6. Name Analysis

Eva Rose    Kristoffer Rose

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)
http://cs.nyu.edu/courses/spring15/CSCI-GA.2130-001/lecture-6.pdf

March 9, 2015
Introduction

Programming Language Basics

Symbol Tables = Environments

Symbols in HACS

Synthesis in HACS
Context

source program

Tokens

Lexical Analysis

Syntax Analysis

Tree

Semantic Analysis

Tree

Intermediate Representation Generator

Tree

Symbol Table

Optimizer

IR

Code Generator

IR

Machine-Dependent Code Optimizer

target machine code
Example Code

```c
int initial = 32;
float rate = .8;
float position = initial + rate * 60;
```
Example Abstract Syntax Tree (AST)
Example Abstract Syntax Tree (AST) + “def-use”

```
int= ⟨id, 2⟩ ⟨num, 32⟩ float= ⟨id, 3⟩ ⟨num, .8⟩ float= ⟨id, 1⟩ + ⟨id, 2⟩ *
⟨id, 3⟩ ⟨num, 60⟩
```

<table>
<thead>
<tr>
<th>id</th>
<th>lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>position</td>
</tr>
<tr>
<td>2</td>
<td>initial</td>
</tr>
<tr>
<td>3</td>
<td>rate</td>
</tr>
</tbody>
</table>
1. Introduction

2. Programming Language Basics

3. Symbol Tables = Environments

4. Symbols in HACS

5. Synthesis in HACS
What’s in a Name?

**Identifier** name that identifies an entity

**Variable** abstract notion referring to particular storage location
What’s in a Name?

**Identifier**  name that identifies an entity

**Variable**  abstract notion referring to particular storage location
What Is Known?

Declaration  gives type (etc.) of name.

Definition  gives value (etc.) of name.
What Is Known?

Declaration gives type (etc.) of name.
Definition gives value (etc.) of name.
What to Call?

**Procedure** any callable entity.

**Function** callable entity that “returns” a value.

**Method** callable entity tied to class or object.
What to Call?

**Procedure** any callable entity.

**Function** callable entity that “returns” a value.

**Method** callable entity tied to class or object.
What to Call?

Procedure  any callable entity.
Function   callable entity that “returns” a value.
Method     callable entity tied to class or object.
What is Passed?

Actual parameters the values that occur in a call.

Formal parameters the variables used to refer to the parameters inside procedure.
What is Passed?

Actual parameters the values that occur in a call.

Formal parameters the variables used to refer to the parameters inside procedure.
How is it Passed?

**Call-by-Value**  actual parameter values computed **before** call.

**Call-by-Reference**  actual parameter must be variable which is aliased with formal parameter.

**Call-by-Name**  actual parameter text executed in context of formal parameter.

**Call-by-Need**  like Call-by-Value but evaluation delayed until first use.

**Lazy**  just evaluate sufficiently to create observed parts of data structures.
How is it Passed?

**Call-by-Value** actual parameter values computed before call.

**Call-by-Reference** actual parameter must be variable which is aliased with formal parameter.

**Call-by-Name** actual parameter text executed in context of formal parameter.

**Call-by-Need** like Call-by-Value but evaluation delayed until first use.

**Lazy** just evaluate sufficiently to create observed parts of data structures.
How is it Passed?

Call-by-Value  actual parameter values computed before call.

Call-by-Reference  actual parameter must be variable which is aliased with formal parameter.

Call-by-Name  actual parameter text executed in context of formal parameter.

Call-by-Need  like Call-by-Value but evaluation delayed until first use.

Lazy  just evaluate sufficiently to create observed parts of data structures.
How is it Passed?

Call-by-Value  actual parameter values computed before call.
Call-by-Reference  actual parameter must be variable which is aliased with formal parameter.
Call-by-Name  actual parameter text executed in context of formal parameter.
Call-by-Need  like Call-by-Value but evaluation delayed until first use.

