COS320: Compiling Techniques

Zak Kincaid

March 26, 2024

Logistics

- HW3 due today
- HW4 released today, due April 11th. You will implement a typechecker and translator for an extension of Oat.

Oat v2

- Specified by a (fairly large) type system
  - ∼20 judgements, ∼80 inference rules
  - Invest some time in making sure you understand how to read them
- Adds several features to the Oat language:
  - Memory safety
    - *nullable* and *non-null* references. Type system enforces no null pointer dereferences.
    - Run-time array bounds checking (like Java, OCaml)
  - Mutable record types
  - Subtyping
    - *ref* <: *ref?*: non-null references are a subtype of nullable references
    - Record subtyping: width but not depth (why?)

Compiling with Types

- Intrinsic view: an ill-typed program is not a program at all
- Compiler translates programs in the source language to programs in the target language
  - Well-typed source programs translate to well-typed target programs

- Intrinsic view: an ill-typed program is not a program at all
- Compiler translates programs in the source language to programs in the target language
    - Well-typed source programs translate to well-typed target programs
    - Compiler may reject ill-typed source programs

- Intrinsic view: an ill-typed program is not a program at all
- Compiler translates programs in the source language to programs in the target language
  - Well-typed source programs translate to well-typed target programs
  - Compiler may reject ill-typed source programs
  - Compiler must ensure that target program is well-typed

- Intrinsic view: an ill-typed program is not a program at all
- Compiler translates programs in the source language to programs in the target language
  - Well-typed source programs translate to well-typed target programs
  - Compiler may reject ill-typed source programs
  - Compiler must ensure that target program is well-typed
- IR may also have its own type system (LLVM)
  - Your backend does not check types, but does throw exceptions for (some) ill-typed programs
  - LLVM does check types: use `--clang` to check that your front-end produces type-correct code

We can think of compilation as translation of derivations of judgements from a source language to a target language

- Each kind of judgement has a different translation category. E.g.,
  - Well-formed types in source become well-formed types in target
  - Expressions in source become (operand, instruction list) pairs in target
  - ...
- Each inference rule corresponds to a case within that category

Oat v1 (HW3) – well-formed types

Judgements take the form:

- $\vdash t$: "t is a well-formed type" (`ty`)
- $\vdash_r \textit{ref}$: "*ref* is a well-formed reference type" (`rty`)
- $\vdash_{rt} \textit{rt}$: "*rt* is a well-formed return type" (`ret_ty`)

TINT
$$\frac{}{\vdash \texttt{int}}$$

TBOOL
$$\frac{}{\vdash \texttt{bool}}$$

TREF
$$\frac{\vdash_r \textit{ref}}{\vdash \textit{ref}}$$

RSTRING
$$\frac{}{\vdash_r \texttt{string}}$$

RARRAY
$$\frac{\vdash t}{\vdash_r t\texttt{[]}}$$

RFUN
$$\frac{\vdash t_1 \quad \dots \quad \vdash t_n \quad \vdash_{rt} \textit{rt}}{\vdash_r (t_1, \dots, t_n) \rightarrow \textit{rt}}$$

RTVOID
$$\frac{}{\vdash_{rt} \texttt{void}}$$

RTTYP
$$\frac{\vdash t}{\vdash_{rt} t}$$

LLVMlite well-formed types

Judgements take the form:

- $T \vdash t$: With named types T, t is a well-formed type
- $T \vdash_s t$: With named types T, t is a well-formed simple type
- $T \vdash_r t$: With named types T, t is a well-formed reference type
- $T \vdash_{rt} t$: With named types T, t is a well-formed return type

LLBOOL
$$\frac{}{T \vdash_s \texttt{i1}}$$

LLINT
$$\frac{}{T \vdash_s \texttt{i64}}$$

LLPTR
$$\frac{T \vdash_r \textit{ref}}{T \vdash_s \textit{ref}*}$$

LLTUPLE
$$\frac{T \vdash t_1 \qquad \ldots \qquad T \vdash t_n}{T \vdash \{t_1, \ldots, t_n\}}$$

LLARRAY
$$\frac{T \vdash t}{T \vdash \texttt{[}n \texttt{ x } t\texttt{]}} \quad n \in \mathbb{N}$$

