Typeful Code Representation at Low-level

Presented by Rui Shi

PL Reading Group

Outline

- An overview of *Typeful Code Representation.*
- Implementing an evaluator for STLC in *ATS/SV*.
- Introduction to Ocaml byte-code.

Object Language vs. Meta Language

- *Object language* is the one to be studied and manipulated.

- *Meta language* is the one in which we construct programs.

Applications which need Code Representation

- Almost all the language related applications need to deal with code representation:
  - Parser
  - Type-checker
  - Program transformer
  - Evaluator
  - …
- In different phases of a compilation, various internal representations are needed.

A Representation of Simply-Typed Lambda Calculus

- We can declare the following datatype in ML to represent *STLC.*

```
datatype EXP =
  EXPvar of string
| EXPlam of string * EXP
| EXPapp of EXP * EXP
```

- For instance, the λ-expression λx:int. λy:int->int.y(x) is now represented as follows:

$$EXPlam("x", EXPlam("y", EXPapp(EXPvar "y", EXPvar "x")))$$

Some Problems

- The type of an object program can not be reflected in the type of its representation.

- ill-typed STLC expressions can also be constructed, but can not be captured by the ML type system, for instance

$$EXPlam("x", EXPapp(EXPvar "x", EXPvar "x"))$$

DeBruijn Indexes

- Variables N = 1 | N^
- Expressions E = N | λ.E | (E1 E2)
- Example: λx.λy.y(x) can be represented in DeBruijn notation as

λ.λ.(1 1^)

Types and Contexts

- Infix -> ::
- Sort ty = int | ty -> ty
- Sort env = nil | ty :: env

A F.O. Typeful Representation using Dependent Types

```
datatype VAR (env, ty) =
  | {G:env, t:ty}. VARone (t :: G, t)
  | {G:env, t1:ty, t2:ty}.
    VARshi (t2 :: G, t1) of VAR (G, t1)

datatype EXP (env, ty) =
  | {G:env, t:ty}. EXPvar (G, t) of VAR (G, t)
  | {G:env, t1:ty, t2:ty}.
      EXPlam (G, t1 -> t2) of EXP (t1 :: G, t2)
  | {G:env, t1:ty, t2:ty}.
      EXPapp (G, t2) of
        EXP (G, t1 -> t2) * EXP (G, t1)
```

Example

$$EXPlam(EXPlam(\\ \quad EXPapp(EXPvar(VARone),\\ \qquad EXPvar(VARshi(VARone)))))$$

Where the above value is assigned the following type:

$$EXP(nil, int \rightarrow (int \rightarrow int) \rightarrow int)$$

Capturing invariants of object programs

- Only well-typed object programs can be constructed and reasoned about.

- For instance, a normalizing function on lambda-expressions can be given the following type:

  {G:env, t: type}. EXP(G, t) -> EXP(G, t)

- …

Other Applications of Typeful Code Representation

- Program Transformation
- Meta Programming
- Distributed Meta-Programming
- XML
- …

Outline

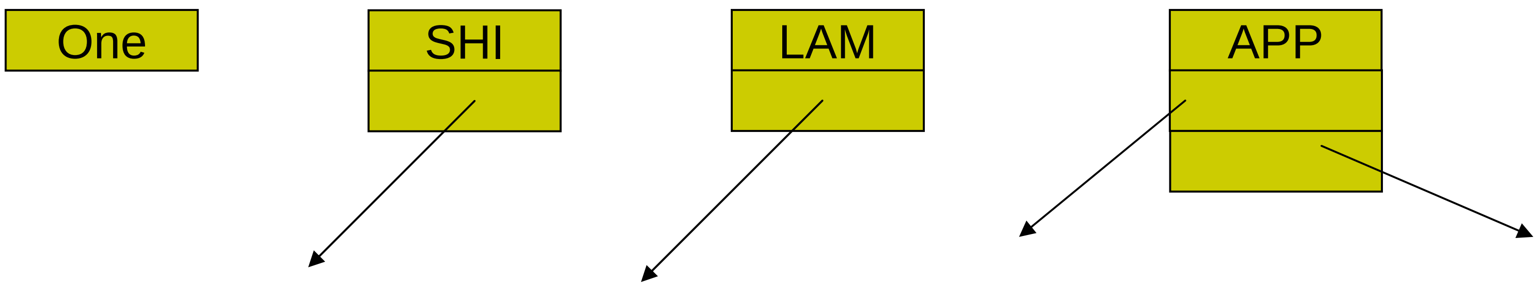
- An overview of *Typeful Code Representation.*

- Implementing an evaluator for STLC in *ATS/SV*.