Lazy  just evaluate sufficiently to create observed parts of data structures.
How is it Passed?

**Call-by-Value** actual parameter values computed before call.

**Call-by-Reference** actual parameter must be variable which is aliased with formal parameter.

**Call-by-Name** actual parameter text executed in context of formal parameter.

**Call-by-Need** like Call-by-Value but evaluation delayed until first use.

**Lazy** just evaluate sufficiently to create observed parts of data structures.
### Static vs Dynamic

**Declarations**  
- Address is fixed *vs* runtime allocated.

**Classes**  
- Shared *vs* per instance.

**Scopes**  
- Tied to program “blocks” (lexical) *vs* runtime stack.
Static vs Dynamic

**Declarations**  Address is fixed *vs* runtime allocated.

**Classes**  Shared *vs* per instance.

**Scopes**  Tied to program “blocks” (lexical) *vs* runtime stack.
Static vs Dynamic

**Declarations**  Address is fixed vs runtime allocated.

**Classes**  Shared vs per instance.

**Scopes**  Tied to program “blocks” (lexical) vs runtime stack.
Environment vs State

- **Environment**
  - names
  - locations
  - values

- **State**
Environment vs State

- Environment
  - names
  - locations
  - values

- State
Environment vs State

- **Environment**
  - names
  - locations
  - **static** (compile-time)

- **State**
  - values
**Environment vs State**

- **Environment** vs **State**
- **names** (environment)
- **locations**
- **values** (run-time)
- **dynamic**

Eva Rose, Kristoffer Rose
Compiler Construction (CSCI-GA.2130-001) 6. Name Analysis
March 9, 2015
Environment vs State

- **Environment**
  - names
  - locations
  - values
  - compiler

- **State**
  - environment
  - state

---

Eva Rose, Kristoffer Rose

Compiler Construction (CSCI-GA.2130-001) 6. Name Analysis

March 9, 2015
Environment vs State

- **Environment**
  - names
  - locations
  - values (run-time)

- **State**
  - environment
  - state
  - locations
  - values (run-time)
Focus of Class... 

We are concerned with... 

- Static scoping. 
- Environments. 
- Planning runtime state.
Focus of Class... 

We are concerned with... 

- Static scoping.
- Environments.
- Planning runtime state.
Focus of Class... 

We are concerned with... 

- Static scoping. 
- Environments. 
- Planning runtime state.
Static (Lexical) Scoping

int = Local Value Scope
Formal Parameters

function

Param

Scope
Exercise 1.6.1

```c
int w, x, y, z;
int i = 4; int j = 5;
{
    int j = 7; i = 6; w = i + j;
}
x = i + j;
{
    int i = 8; y = i + j;
}
z = i + j;
```
Exercise 1.6.4

```c
#define a (x+1)
int x = 2;

void b() {
    x = a; printf("%d\n", x);
}
void c() {
    int x = 1; printf("%d\n", a);
}
void main() { b(); c(); }
```
Introduction

Programming Language Basics

Symbol Tables = Environments

Symbols in HACS

Synthesis in HACS
Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
- Really a tree: common ancestor blocks can be shared.
- Can get messy if hash table is used.
- Interferes with semantic rules/actions.
- *We shall fix this! It is a little different from the book.*
Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.
Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.
Introduction Programming Language Basics Symbol Tables = Environments Symbols in HACS Synthesis in HACS

Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- We shall fix this! It is a little different from the book.
Symbol Tables

- Traditional method for managing binders in system.
- Logically **one symbol table per scope**.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.