LLSIMPLE
$$\frac{\vdash_s t}{\vdash t}$$

LLRTVOID
$$\frac{}{T \vdash_{rt} \texttt{void}}$$

LLRTSIMPLE
$$\frac{T \vdash_s t}{T \vdash_{rt} t}$$

LLRCHAR
$$\frac{}{T \vdash_r \texttt{i8}}$$

LLRTYPE
$$\frac{T \vdash t}{T \vdash_r t}$$

LLRFUN
$$\frac{T \vdash_{rt} \textit{rt} \qquad T \vdash_s t_1 \qquad \ldots \qquad T \vdash_s t_n}{T \vdash_r \textit{rt}(t_1, \ldots, t_n)}$$

LLNAMED
$$\frac{}{T \vdash \%uid} \quad \%uid \in T$$

Translating well-formed types

- Each well-formed Oat type is translated to a well-formed LLVM type
  - types $\rightarrow$ simple types (`cmp_ty`)
  - reference types $\rightarrow$ reference types (`cmp_rty`)
  - return types $\rightarrow$ return types (`cmp_ret_ty`)
- Use $\rightsquigarrow$ to denote translation of derivations

Translating well-formed types

Suppose we have a well-formed type Oat type, $\vdash t$. There are three inference rules:

$$\textsc{TInt}\ \frac{}{\vdash \texttt{int}} \qquad \textsc{TBool}\ \frac{}{\vdash \texttt{bool}} \qquad \textsc{TRef}\ \frac{\vdash_r \textit{ref}}{\vdash \textit{ref}}$$

Each has a corresponding case:

- $\left(\textsc{TInt}\ \frac{}{\vdash \texttt{int}}\right) \rightsquigarrow \left(\textsc{LLInt}\ \frac{}{\vdash_s \texttt{i64}}\right)$
- $\left(\textsc{TBool}\ \frac{}{\vdash \texttt{bool}}\right) \rightsquigarrow \left(\textsc{LLBool}\ \frac{}{\vdash_s \texttt{i1}}\right)$

Translating well-formed types

Suppose we have a well-formed type Oat type, $\vdash t$. There are three inference rules:

$$\textsc{TInt}\ \frac{}{\vdash \texttt{int}} \qquad \textsc{TBool}\ \frac{}{\vdash \texttt{bool}} \qquad \textsc{TRef}\ \frac{\vdash_r \textit{ref}}{\vdash \textit{ref}}$$

Each has a corresponding case:

- $\left(\textsc{TInt}\ \frac{}{\vdash \texttt{int}}\right) \rightsquigarrow \left(\textsc{LLInt}\ \frac{}{\vdash_s \texttt{i64}}\right)$
- $\left(\textsc{TBool}\ \frac{}{\vdash \texttt{bool}}\right) \rightsquigarrow \left(\textsc{LLBool}\ \frac{}{\vdash_s \texttt{i1}}\right)$
- $\left(\textsc{TRef}\ \frac{\vdash_r \textit{ref}}{\vdash \textit{ref}}\right) \rightsquigarrow \left(\textsc{LLPtr}\ \frac{\vdash_r t}{\vdash_s t*}\right)$, where $(\vdash_r \textit{ref}) \rightsquigarrow (\vdash_r t)$

Translating well-formed array types

- In Oat v2, arrays accesses are checked at runtime
- **Recall**: Can implement run-time array access checking by allocating additional memory at the beginning of the array to store its size
- In Oat v1, arrays accesses are unchecked, but for forwards-compatibility we represent arrays in the same way.

Translating well-formed array types

- In Oat v2, arrays accesses are checked at runtime
- **Recall**: Can implement run-time array access checking by allocating additional memory at the beginning of the array to store its size
- In Oat v1, arrays accesses are unchecked, but for forwards-compatibility we represent arrays in the same way.

$$\textsc{RArray}\ \frac{\vdash t}{\vdash_r t\texttt{[]}} \rightsquigarrow \textsc{LLRType}\ \dfrac{\textsc{LLTuple}\ \dfrac{\textsc{LLSimple}\ \dfrac{\textsc{LLInt}\ \dfrac{}{\vdash_s \texttt{i64}}}{\vdash \texttt{i64}} \quad \textsc{LLArray}\ \dfrac{\textsc{LLSimple}\ \dfrac{\vdash_s t'}{\vdash t'}}{\vdash \texttt{[0x}t'\texttt{]}}}{\vdash \{\texttt{i64}, \texttt{[0x}t'\texttt{]}\}}}{\vdash_r \{\texttt{i64}, \texttt{[0x}t'\texttt{]}\}}$$

where $\vdash t \rightsquigarrow \vdash_s t'$

Summary of type translation

Succint notation: $[\![\vdash J]\!] = J'$ denotes that a derivation with root J translates to a derivation with root J'