- Introduction to Ocaml byte-code.

An Representation of Lambda Expressions in C

- We have 4 kinds of nodes for representing languages constructs, respectively.

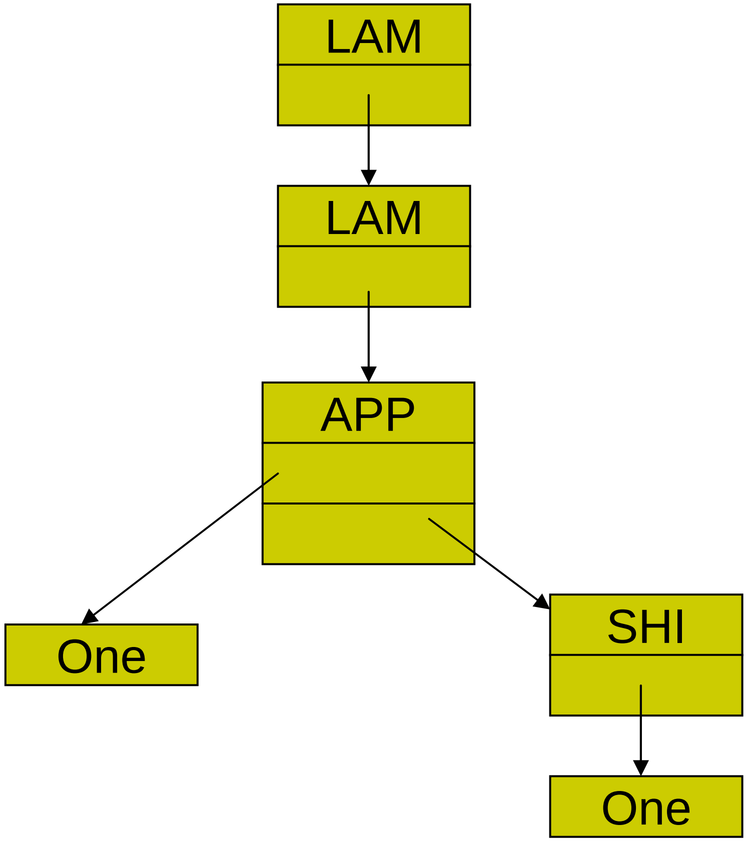


- The above nodes corresponds to value constructors in ML.

datatype exp = One | Shi of exp | Lam of exp
| App of exp * exp

Example

- λx:int. λy:int->int.y(x)



Concerns about Implementing an Evaluator in C

- The type of an object program is completely lost.
- A lot of memory operations are involved, and thus potential illegal memory access is a serious problem.

Encoding datatypes in ATS/SV

- Pair:

```
typedef pair (a1: type, a2: type) =
    [l: addr] (!(a1 @ l), !(a2 @ l+1) | ptr(l))
```

- Sum:

```
typedef sum (a1: type, a2: type) =
    [l: addr, i: two]
      (!(int (i) @ l),
        {i == 0} !(a1 @ l+1),
        {i == 1} !(a2 @ l+1) | ptr(l))
```

A Typeful Representation of STLC in AST/SV

- typedef ast (g: env, t: type) =

```
[l: addr, i: four]
  (!(int(i) @ l),
  {i == 0} [g':env | g = t :: g'] !(),
  {i == 1} [g': env, t1: type | g = t1 :: g']
             !(ast(g', t) @ l+1),
  {i == 2} [t1: type, t2: type | t = t1 -> t2]
             !(ast(t1 :: g, t2) @ l+1),
  {i == 3} [t1: type]
             !(ast(g, t1 -> t) @ l+1, ast(g, t1) @ l+2) | ptr(l))
```

Comparing programs in C and ATS/SV

```
value makeApp(value e1, value e2)
{
    char * _res_ = (char *)
      malloc(sizeof(header_t) + APPSIZE
      * sizeof(value));

    *(header_t *) _res_ =
        Make_header(APPSIZE, Black,
      APP);

    *(value *) ((header_t *)_res_ + 1)
        = e1;
    *((value *) ((header_t *)_res_ + 1) + 1)
        = e2;

    return ((value) (_res_)  << 1);
}
```

```
fun makeApp {g: env, t1: type, t: type}
        (e1: ast(g,  t1 -> t), e2: ast(g, t1)):
          ast(g, t) =
  let
    val '(pf1, pf2, pf3 | p) = alloc3()

    val '(pf1 | _) = setVar(pf1 | p, 3)

    val '(pf2 | _) = setVar(pf2 | p+1, e1)

    val '(pf3 | _) = setVar(pf3 | p+2, e2)
  in
    '(invar pf1, prunit, prunit, prunit,
     '(invar pf2, invar pf3) | p)
  end
```

Implementing an Evaluator in ATS/SV

- Only well-typed object programs can be constructed and stored in memory.
- Various invariants related to types of object programs can be captured in ATS/SV.
- Illegal memory operations can also be prevented.

