COS 320 Types III

- hw3 due today
- hw4 released today, due April 1st
  (typechecker and translator for Oat)

Oat v2:
- specified by fairly large type system
- Several features:
  - ★ Memory safety
  - ★ Mutable record types
  - ★ Subtyping
    - non-null ref's are subtype of nullable refs (ref <: ref?)
    - width subtyping for record

Compiling w/ a type system

- take intrinsic view: types are syntactic part of program
  → program w/o types is not program.
- so only look at well-typed programs.

- Compiler translates programs in source language to target language.
  - ★ Explicit typechecking pass to throw out ill-typed programs, or
  - ★ translation pass to throw exceptions.
  - ★ Compiler assumes (lang checks if front-end produces target program is well-typed. type-correct code.

Compilation - translation of derivation of judgements from src lang to proofs in target lang

Each kind of judgement has a different translation category

- Well-formed types in src become well-formed types in target
- Exprs in src become (operand, instr list) in target.

Each inference rule corresponds to a case within that category

example: dat v1

- Implicit type system

$\vdash t$ "t is well formed type"

$\vdash_r ref$ "ref is well-formed ref-type"

$\vdash_{rt} rt$ "rt is well-formed return type"

$$\text{TInt}\ \frac{}{\vdash \text{int}} \qquad \text{TBool}\ \frac{}{\vdash \text{bool}} \qquad \text{TRef}\ \frac{\vdash_r ref}{\vdash ref}$$

$$\text{RString}\ \frac{}{\vdash_r \text{string}} \qquad \text{RArray}\ \frac{\vdash t}{\vdash_r t[]} \qquad \text{RFun}\ \frac{\vdash t_1 \quad \ldots \vdash t_n \quad \vdash_{rt} rt}{\vdash (t_1 \ldots t_n) \to rt}$$

$$\text{RTVoid}\ \frac{}{\vdash_{rt} \text{void}} \qquad \text{RTTyp}\ \frac{\vdash t}{\vdash_{rt} t}$$

LLVMdite type system

Judgements:

- $T \vdash t$ within named types T, t is well-formed type.
- $T \vdash_s t$ within named types T, t is well-formed simple type
- $T \vdash_{rs} t$ within named types T, t is well-formed ref type
- $T \vdash_{rt} t$ within ..... t is w.f. return type

$$\frac{}{T \vdash_s i1}\ \text{LLBool} \qquad \frac{}{T \vdash_s i64}\ \text{LLInt} \qquad \frac{T \vdash_r ref}{T \vdash_s ref*}\ \text{LLPtr}$$

$$\frac{T \vdash t_1 \ldots T \vdash t_n}{T \vdash \{t_1 \ldots t_n\}}\ \text{LLTuple} \qquad \frac{T \vdash t}{T \vdash [n \times t]}\ \text{LLArray} \quad n \neq 0 \qquad \frac{\vdash_s t}{\vdash t}\ \text{notLLSimple}$$

$$\frac{}{T \vdash_r i8}\ \text{LLChar} \qquad \frac{}{}\ \text{LLRtSimple} \qquad \frac{}{}\ \text{LLRtVoid}$$

$$\frac{T \vdash t}{T \vdash_r t}\ \text{LLRtype} \qquad \frac{}{}\ \text{LLRfun}$$

$$\frac{}{T \vdash \%vid}\ \text{LLNamed} \quad \%vid \in T$$

↑ ② ↑ ①

Just using %vid as type is well formed if ①: vid is contained in T.

Translating Well formed Types

Each well-formed oat type ⟶ well-formed LLVM type

- types ⟶ simple types
- ref types ⟶ ref types
- ret types ⟶ ret types

Use $\rightsquigarrow$ to denote translation of derivations.

Axioms: $\frac{}{\vdash \text{int}}\ \text{TInt}$ $\frac{}{\vdash \text{bool}}\ \text{TBool}$ $\frac{\vdash_r \text{ref}}{\vdash \text{ref}}\ \text{TRef}$.

$$\left(\frac{}{\vdash \text{int}}\ \text{TInt}\right) \rightsquigarrow \left(\frac{}{\vdash_s i64}\ \text{LLInt}\right)$$

$$\left(\frac{}{\vdash \text{bool}}\ \text{TBool}\right) \rightsquigarrow \left(\frac{}{\vdash_s i1}\ \text{LLBool}\right)$$

$$\star\left(\frac{\vdash_r \text{ref}}{\vdash \text{ref}}\ \text{TRef}\right) \rightsquigarrow \left(\frac{\vdash_r t}{\vdash t*}\ \text{LLPtr}\right)$$

where $(\vdash_r \text{ref}) \rightsquigarrow (\vdash_r t)$

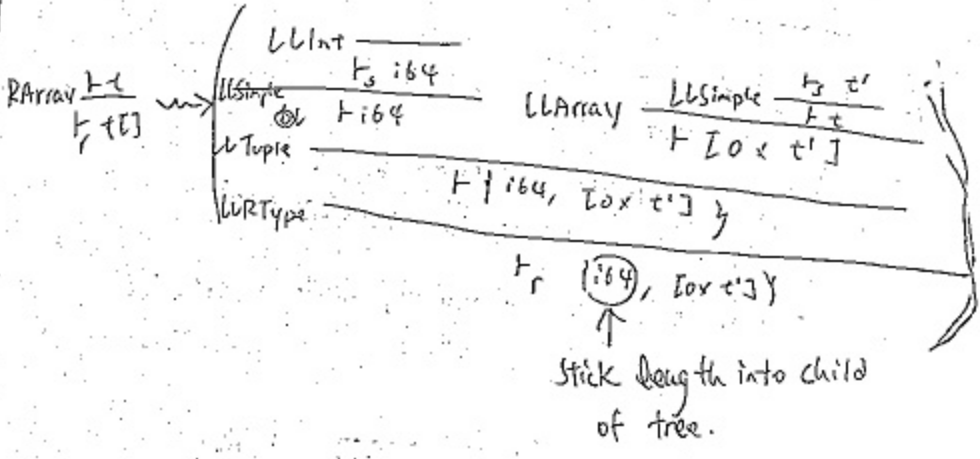
extension of type judgment tree.