- $[\![\vdash \texttt{int}]\!] = \vdash_s \texttt{i64}$
- $[\![\vdash \texttt{bool}]\!] = \vdash_s \texttt{i1}$
- $[\![\vdash \textit{ref}]\!] = \vdash_s t*$, where $\vdash_r t$ = $[\![\vdash_r \textit{ref}]\!]$
- $[\![\vdash_r \texttt{string}]\!] = \vdash_r \texttt{i8}$
- $[\![\vdash_r t\texttt{[]}]\!] = \vdash_r \{\texttt{i64}, \texttt{[0x}t'\texttt{]}\}$, where $\vdash_s t' = [\![\vdash t]\!]$
- $[\![\vdash_r (t_1, \ldots, t_n) \to rt]\!] = \vdash_{rt} \textit{rt}'(t'_1, \ldots, t'_n)$, where
  - $\vdash_{rt} \textit{rt} = [\![\vdash_{rt} rt]\!]$,
  - $\vdash_s t'_1 = [\![\vdash t_1]\!]$, ..., $\vdash_s t'_n = [\![\vdash t_n]\!]$
- $[\![\vdash_{rt} \texttt{void}]\!] = \vdash_{rt} \texttt{void}$
- $[\![\vdash_{rt} t]\!] = \vdash_{rt} t$, where $\vdash_s t = [\![\vdash t]\!]$

(see: `cmp_ty`, `cmp_rty`, `cmp_ret_ty` in HW3)

Well-formed codestreams

Judgements take the form

- $\Gamma \vdash s \Rightarrow \Gamma'$: "under type environment Γ, code stream s is well-formed and results in type environment Γ'"
- $\Gamma \vdash opn : t$: "under type environment Γ, operand *opn* has type t"

Id

$$\frac{}{\Gamma \vdash id : t}\ \Gamma(id) = t$$

Num

$$\frac{}{\Gamma \vdash n : \texttt{i64}}\ n \in \mathbb{Z}$$

Add

$$\frac{\Gamma \vdash \mathit{opn}_1 : \texttt{i64} \qquad \Gamma \vdash \mathit{opn}_2 : \texttt{i64}}{\Gamma \vdash \%uid = \mathsf{add}\ \texttt{i64}\ \mathit{opn}_1, \mathit{opn}_2 \Rightarrow \Gamma\{\%uid \mapsto \texttt{i64}\}}\ \%uid \notin dom(\Gamma)$$

Seq

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

Base

$$\frac{}{\Gamma \vdash \epsilon \Rightarrow \Gamma}$$

... lots more

Well-typed expressions

$$\textsc{Var}\quad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{Add}\quad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \ldots$$

Expression compilation (`cmp_exp`) translates a type judgement $\Gamma \vdash e : t$ to

- A codestream judgement $\Gamma_{ll} \vdash s \Rightarrow \Gamma'_{ll}$, and
- An operand judgement $\Gamma'_{ll} \vdash$ *opn* $: t_{ll}$

How can translate $\Gamma \vdash x : t$ (i.e., VAR)?

How can translate $\Gamma \vdash x : t$ (i.e., VAR)?

- Need a symbol table `ctxt`, which maps Oat identifiers to LLVMlite operand judgements
  - The operand associated with a variable x is a *pointer* to the memory location associated with x

How can translate $\Gamma \vdash x : t$ (i.e., VAR)?

- Need a symbol table ctxt, which maps Oat identifiers to LLVMlite operand judgements
  - The operand associated with a variable x is a *pointer* to the memory location associated with x
- To compute $[\![\Gamma \vdash x : t]\!](\texttt{ctxt})$, first let $(id, t*) = \texttt{ctxt}(x)$, then:
  - Define $[\![\texttt{ctxt}]\!]$ to be the (LLVM) type environment associated with ctxt
    - $[\![\epsilon]\!] = \epsilon$ (empty context translates to empty context)
    - $[\![\texttt{ctxt}, x \mapsto (id, t)]\!] = \Gamma_{ll}, id \mapsto t$, where $[\![\texttt{ctxt}]\!] = \Gamma_{ll}$
  - Codestream: $[\![\texttt{ctxt}]\!] \vdash \%uid = \texttt{load}\ \ t,\ \ t*\ \ id \Rightarrow [\![\texttt{ctxt}]\!]\{\%uid \mapsto t\}$
  - Operand: $[\![\texttt{ctxt}]\!]\{\%uid \mapsto t\} \vdash \%uid : t$

How can translate $\Gamma \vdash x : t$ (i.e., VAR)?