*We shall fix this! It is a little different from the book.*
Symbol Tables

- Traditional method for managing binders in system.
- Logically one symbol table per scope.
  - Really a tree: common ancestor blocks can be shared.
  - Can get messy if hash-table is used.
- Interferes with semantic rules/actions.
- *We shall fix this!* It is a little different from the book.
HACS is *Higher-order* Attribute Contraction Schemes

- Traditional:

  \[
  P \to S^* \\
  S \to \text{int } V = E; \mid \text{print } V;
  \]

- Combine Scoping and Grammar:

  \[
  P \to S \\
  S \to \text{int } V = E; S \mid \text{print } V; S \mid \epsilon
  \]
HACS is *Higher-order* Attribute Contraction Schemes

- **Traditional:**

  \[ P \rightarrow S^* \]
  \[ S \rightarrow \text{int } V = E; \mid \text{print } V; \]

- **Combine Scoping and Grammar:**

  \[ P \rightarrow S \]
  \[ S \rightarrow \text{int } V = E; S \mid \text{print } V; S \mid \epsilon \]
Environment Example

```
int =  
⟨id, 2⟩  ⟨num, 32⟩

float =  
⟨id, 3⟩  ⟨num, .8⟩

float =  
+  
⟨id, 1⟩

+  
⟨id, 2⟩  *  
⟨id, 3⟩  ⟨num, 60⟩
```
Environment Example

```
int =
⟨id, 2⟩
  ⟨num, 32⟩

float =
⟨id, 3⟩
  ⟨num, .8⟩

float =
⟨id, 1⟩
  +
  ⟨id, 2⟩
    *
    ⟨id, 3⟩
      ⟨num, 60⟩
```

<table>
<thead>
<tr>
<th>id</th>
<th>lexeme</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>initial</td>
<td>int</td>
</tr>
</tbody>
</table>
Environment Example

int = ⟨id, 2⟩
  |  ⟨num, 32⟩
  ↓
  ⟨id, 3⟩
  |  ⟨num, .8⟩
  ↓
  ⟨id, 1⟩

float = ⟨id, 3⟩
  |  ⟨num, 60⟩
  ↓
  ⟨id, 2⟩

float =
  |  +
  ↓
  ⟨id, 3⟩

float =
  |  *
  ↓
  ⟨id, 3⟩

id | lexeme | type
----|--------|-----
2   | initial | int

id | lexeme | type
----|--------|-----
2   | initial | int
3   | rate    | float

id | lexeme | type
----|--------|-----
2   | initial | int

Environment Example

\[
\begin{align*}
\text{int} &= \langle \text{id}, 2 \rangle \\
\text{float} &= \langle \text{id}, 3 \rangle \\
\text{float} &= \langle \text{id}, 1 \rangle \\
\text{float} &= \langle \text{id}, 2 \rangle + \langle \text{id}, 3 \rangle \times \langle \text{num}, 60 \rangle
\end{align*}
\]
Environment Example with *Stack*

```
int = ⟨id, 2⟩  float = ⟨id, 3⟩  float = ⟨id, 1⟩
   ⟨num, 32⟩   ⟨num, .8⟩    +
```
Environment Example with Maps
Environment Example with *Stack of Maps*

- **int**:
  - `<id, 2>`
  - `<num, 32>`

- **float**:
  - `<id, 3>`
  - `<num, .8>`
  - `<id, 1>`
  - `<id, 2>`
  - `+`
  - `*`
    - `<id, 3>`
    - `<num, 60>`

- **Symbols in HACS**

**Symbols in Environments**

- `int = {initial \mapsto \text{int}}`
- `e = \{\text{rate} \mapsto \text{float}\}`
- `e = \{\text{position} \mapsto \text{float}\}`
Binding Construct with Local Symbol Table = Environment

\[ e = \{ x \mapsto \text{int}, \ldots \} \]
Binding Construct \textit{with Local Symbol Table = Environment II}

\[
\begin{align*}
\text{int}= & \{x \mapsto \text{int,} \ldots\} \\
\text{float}= & \{y \mapsto \text{float,} x \mapsto \text{int,} \ldots\}
\end{align*}
\]
Shadowing

```c
int x = 32;
int y;
{
    float x = .8;
    float y = x + x * 8;
}

y = y + x;
```
Binding Construct with Local Symbol Table = Environment III

