COS320: Compiling Techniques

Zak Kincaid

April 6, 2019
Compiling with Types
• Intrinsic view: a term that cannot be typed is not a term at all
  • Compiler cannot translate terms that cannot be typed
• If target language is typed, this imposes an additional burden:
  • Well-typed programs in the source language translate to well-typed programs in the target language
• Intrinsic view: a term that cannot be typed is not a term at all
  • Compiler cannot translate terms that cannot be typed
• If target language is typed, this imposes an additional burden:
  • Well-typed programs in the source language translate to well-typed programs in the target language
• 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”
- \( \vdash_r ref \): “\( ref \) is a well-formed reference type”
- \( \vdash_{rt} rt \): “\( rt \) is a well-formed return type”

\[\begin{array}{ccc}
\text{TINT} & \text{TBOOL} & \text{TREF} \\
\hline \\
\vdash \text{int} & \vdash \text{bool} & \vdash_r \text{ref} \\
\hline \\
\text{RSTRING} & \text{RAARRAY} & \text{RFUN} \\
\hline \\
\vdash_r \text{string} & \vdash_r t[] & \vdash_r (t_1, \ldots, t_n) \rightarrow rt \\
\hline \\
\text{RTVOID} & \text{RTTYP} \\
\hline \\
\vdash_{rt} \text{void} & \vdash_{rt} t \\
\end{array}\]
# 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Judgement</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{LLBOOL}</td>
<td>( T \vdash s i1 )</td>
<td>( T \vdash s \text{ref*} )</td>
</tr>
<tr>
<td>\text{LLINT}</td>
<td>( T \vdash s i64 )</td>
<td>( T \vdash { t_1, \ldots, t_n } )</td>
</tr>
<tr>
<td>\text{LLPTR}</td>
<td>( T \vdash_{rt} \text{ref} )</td>
<td>( T \vdash [n \times t] ) ( n \in \mathbb{N} )</td>
</tr>
<tr>
<td>\text{LLTUPLE}</td>
<td>( T \vdash t_1 )</td>
<td>( T \vdash_s t )</td>
</tr>
<tr>
<td>\text{LLARRAY}</td>
<td>( T \vdash_t )</td>
<td>( T \vdash_s r \text{t} )</td>
</tr>
<tr>
<td>\text{LLRVOID}</td>
<td>( T \vdash_{rt} \text{void} )</td>
<td>( T \vdash_s t )</td>
</tr>
<tr>
<td>\text{LLRTSIMPLE}</td>
<td>( T \vdash_s t )</td>
<td>( T \vdash_s t )</td>
</tr>
<tr>
<td>\text{LLRFCAST}</td>
<td>( T \vdash_t )</td>
<td>( T \vdash_r \text{ref} ) ( t_1, \ldots, t_n )</td>
</tr>
<tr>
<td>\text{LLRFUN}</td>
<td>( T \vdash_{rt} \text{t} )</td>
<td>( T \vdash_s t )</td>
</tr>
<tr>
<td>\text{LLNAMED}</td>
<td>( T \vdash %uid )</td>
<td>( %uid \in T )</td>
</tr>
</tbody>
</table>
Translating well-formed types

- Each well-formed Oat type is translated to a well-formed LLVM type
  - types → simple types
  - reference types → reference types
  - return types → return types
- Use \([\cdot]\) to denote translation
  - i.e., \([\leftarrow \text{int}] =\leftarrow_s \text{i64}\) denotes that the Oat type int is translated to the (simple) LLVMlite type i64
Translating well-formed types

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

\[
\begin{array}{ccc}
\text{TINT} & \text{TBOOL} & \text{TREF} \\
\hline
\vdash \text{int} & \vdash \text{bool} & \vdash \text{ref} \\
\end{array}
\]

Each has a corresponding case:

- Case TINT: \( \vdash \text{int} = \vdash s \text{ i64} \) (well-formed by LLInt)
- Case TBOOL: \( \vdash \text{bool} = \vdash s \text{ i1} \) (well-formed by LLBool)
Translating well-formed types

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

\[
\begin{align*}
\text{TINT} & \quad \quad \text{TBOOL} & \quad \quad \text{TREF} \\
\hline
\vdash \text{int} & \quad \quad \vdash \text{bool} & \quad \quad \vdash \text{ref} \\
\end{align*}
\]

Each has a corresponding case:

