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
  - \(\sim\)20 judgements, \(\sim\)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
    - \(\text{ref} \ll\text{ref}?:\) 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
Judgements take the form:

- $\vdash t$: “$t$ is a well-formed type” ($\text{ty}$)
- $\vdash_r \text{ref}$: “$\text{ref}$ is a well-formed reference type” ($\text{rty}$)
- $\vdash_{rt} \text{rt}$: “$\text{rt}$ is a well-formed return type” ($\text{ret\_ty}$)

\[
\begin{align*}
&T\text{INT} & & T\text{BOOL} & & T\text{REF} \\
&\vdash \text{int} & & \vdash \text{bool} & & \vdash \text{ref} \\
&R\text{STRING} \\
&\vdash_r \text{string} & &\vdash_r t[] \\
&R\text{ARRAY} \\
&\vdash t & & \vdash t_1 \ldots \vdash t_n & & \vdash_{rt} \text{rt} \\
&R\text{FUN} \\
&\vdash (t_1, \ldots, t_n) \rightarrow \text{rt} \\
&R\text{VOID} \\
&\vdash_{rt} \text{void} \\
&R\text{TYPE} \\
&\vdash t & & \vdash_{rt} t
\end{align*}
\]
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

\[
\begin{array}{cccccc}
\text{LLBOOL} & \text{LLINT} & \text{LLPTR} & \text{LLTUPLE} & \text{LLARRAY} \\
T \vdash_s i1 & T \vdash_s i64 & T \vdash_r \text{ref*} & T \vdash t_1 \ldots T \vdash t_n & T \vdash [t \times n] \quad n \in \mathbb{N} \\
\hline
\text{LLSIMPLE} & \vdash_s t & \vdash t \\
\text{LLRTVOID} & \vdash t \vdash_r \text{void} \\
\text{LLRTSIMPLE} & T \vdash_r t \\
\text{LLRCHAR} & T \vdash r i8 \\
\text{LLRTTYPE} & T \vdash_r t \\
\text{LLRFUN} & T \vdash_r t \quad T \vdash s t_1 \ldots T \vdash s t_n \\
\hline
\text{LLNAMED} & \quad T \vdash \%u \text{id} \quad \%u \text{id} \in T
\end{array}
\]
Translating well-formed types

- Each well-formed Oat type is translated to a well-formed LLVM type
  - types $\rightarrow$ simple types ($\text{cmp\_ty}$)
  - reference types $\rightarrow$ reference types ($\text{cmp\_rty}$)
  - return types $\rightarrow$ return types ($\text{cmp\_ret\_ty}$)

- Use $\rightsquigarrow$ to denote translation of derivations
Translating well-formed types

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

- **TINT**
  
  $\Gamma \vdash \text{int}$

- **TBOOL**
  
  $\Gamma \vdash \text{bool}$

- **TREF**
  
  $\Gamma \vdash \text{ref}$

Each has a corresponding case:

- $\left( \frac{\text{TINT} \quad \Gamma \vdash \text{int}}{} \right) \leadsto \left( \frac{\text{LLINT} \quad s \vdash \text{i64}}{} \right)$

- $\left( \frac{\text{TBOOL} \quad \Gamma \vdash \text{bool}}{} \right) \leadsto \left( \frac{\text{LLBOOL} \quad s \vdash \text{i1}}{} \right)$
Translating well-formed types

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

- **TINT**
  \[
  \frac{\vdash \texttt{int}}{\vdash \texttt{int}}
  \]

- **TBOOL**
  \[
  \frac{\vdash \texttt{bool}}{\vdash \texttt{bool}}
  \]

- **TREF**
  \[
  \frac{\vdash_r \texttt{ref}}{\vdash \texttt{ref}}
  \]

Each has a corresponding case:

- **TINT**
  \[
  \left( \frac{\vdash \texttt{int}}{} \right) \rightsquigarrow \left( \frac{\vdash \texttt{i64}}{s} \right)
  \]

- **TBOOL**
  \[
  \left( \frac{\vdash \texttt{bool}}{} \right) \rightsquigarrow \left( \frac{\vdash \texttt{i1}}{s} \right)
  \]

- **TREF**
  \[
  \left( \frac{\vdash_r \texttt{ref}}{} \right) \rightsquigarrow \left( \frac{\vdash \texttt{t}}{s} \right)
  \]

where \((\vdash_r \texttt{ref}) \rightsquigarrow (\vdash_r \texttt{t})\)
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.

\[
\begin{align*}
\text{RArray} & \vdash t \\
\text{LLTuple} & \vdash i64 \\
\text{LLRTYPE} & \vdash \{i64, [0x{t'}]\} \\
\text{LLSimple} & \vdash_s t' \\
\text{LLArray} & \vdash [0x{t'}]
\end{align*}
\]

where \( \vdash t \leadsto \vdash_s t' \)
Summary of type translation

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

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

(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 $\Gamma$, code stream $s$ is well-formed and results in type environment $\Gamma'$”
- $\Gamma \vdash \text{opn} : t$: “under type environment $\Gamma$, operand $\text{opn}$ has type $t$”

\[
\begin{align*}
\text{Id} & \quad \text{Num} \\
\Gamma \vdash id : t & \quad \Gamma \vdash n : \text{i64} \\
\Gamma(id) = t & \quad n \in \mathbb{Z}
\end{align*}
\]

