"} \]

\[ \land \text{tag}(\text{sel}(\nu, m)) = \text{"Int"} \} \]
McCarthy axioms

\[
\forall d, k, k', x. \, k \neq k' \implies
\]

\[
\begin{align*}
\text{sel}(\text{upd}(d, k, x), k) &= x \\
\text{sel}(\text{upd}(d, k, x), k') &= \text{sel}(d, k') \\
\text{sel}(\text{empty}, k) &= \text{bot}
\end{align*}
\]
Lambdas

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

\[ d ::= \{ \nu \mid \text{tag}(\nu) = \text{"Dict"} \} \]

**Key Challenge:**
How to describe arrow in a formula?

\[
T ::= \{ \nu \mid p \}
\]

\[
p ::= p \land q \mid \ldots
\]

\[
| \ x : T_1 \rightarrow T_2
\]

\[
| \ x = y \mid x < y \mid \ldots
\]

\[
| \ \text{tag}(x) = \text{"Int"} \mid \ldots
\]

\[
| \ \text{sel}(x,y) = z \mid \ldots
\]
## Refinement Types

**B ::=** `Int` | `Bool` | ...

**T ::=** `{ν:B|p}`

\[ x \colon T_1 \rightarrow T_2 \]

\[ p ::= p \land q \mid \ldots \]

\[ x = y \mid x < y \mid \ldots \]

### if \( x = 1 \) then

\[ \ldots \]

### else

\[ \ldots \]

aka “subset types” and “contract types”

**Suppose**

\[ \Gamma \equiv x : \{ν:\text{Int} | 0 < ν < 3\} \]

\[ \begin{array}{c}
\Gamma_1 \equiv \Gamma, x = 1 \quad \Rightarrow \quad x = 1 \text{ on then-branch} \\
\Gamma_2 \equiv \Gamma, x \neq 1 \quad \Rightarrow \quad x = 2 \text{ on else-branch}
\end{array} \]

- Guard formulas stored in typing environment
- Logic: a natural way to track control flow!
Type for foo, bottom-up
let foo f d = 
  if tagof f = "Str" 
  then d.n + d[f](0) 
  else d.n + f(0)
let foo f d = 
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)

Int = \{ ν | \text{tag}(ν) = "Int" \}

x:T₁ → T₂ = \{ ν | ν :: x:T₁ → T₂ \}

Str(x) = \text{tag}(x) = "Str"

foo ::

f: \{ ν | \text{tag}(ν) = "Str" \} \land \{ ν | ν :: \text{Int} → \text{Int} \}
→ d: \{ ν | \text{tag}(ν) = "Dict" \}
  \land \text{tag}(\text{sel}(ν,"n")) = "Int"
  \land \text{tag}(f) = "Str" ⇒ \text{sel}(ν,f) :: \text{Int} → \text{Int} \}
→ \text{Int}

T ::= \{ ν | p \}
U ::= x:T₁ → T₂
p ::= p \land q | ...
  | x = y | x < y | ...
  | \text{tag}(x) = "Int" | ...
  | \text{sel}(x,y) = z | ...
  | x :: U

79
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)

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

Int = {ν | tag(ν)="Int" } 
x:T₁ → T₂ = {ν | ν ::: x:T₁ → T₂ } 
Str(x) = tag(x)="Str" 
  x.k = sel(ν,"k") 
  x[k] = sel(ν,k)
let foo f d =
  if tagof f = "Str"
  then d.n + d[f](0)
  else d.n + f(0)

f: {ν | Str(ν) ∨ ν :: Int → Int}
→ d: {ν | Dict(ν) ∧ Int(ν.n) ∧ (Str(f) ⇒ ν[f] :: Int → Int)}
→ Int