Ongoing Work

- Provide a typeful representation for low-level object languages
  - Typed byte-code language
  - Typed assembly language
- Implementing runtime systems for low-level languages.

Outline

- An overview of *Typeful Code Representation.*
- Implementing a runtime system in *ATS/SV*.
- Introduction to Ocaml byte-code language.

Syntax

(Program)	$\mathbb{P}$	$::=$	$(\mathbb{I}, \mathbb{A}, \mathbb{G}, \mathbb{S}_a, \mathbb{S}_r)$
(Accumulator)	$\mathbb{A}$	$::=$	v (constant or closure)
(Environment)	$\mathbb{G}$	$::=$	$\epsilon \mid v :: \mathbb{G}$
(Argument Stack)	$\mathbb{S}_a$	$::=$	$\epsilon \mid v :: \mathbb{S}_a$
(Return Stack)	$\mathbb{S}_r$	$::=$	$\epsilon \mid v :: \mathbb{S}_r$
(Closure)	a	$::=$	$(\mathbb{I}, \mathbb{G})$
(Constant)	c	$::=$	integers | strings | ...
(Value)	v	$::=$	$a \mid c$
(Instruction)	i	$::=$	**ACCESS**(n) | **PUSHMARK** | **PUSH** | **APPLY** **GRAB** | **CUR**$(\mathbb{I})$ | **LET** | **ENDLET** | **DUMMY** **UPDATE** | ...
(Instruction Sequence)	$\mathbb{I}$	$::=$	$\epsilon \mid i; \mathbb{I}$

Accessing Variables

$$(\mathbf{ACCESS}(n); \mathbb{I}, v, v_0 :: \ldots :: v_n :: \ldots :: \epsilon, s, r)$$

$$\longmapsto$$

$$(\mathbb{I}, v_n, v_0 :: \ldots :: v_n :: \ldots :: \epsilon, s, r)$$

Abstraction

$$
\begin{aligned}
(\mathbf{CUR}(\mathbb{I}_1); \mathbb{I}_0, v, g, s, r) &\longmapsto (\mathbb{I}_0, (\mathbb{I}_1, g), g, s, r) \\
(\mathbf{GRAB}; \mathbb{I}_0, v, g_0, \star :: s, (\mathbb{I}_1, g_1) :: r) &\longmapsto (\mathbb{I}_1, (\mathbb{I}_0, g_0), g_1, s, r) \\
(\mathbf{GRAB}; \mathbb{I}, v, g, v_0 :: s, r) &\longmapsto (\mathbb{I}, v, v_0 :: g, s, r) \\
(\mathbf{RETURN}; \mathbb{I}_0, v, g_0, \star :: s, (\mathbb{I}_1, g_1) :: r) &\longmapsto (\mathbb{I}_1, v, g_1, s, r) \\
(\mathbf{RETURN}; \mathbb{I}_0, (\mathbb{I}_1, g_1), g_0, v_0 :: s, r) &\longmapsto (\mathbb{I}_1, (\mathbb{I}_1, g_1), v_0 :: g_1, s, r)
\end{aligned}
$$

Application

$$
\begin{aligned}
(\mathbf{APPTERM}; \mathbb{I}_0, (\mathbb{I}_1, g_1), g_0, v_0 :: s, r) &\longmapsto (\mathbb{I}_1, (\mathbb{I}_1, g_1), v_0 :: g_1, s, r) \\
(\mathbf{APPLY}; \mathbb{I}_0, (\mathbb{I}_1, g_1), g_0, v_0 :: s, r) &\longmapsto (\mathbb{I}_1, (\mathbb{I}_1, g_1), v_0 :: g_1, s, (\mathbb{I}_0, g_0) :: r) \\
(\mathbf{PUSH}; \mathbb{I}_0, v, g, s, r) &\longmapsto (\mathbb{I}_0, v, g, v :: s, r) \\
(\mathbf{PUSHMARK}; \mathbb{I}_0, v, g, s, r) &\longmapsto (\mathbb{I}_0, v, g, \star :: s, r)
\end{aligned}
$$