- Case TINT: \( \llbracket \vdash \text{int} \rrbracket = \vdash_s \text{i64} \) (well-formed by LLInt)
- Case TBOOL: \( \llbracket \vdash \text{bool} \rrbracket = \vdash_s \text{i1} \) (well-formed by LLBool)
- Case TREF: By induction on the derivation, \( \llbracket \vdash \text{ref} \rrbracket \) is a valid judgement of an LLVM reference type, say \( \vdash_r t \)

\[
\begin{align*}
\text{TREF} \quad \text{LLPTR} \\
\vdash_r \text{ref} \quad \vdash_r t (= \llbracket \vdash_r \text{ref} \rrbracket) \\
\hline
\vdash \text{ref} \rightarrow \vdash_r t \\
\end{align*}
\]

\( \text{l.e., } \llbracket \vdash \text{ref} \rrbracket = \vdash_s t* \), where \( \vdash_r t = \llbracket \vdash_r \text{ref} \rrbracket \)
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.
Summary of type translation

- \( \llbracket \text{int} \rrbracket = \llbracket s \rrbracket \text{i64} \)
- \( \llbracket \text{bool} \rrbracket = \llbracket s \rrbracket \text{i1} \)
- \( \llbracket \text{ref} \rrbracket = \llbracket s \rrbracket \text{t*}, \text{where} \llbracket r \rrbracket t = \llbracket r \rrbracket \text{ref} \)
- \( \llbracket r \rrbracket \text{string} = \llbracket r \rrbracket \text{i8} \)
- \( \llbracket r \rrbracket t[\text{]}] = \llbracket r \rrbracket \{ \text{i64}, [0 \times t'] \}, \text{where} \llbracket s \rrbracket t' = \llbracket t \rrbracket \)
- \( \llbracket r \rrbracket (t_1, ..., t_n) \rightarrow r t\rrbracket = \llbracket r t \rrbracket r t'(t'_1, ..., t'_{n}), \text{where} \)
  - \( \llbracket r t \rrbracket r t = \llbracket r t \rrbracket r t, \)
  - \( \llbracket s \rrbracket t'_1 = \llbracket t_1 \rrbracket, ..., \llbracket s \rrbracket t'_n = \llbracket t_n \rrbracket \)
- \( \llbracket r t \rrbracket \text{void} = \llbracket r t \rrbracket \text{void} \)
- \( \llbracket r t \rrbracket t\rrbracket = \llbracket r t \rrbracket t, \text{where} \llbracket s \rrbracket t = \llbracket t \rrbracket \)

(see: cmp_ty, cmp_rty, cmp_ret_ty in HW4)
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 $t$”

\[
\frac{}{\Gamma \vdash \text{id} : \Gamma(\text{id})}
\]

\[
\frac{}{\Gamma \vdash n : \text{i64} \quad n \in \mathbb{Z}}
\]

\[
\frac{\Gamma \vdash \text{opn}_1 : \text{i64} \quad \Gamma \vdash \text{opn}_2 : \text{i64}}{T, \Gamma \vdash \%\text{uid} = \text{add} \ 164 \ \text{opn}_1, \text{opn}_2 \Rightarrow \Gamma\{\%\text{uid} \mapsto \text{i64}\} \quad \%\text{uid} \notin \text{dom}(\Gamma)}
\]

\[
\frac{T, \Gamma \vdash s_1 \Rightarrow \Gamma' \quad T, \Gamma' \vdash s_2 \Rightarrow \Gamma''}{T, \Gamma \vdash s_1, s_2 \Rightarrow \Gamma''}
\]

\[
\frac{T, \Gamma \vdash \epsilon \Rightarrow \Gamma}{T, \Gamma \vdash \epsilon \Rightarrow \Gamma} \quad \text{...lots more}
\]
Well-typed expressions

\[
\text{VAR} \quad \text{ADD} \\
\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\]

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

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

- Need a symbol table $\text{ctxt}$, 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 $\llbracket \Gamma \vdash x : t \rrbracket (\text{ctxt})$, first let $(id, t*) = \text{ctxt}(x)$, then:
  - Define $\llbracket \text{ctxt} \rrbracket$ to be the type environment associated with $\text{ctxt}$
  - Codestream: $\llbracket \text{ctxt} \rrbracket \vdash %uid = \text{load} t* \; \text{opn} \Rightarrow \llbracket \text{ctxt} \rrbracket\{ %uid \mapsto t \}$
  - Operand: $\llbracket \text{ctxt} \rrbracket\{ %uid \mapsto t \} \vdash %uid : t$
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 $(id, t*) = \text{ctxt}(x)$, then:
  - Define $[\text{ctxt}]$ to be the type environment associated with $\text{ctxt}$
  - Codestream: $[\text{ctxt}] \vdash %\text{uid} = \text{load} \ t* \text{ opn} \Rightarrow [\text{ctxt}]{%\text{uid} \mapsto t}$
  - Operand: $[\text{ctxt}]{%\text{uid} \mapsto t} \vdash %\text{uid} : t$