\[ \text{int} = x \]

\[ \text{float} = \{ x \mapsto \text{int}, \ldots \} \]

\[ \downarrow \downarrow \]

\[ S \]

\[ e = \{ x \mapsto \text{int}, \ldots \} \]
## Example

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} := E_1; S_2$</td>
<td>$E_1.e = S.e; \ S_2.e = \text{Extend}(S.e, \ \text{id}.sym, E_1.t)$</td>
</tr>
<tr>
<td>$</td>
<td>{ S_1 } S_2$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td></td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T_2$</td>
<td>$E_1.e = E.e; \ T_2.e = E.e; \ E.t = \text{Unif}(E_1.t, T_2.t)$</td>
</tr>
<tr>
<td>$</td>
<td>T_1$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 \ast F_2$</td>
<td>$T_1.e = T.e; \ F_2.e = T.e; \ T.t = \text{Unif}(T_1.t, F_2.t)$</td>
</tr>
<tr>
<td>$</td>
<td>F_1$</td>
</tr>
<tr>
<td>$F \rightarrow \text{id}$</td>
<td>$F.t = \text{Lookup}(F.e, \ \text{id}.sym)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{int}$</td>
</tr>
<tr>
<td>$</td>
<td>\text{float}$</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th><strong>Production</strong></th>
<th><strong>Semantic Rules</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow \text{id} := E_1 ; S_2 )</td>
<td>( E_1.e = S.e ; S_2.e = \text{Extend}(S.e, \text{id.sym}, E_1.t) )</td>
</tr>
<tr>
<td>( \mid { S_1 } \ S_2 )</td>
<td>( S_1.e = S.e ; S_2.e = S.e )</td>
</tr>
<tr>
<td>( \epsilon )</td>
<td></td>
</tr>
<tr>
<td>( E \rightarrow E_1 + T_2 )</td>
<td>( E_1.e = E.e ; T_2.e = E.e ; E.t = \text{Unif}(E_1.t, T_2.t) )</td>
</tr>
<tr>
<td>( \mid T_1 )</td>
<td>( T_1.e = E.e ; E.t = T_1.t )</td>
</tr>
<tr>
<td>( T \rightarrow T_1 \ast F_2 )</td>
<td>( T_1.e = T.e ; F_2.e = T.e ; T.t = \text{Unif}(T_1.t, F_2.t) )</td>
</tr>
<tr>
<td>( \mid F_1 )</td>
<td>( F_1.e = T.e ; T.t = F_1.t )</td>
</tr>
<tr>
<td>( F \rightarrow \text{id} )</td>
<td>( F.t = \text{Lookup}(F.e, \text{id.sym}) )</td>
</tr>
<tr>
<td>( \mid \text{int} )</td>
<td>( E.t = \text{Int} )</td>
</tr>
<tr>
<td>( \mid \text{float} )</td>
<td>( E.t = \text{Float} )</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th><strong>Production</strong></th>
<th><strong>Semantic Rules</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} ::= E_1; S_2$</td>
<td>$E_1.e = S.e$; $S_2.e = \text{Extend}(S.e, \text{id}.sym, E_1.t)$</td>
</tr>
<tr>
<td>$\mid { S_1 } S_2$</td>
<td>$S_1.e = S.e$; $S_2.e = S.e$</td>
</tr>
<tr>
<td>$\mid \epsilon$</td>
<td></td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T_2$</td>
<td>$E_1.e = E.e$; $T_2.e = E.e$; $E.t = \text{Unif}(E_1.t, T_2.t)$</td>
</tr>
<tr>
<td>$\mid T_1$</td>
<td>$T_1.e = E.e$; $E.t = T_1.t$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F_2$</td>
<td>$T_1.e = T.e$; $F_2.e = T.e$; $T.t = \text{Unif}(T_1.t, F_2.t)$</td>
</tr>
<tr>
<td>$\mid F_1$</td>
<td>$F_1.e = T.e$; $T.t = F_1.t$</td>
</tr>
<tr>
<td>$F \rightarrow \text{id}$</td>
<td>$F.t = \text{Lookup}(F.e, \text{id}.sym)$</td>
</tr>
<tr>
<td>$\mid \text{int}$</td>
<td>$E.t = \text{Int}$</td>
</tr>
<tr>
<td>$\mid \text{float}$</td>
<td>$E.t = \text{Float}$</td>
</tr>
</tbody>
</table>
Introduction