\[
\begin{align*}
\text{ADD} & \\
\Gamma \vdash \text{opn}_1 : \text{i64} & \quad \Gamma \vdash \text{opn}_2 : \text{i64} \\
\Gamma \vdash \%\text{uid} = \text{add i64 } \text{opn}_1, \text{opn}_2 & \Rightarrow \Gamma\{\%\text{uid} \mapsto \text{i64}\} \\
\text{%uid} \not\in \text{dom}(\Gamma)
\end{align*}
\]

\[
\begin{align*}
\text{SEQ} & \\
\Gamma \vdash s_1 \Rightarrow \Gamma' & \quad \Gamma' \vdash s_2 \Rightarrow \Gamma'' \\
\Gamma \vdash s_1, s_2 \Rightarrow \Gamma''
\end{align*}
\]

\[
\begin{align*}
\text{BASE} & \\
\Gamma \vdash \epsilon \Rightarrow \Gamma
\end{align*}
\]

...lots more
Well-typed expressions

\[
\begin{align*}
\text{VAR} & \quad \text{ADD} \\
\Gamma \vdash x : \Gamma(x) & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\quad & \quad \Gamma \vdash e_1 + e_2 : \text{int}
\end{align*}
\]

Expression compilation \((\text{cmp}\_\text{exp})\) translates a type judgement \(\Gamma \vdash e : t\) to

- A codestream judgement \(\Gamma_u \vdash s \Rightarrow \Gamma'_u\), and
- An operand judgement \(\Gamma'_u \vdash \text{opn} : t_u\)
How can translate $\Gamma \vdash x : t$ (i.e., VAR)?
How can translate $\Gamma \vdash x : t$ (i.e., VAR)?

- Need a symbol table $\text{ctxt}$, 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 $\llbracket \Gamma \vdash x : t \rrbracket (ctxt)$, first let $(id, t\star) = ctxt(x)$, then:
  - Define $\llbracket ctxt \rrbracket$ to be the (LLVM) type environment associated with $ctxt$
    
    - $\llbracket \epsilon \rrbracket = \epsilon$ (empty context translates to empty context)
    
    - $\llbracket ctxt, x \mapsto (id, t) \rrbracket = \Gamma_u, id \mapsto t$, where $\llbracket ctxt \rrbracket = \Gamma_u$

- **Codestream:** $\llbracket ctxt \rrbracket \vdash %uid = load \ t, \ t\star \ id \Rightarrow \llbracket ctxt \rrbracket \{ %uid \mapsto t \}$
- **Operand:** $\llbracket ctxt \rrbracket \{ %uid \mapsto t \} \vdash %uid : \ t$
How can translate $\Gamma \vdash x : t$ (i.e., VAR)?

- Need a symbol table $\text{ctxt}$, 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 $[[\Gamma \vdash x : t]](\text{ctxt})$, first let $(id, t\ast) = \text{ctxt}(x)$, then:
  - Define $[[\text{ctxt}]]$ to be the (LLVM) type environment associated with $\text{ctxt}$
    - $[[\epsilon]] = \epsilon$ (empty context translates to empty context)
    - $[[\text{ctxt}, x \mapsto (id, t)]] = \Gamma_u, id \mapsto t$, where $[[\text{ctxt}]] = \Gamma_u$
  - Codestream: $[[\text{ctxt}]] \vdash \%uid = \text{load } t, t\ast id \Rightarrow [[\text{ctxt}]]\{\%uid \mapsto t\}$
  - Operand: $[[\text{ctxt}]]\{\%uid \mapsto t\} \vdash \%uid : t$

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

- Need a symbol table $\text{ctxt}$, 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 $[[\Gamma \vdash x : t]](\text{ctxt})$, first let $(id, t*) = \text{ctxt}(x)$, then:
  - Define $[[\text{ctxt}]]$ to be the (LLVM) type environment associated with $\text{ctxt}$
    - $[[\epsilon]] = \epsilon$ (empty context translates to empty context)
    - $[[\text{ctxt}, x \mapsto (id, t)]] = \Gamma_u, id \mapsto t$, where $[[\text{ctxt}]] = \Gamma_u$
  - Codestream: $[[\text{ctxt}]] \vdash \%uid = \text{load } t, t* id \Rightarrow [[\text{ctxt}]]\{\%uid \mapsto t\}$
  - Operand: $[[\text{ctxt}]]\{\%uid \mapsto t\} \vdash \%uid : t$

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

- Let $([[\text{ctxt}]] \vdash s_1 \Rightarrow \Gamma_1, \Gamma_1 \vdash opn_1 : \text{i64}) = [[e_1]](\text{ctxt})$
- Let $(\Gamma_1 \vdash s_2 \Rightarrow \Gamma_2, \Gamma_2 \vdash opn_2 : \text{i64}) = [[e_2]](\text{ctxt})$
- Codestream: $[[\text{ctxt}]] \vdash s_1, s_2, \%uid = \text{add } \text{i64 } opn_1, opn_2 \Rightarrow \Gamma_2\{\%uid \mapsto \text{i64}\}$
- Operand: $\Gamma_2\{\%uid \mapsto \text{i64}\} \vdash \%uid : \text{i64}$
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 `L1.ty * L1.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).