- Need a symbol table $\texttt{ctxt}$, which maps Oat identifiers to LLVMlite operand judgements
  - The operand associated with a variable x is a *pointer* to the memory location associated with x
- To compute $[\![\Gamma \vdash x : t]\!](\texttt{ctxt})$, first let $(id, t*) = \texttt{ctxt}(x)$, then:
  - Define $[\![\texttt{ctxt}]\!]$ to be the (LLVM) type environment associated with $\texttt{ctxt}$
    - $[\![\epsilon]\!] = \epsilon$ (empty context translates to empty context)
    - $[\![\texttt{ctxt}, x \mapsto (id, t)]\!] = \Gamma_{ll}, id \mapsto t$, where $[\![\texttt{ctxt}]\!] = \Gamma_{ll}$
  - Codestream: $[\![\texttt{ctxt}]\!] \vdash \%uid = \texttt{load}\ \ t,\ \ t*\ \ id \Rightarrow [\![\texttt{ctxt}]\!]\{\%uid \mapsto t\}$
  - Operand: $[\![\texttt{ctxt}]\!]\{\%uid \mapsto t\} \vdash \%uid : t$

How can we translate $\Gamma \vdash e_1 + e_2 : \texttt{int}$ (i.e., ADD)?

How can translate $\Gamma \vdash x : t$ (i.e., VAR)?

- Need a symbol table `ctxt`, which maps Oat identifiers to LLVMlite operand judgements
  - The operand associated with a variable x is a *pointer* to the memory location associated with x
- To compute $[\![\Gamma \vdash x : t]\!](\texttt{ctxt})$, first let $(id, t*) = \texttt{ctxt}(x)$, then:
  - Define $[\![\texttt{ctxt}]\!]$ to be the (LLVM) type environment associated with `ctxt`
    - $[\![\epsilon]\!] = \epsilon$ (empty context translates to empty context)
    - $[\![\texttt{ctxt}, x \mapsto (id, t)]\!] = \Gamma_{ll}, id \mapsto t$, where $[\![\texttt{ctxt}]\!] = \Gamma_{ll}$
  - Codestream: $[\![\texttt{ctxt}]\!] \vdash \%uid = \texttt{load}\ t,\ t*\ id \Rightarrow [\![\texttt{ctxt}]\!]\{\%uid \mapsto t\}$
  - Operand: $[\![\texttt{ctxt}]\!]\{\%uid \mapsto t\} \vdash \%uid : t$

How can we translate $\Gamma \vdash e_1 + e_2 : \texttt{int}$ (i.e., ADD)?

- Let $([\![\texttt{ctxt}]\!] \vdash s_1 \Rightarrow \Gamma_1, \Gamma_1 \vdash opn_1 : \texttt{i64}) = [\![e_1]\!](\texttt{ctxt})$
- Let $(\Gamma_1 \vdash s_2 \Rightarrow \Gamma_2, \Gamma_2 \vdash opn_2 : \texttt{i64}) = [\![e_2]\!](\texttt{ctxt})$
- Codestream: $[\![\texttt{ctxt}]\!] \vdash s_1, s_2, \%uid = \texttt{add i64}\ \textit{opn}_1, \textit{opn}_2) \Rightarrow \Gamma_2\{\%uid \mapsto \texttt{i64}\}$
- Operand: $\Gamma_2\{\%uid \mapsto \texttt{i64}\} \vdash \%uid : \texttt{i64}$

Summary

- *Semantic analysis* phase takes AST as input, constructs symbol table and performs well-formedness checks

Summary

- *Semantic analysis* phase takes AST as input, constructs symbol table and performs well-formedness checks
- Well-formedness derivations can impact compilation. E.g.,
  - `x.field` gets compiled differently depending on the type of `x`
  - We may have to emit bitcasts for uses of subsumption

Summary

- *Semantic analysis* phase takes AST as input, constructs symbol table and performs well-formedness checks
- Well-formedness derivations can impact compilation. E.g.,
  - `x.field` gets compiled differently depending on the type of `x`
  - We may have to emit bitcasts for uses of subsumption
- Compiler translates derivations of well-formedness judgements in the source language to derivations of well-formedness judgements in the target language
  - In an implementation, this viewpoint *implicit*
    - Don't need to do all the bookkeeping involved in manipulating derivations
  - *But* it is helpful for understanding how to organize the translation
    - E.g., `cmp_exp` returns a triple `Ll.ty * Ll.operand * stream`
      In a sense: infers derivations in the source language "on the way down"
      builds derivations in the target language "on the way up"
      Only remembers the type of the operand (used in some compilation rules).