Programming Language Basics

Symbol Tables = Environments

Symbols in HACS

Synthesis in HACS
Binding Construct

```
int= x

V_1

float= e = \{x \mapsto \text{int}, \ldots\}

\downarrow \downarrow

y

V_2

S e = \{y \mapsto \text{float}, x \mapsto \text{int}, \ldots\}
```
Binding Construct à la HACS

\[ \begin{align*}
V_1 & \quad \text{int} = [x] \\
V_2 & \quad \text{float} = [y] \\
S & \quad e = \{y \mapsto \text{float}, x \mapsto \text{int}, \ldots\} \\
& \quad e = \{x \mapsto \text{int}, \ldots\}
\end{align*} \]
Binding Construct à la HACS

\[
\begin{align*}
\text{int} & = V_1 \\
\text{float} & = [x] V_2 \\
S & = e = \{x \mapsto \text{int}, \ldots\} \\
S & = e = \{x \mapsto \text{float}, \ldots\}
\end{align*}
\]
HACS is Higher-order Attribute Contraction Schemes II

\[
P \rightarrow S \\
S \rightarrow \text{int } V^x = E; S^x | \text{print } V; S | \epsilon
\]

\[
\text{sort } V | \text{symbol } [\langle \text{ID} \rangle] ; \\
\text{sort } P | [\langle S \rangle] ; \\
\text{sort } S | [\text{int } \langle V \text{ binds } x \rangle = \langle E \rangle; \langle S[x \text{ as } V] \rangle ] \\
| [\text{print } \langle V \rangle; \langle S \rangle ] \\
| [ ] ;
\]
HACS is *Higher-order* Attribute Contraction Schemes II

\[
P \rightarrow S
\]

\[
S \rightarrow \text{int } V_x = E; S^x \mid \text{print } V; S \mid \epsilon
\]

sort \( V \) \mid symbol \( [[\langle \text{ID} \rangle]] \) ;

sort \( P \) \mid \( [[\langle S \rangle]] \) ;

sort \( S \) \mid \( [[\text{int } \langle V \text{ binds } x \rangle = \langle E \rangle; \langle S[x \text{ as } V] \rangle]] \)

\mid \( [[\text{print } \langle V \rangle; \langle S \rangle]] \)

\mid \( [] \) ;
1. Introduction

2. Programming Language Basics

3. Symbol Tables = Environments

4. Symbols in HACS

5. Synthesis in HACS
Example: Type Synthesis SDD

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E.t = \text{Unif}(E_1.t, E_2.t)$ (1)</td>
</tr>
<tr>
<td>$E_1 \times E_2$</td>
<td>$E.t = \text{Unif}(E_1.t, E_2.t)$ (2)</td>
</tr>
<tr>
<td>int</td>
<td>$E.t = \text{Int}$ (3)</td>
</tr>
<tr>
<td>float</td>
<td>$E.t = \text{Float}$ (4)</td>
</tr>
</tbody>
</table>
Example: Type Synthesis HACS Sorts