In Oat v2, array accesses are checked at runtime.

Recall: Can implement runtime array access ✓ checking by allocating additional field storing the length.

⟶ We do exactly that:

Array Access:



Stick length into child of tree.

Tedious to write out trees → leave trees implicit.

Succint notation: $[\![\vdash J]\!] = J'$

derivation w/ root J translates to derivation w/ root J'

- $[\![\vdash \text{int}]\!] = \vdash_S i64$
- $[\![\vdash \text{bool}]\!] = \vdash_S i1$
- $[\![\vdash \text{ref}]\!] = \vdash_S t*$, where $\vdash_r t = [\![\text{ref}]\!]$

... ...

Well-formed Codestreams

Judgements take the form

① $\Gamma \vdash s \Longrightarrow \Gamma'$:

under typeenv Γ, codestream s is well formed ... results in typeenv Γ'

② $\Gamma \vdash opn : t$ "under typeenv Γ, operand opn has type t"

$$\text{Id}\ \frac{}{\vdash : id : \Gamma(:id)} \qquad \text{Num}\ \frac{}{\Gamma \vdash n : i64}\ n \in \mathbb{Z}$$

$$\text{Add}\ \frac{\Gamma \vdash opn_1 : i64 \qquad \Gamma \vdash opn_2 : i64}{\Gamma \vdash \%uid = add\ i64\ opn_1, opn_2 \Rightarrow \Gamma\{\%uid \mapsto i64\}}\ \%uid \notin dom(\Gamma)$$

$\{\%uid \mapsto i64\}$: Modify the type environment. "Intuitive easy" parsing

side condition: uid is not in domain of Γ; enforce SSA property

$$\text{Seq}\ \frac{\Gamma \vdash s_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash s_2 \Rightarrow \Gamma''}{\Gamma \vdash s_1, s_2 \Rightarrow \Gamma''}$$

order is important for SSA property.

Well-typed expressions

- translates a type judgement $\Gamma \vdash e : \tau$ (in oat) to
  codestream judgement $\Gamma_\mu \vdash s \Rightarrow \Gamma'_{\mu'}$
  operand judgement $\Gamma'_{\mu'} \vdash \text{opn} : t_\mu$

  if, execute the code stream, then operand has value that the expression would take.

Goal: well typed exprs in oat $\longrightarrow$ derivation of codestreams, operand judgements.

D: $\dfrac{}{\Gamma \vdash x : t}$ Var

How to translate $\Gamma \vdash x : t$ (Var):

* Need symbol table ctxt : Oat identifiers $\longrightarrow$ (LLVMlite operand judgements).
  (Operand associated w/ variable x is pointer to memory location associated w/ x).

* to compute $[\![\Gamma \vdash x : t]\!]$ (ctxt), first

first let (id, t') = ctxt(x). then

- define $[\![\text{ctxt}]\!]$ to be the LLVM typeond associated w/ ctxt.
  - $[\![\epsilon]\!] = \epsilon$ (empty context → empty assoc)
  - $[\![\text{ctxt}, x \mapsto (id, t')]\!] = \Gamma_\mu, id \mapsto t$
    where $[\![\text{ctxt}]\!] = \Gamma_\mu$.
- Codestream: $[\![\text{ctxt}]\!] \vdash \%id = \text{load } \tau, \tau* id \Rightarrow [\![\text{ctxt}]\!]\{\%id \to t\}$.
- Operand: $[\![\text{ctxt}]\!]\{\%id \to t\} \vdash \%id : t$.

Q translate $\Gamma \vdash e_1 + e_2 : \text{int}$ (Add).

- let $([\![ctxt]\!] \vdash s_1 \Rightarrow \Gamma_2, \Gamma_1 \vdash opn_1 : i64) = [\![e_1]\!](ctxt)$.
- let $([\![ctxt]\!] \vdash s_2 \Rightarrow \Gamma_2, \Gamma_2 \vdash opn_2 : i64) = [\![e_2]\!](ctxt)$.
- codestream = $\Gamma_1 + \Gamma_2 \vdash s_1, s_2, \%uid = \text{add } i64\ opn_1, opn_2$.
  $\Longrightarrow (\Gamma_1 + \Gamma_2)\{\%uid \to i64\}$.
- Operand: $(\Gamma_1 + \Gamma_2)\{\%uid \mapsto i64\} \vdash \%uid : i64$.

Tedious — but this is what a compiler does.

Summary:

- Semantic analysis - takes AST as input
  - constructs symbol table
  - performs well-formedness checks

- well-formedness derivations impact compilation.
  eg: ① x.field gets compiled differently depending on type of α

  ② Might have to emit bitcasts for subsumption.
  (LLVM type system does not have subtyping).

Summary, cont

- Tedious: Compiler translates derivations of well-formedness judgements in source language to derivations of well-formedness judgements in target language.

→ In an implementation, this viewpoint is implicit.
  - Don't need to do all the bookkeeping involved in manipulating derivations.
  - But it is helpful for understanding how