How can 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 LLVM-lite 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 type environment associated with $\text{ctxt}$.
  - Codestream: $[[\text{ctxt}]] \vdash \%uid = \text{load} \ t* \ \text{opn} \Rightarrow [[\text{ctxt}]]\{\%uid \mapsto t\}$
  - Operand: $[[\text{ctxt}]]\{\%uid \mapsto t\} \vdash \%uid : t$

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

- Let $([[\text{ctxt}]] \vdash s_1 \Rightarrow \Gamma_1, \Gamma_1 \vdash \text{opn}_1 : \text{i64}) = [[e_1]](\text{ctxt})$
- Let $([[\text{ctxt}]] \vdash s_2 \Rightarrow \Gamma_2, \Gamma_2 \vdash \text{opn}_2 : \text{i64}) = [[e_2]](\text{ctxt})$
- Codestream: $\Gamma_1 + \Gamma_2 \vdash s_1, s_2, \%uid = \text{add} \ \text{i64} \ \text{opn}_1, \text{opn}_2) \Rightarrow (\Gamma_1 + \Gamma_2)\{\%uid \mapsto \text{i64}\}$
- Operand: $(\Gamma_1 + \Gamma_2)\{\%uid \mapsto \text{i64}\} \vdash \%uid : \text{i64}$
Global initializers

One would expect the following coherence property:

\[ \Gamma \vdash e : t \text{ translates to the codestream judgement } \Gamma_{ul} \vdash s \Rightarrow \Gamma'_{ul} \text{ and the operand judgement } \Gamma'_{ul} \vdash \text{opn} : t_{ul}, \text{ then } [t] = t_{ul} \]

- I.e., “compilation preserved types”
Global initializers

One would expect the following coherence property:

If $\Gamma \vdash e : t$ translates to the codestream judgement $\Gamma_\mathll < s \Rightarrow \Gamma_\mathll$ and the operand judgement $\Gamma_\mathll < opn : t_\mathll$, then $[t] = t_\mathll$

- I.e., “compilation preserved types”
- There is some subtlety in making this work!
  - Global declaration of string / array constants must compile to types with known length
    - E.g., `var x = int[]{0,1}` translates to `@x = global { 2, {0, 1} }`
    - Oat: `x` has type `int[]`
    - LLVM: `@x` has type `{ i64, [2xi64] } (≠ [int[]] = { i64, [0xi64] })`
  - `cmp_exp_as` is a variant of `cmp_exp` that ensures type preservation via bitcast.
Oat v2 (HW5)

- Specified by a (fairly large) type system
  - Invest some time in making sure you understand how to read the judgements and inference rules

- 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
    - \textit{ref} <: \textit{ref}?: non-null references are a subtype of nullable references
    - Record subtyping: width but not depth (\textit{why?})
Subtyping and type inference

\[
\text{SUBSUMPTION} \quad 
\frac{\Gamma \vdash e : s \quad \vdash s <: t} {\Gamma \vdash e : t}
\]

- **Challenge:**
  - In the presence of the subsumption rule, a term may have more than one type (how can we infer types for a declaration like \texttt{var x = exp}?)
  - Subsumption destroys syntax-directed quality of the type system

- **Solution:**
  - Do not use subsumption! Integrate subtyping into other inference rules. E.g.,

\[
\text{TYP\_CARR} \quad 
\frac{H \vdash t \quad H; G; L \vdash e_1 : t \quad \ldots \quad H; G; L \vdash e_n : t} {H; G; L \vdash \text{new } t[]\{e_1, \ldots e_n\}}
\]
Subtyping and type inference

### Subsumption

\[ \Gamma \vdash e : s \quad \vdash s <: t \]
\[ \Gamma \vdash e : t \]

- Challenge:
  - In the presence of the subsumption rule, a term may have more than one type \( \text{(how can we infer types for a declaration like } \text{var } x = \text{exp}?) \)
  - Subsumption destroys syntax-directed quality of the type system

- Solution:
  - Do not use subsumption! Integrate subtyping into other inference rules. E.g.,

\[ \text{TYP}_{-}\text{CARR} \]
\[ H \vdash t \quad H; G; L \vdash e_1 : t_1 \quad ... \quad H; G; L \vdash e_n : t_n \quad H \vdash t_1 <: t \quad ... \quad H \vdash t_n <: t \]
\[ H; G; L \vdash \text{new } t[\{ e_1, ... e_n \}] \]