\texttt{sort\ Type\ |\ Int\ |\ Float;}
Example: Type Synthesis HACS Unification Rules

\[
\text{sort Type} \mid \text{scheme Unif}(	ext{Type, Type}); \\
\text{Unif}(\text{Int, Int}) \rightarrow \text{Int}; \\
\text{Unif}(\text{Float, #}) \rightarrow \text{Float}; \\
\text{Unif}(\#, \text{Float}) \rightarrow \text{Float};
\]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad | \quad E.t = \text{Unif}(E_1.t, E_2.t) \quad (1) \]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad | \quad E.t = \text{Unif}(E_1.t, E_2.t) \quad (1) \]

\[ \langle \text{Exp} \uparrow_{\text{type}(#1)} \rangle + \langle \text{Exp} \uparrow_{\text{type}(#2)} \rangle \rangle \uparrow_{\text{type}(\text{Unif}(#1, #2))} \]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad | \quad E.t = \text{Unif}(E_1.t, E_2.t) \]  

\[ \llbracket \langle Exp \#1 \uparrow \text{type}(\#t1) \rangle + \langle Exp \#2 \uparrow \text{type}(\#t2) \rangle \rrbracket \uparrow \text{type}(\text{Unif}(\#t1, \#t2)) \]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad | \quad E.t = \text{Unif}(E_1.t, E_2.t) \quad (1) \]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad \mid \quad E.t = \text{Unif}(E_1.t, E_2.t) \quad (1) \]

\[ \llbracket \langle \text{Exp} \#1 \uparrow \text{type}(\#t_1) \rangle + \langle \text{Exp} \#2 \uparrow \text{type}(\#t_2) \rangle \rrbracket \uparrow \text{type}(\text{Unif}(\#t_1, \#t_2)) \]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad | \quad E.t = \text{Unif}(E_1.t, E_2.t) \quad (1) \]

\[ \ll (\text{Exp}\#1 \uparrow \text{type} (#t1)) + (\text{Exp}\#2 \uparrow \text{type} (#t2)) \rr \uparrow \text{type} (\text{Unif} (#t1, #t2)) \]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad | \quad E.t = \text{Unif}(E_1.t, E_2.t) \]  

\[ \langle \langle \text{Exp} \#1 \uparrow \text{type}(\#t1) \rangle \rangle + \langle \langle \text{Exp} \#2 \uparrow \text{type}(\#t2) \rangle \rangle \uparrow \text{type}(\text{Unif}(\#t1, \#t2)) \]
Example: Type Synthesis SDD Rule (1)

\[ E \rightarrow E_1 + E_2 \quad | \quad E.t = \text{Unif}(E_1.t, E_2.t) \]  

\[
\left\langle \langle \text{Exp} \#1 \uparrow \text{type}(\#t1) \rangle + \langle \text{Exp} \#2 \uparrow \text{type}(\#t2) \rangle \right\rangle \uparrow \text{type}(\text{Unif}(\#t1, \#t2))
\]
Example: Type Synthesis HACS

attribute \( \uparrow \text{type}(\text{Type}) \);

sort Exp \( \uparrow \text{type} \);

\[
\begin{align*}
\left[ \langle \text{Exp}\#1 \uparrow \text{type}(\#t1) \rangle + \langle \text{Exp}\#2 \uparrow \text{type}(\#t2) \rangle \right] & \uparrow \text{type}(\text{Unif}(\#t1, \#t2)); \\
\left[ \langle \text{Exp}\#1 \uparrow \text{type}(\#t1) \rangle \ast \langle \text{Exp}\#2 \uparrow \text{type}(\#t2) \rangle \right] & \uparrow \text{type}(\text{Unif}(\#t1, \#t2)); \\
\left[ \langle \text{Int}\# \rangle \right] & \uparrow \text{type}(\text{Int}); \\
\left[ \langle \text{Float}\# \rangle \right] & \uparrow \text{type}(\text{Float});
\end{align*}
\]
More HACS . . .

HACS manual section 7.

http://crsx.org/hacs.pdf
Questions?

evarose@cs.nyu.edu    krisrose@cs.nyu.edu