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

Oat v1 (HW4) – well-formed types

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"

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

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

$$\textsc{RTVoid}\quad \frac{}{\vdash_{rt} \texttt{void}} \qquad \textsc{RTTyp}\quad \frac{\vdash t}{\vdash_{rt} t}$$

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

$$\textsc{LLBool}\ \frac{}{T \vdash_s \texttt{i1}} \qquad \textsc{LLInt}\ \frac{}{T \vdash_s \texttt{i64}} \qquad \textsc{LLPtr}\ \frac{T \vdash_r \textit{\textbf{ref}}}{T \vdash_s \textit{\textbf{ref}}*}$$

$$\textsc{LLTuple}\ \frac{T \vdash t_1 \quad \ldots \quad T \vdash t_n}{T \vdash \{t_1, \ldots, t_n\}} \qquad \textsc{LLArray}\ \frac{T \vdash t}{T \vdash \texttt{[}n \texttt{ x } t\texttt{]}}\ n \in \mathbb{N} \qquad \textsc{LLSimple}\ \frac{\vdash_s t}{\vdash t}$$

$$\textsc{LLRTVoid}\ \frac{}{T \vdash_{rt} \texttt{void}} \qquad \textsc{LLRTSimple}\ \frac{T \vdash_s t}{T \vdash_{rt} t} \qquad \textsc{LLRChar}\ \frac{}{T \vdash_r \texttt{i8}} \qquad \textsc{LLRType}\ \frac{T \vdash t}{T \vdash_r t}$$

$$\textsc{LLRFun}\ \frac{T \vdash_{rt} \textbf{\textit{rt}} \quad T \vdash_s t_1 \quad \ldots \quad T \vdash_s t_n}{T \vdash_r \textbf{\textit{rt}}(t_1, \ldots, t_n)}$$

$$\textsc{LLNamed}\ \frac{}{T \vdash \%uid}\ \%uid \in T$$

Translating well-formed types

- Each well-formed Oat type is translated to a well-formed LLVM type
  - types $\rightarrow$ simple types
  - reference types $\rightarrow$ reference types
  - return types $\rightarrow$ return types
- Use $[\![\cdot]\!]$ to denote translation
  - I.e., $[\![\vdash \texttt{int}]\!] = \vdash_s \texttt{i64}$ 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:

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

Each has a corresponding case:

- Case TINT: $[\![\vdash \texttt{int}]\!] = \vdash_s \texttt{i64}$ (well-formed by `LLInt`)
- Case TBOOL: $[\![\vdash \texttt{bool}]\!] = \vdash_s \texttt{i1}$ (well-formed by `LLBool`)

Translating well-formed types

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

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

Each has a corresponding case:

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

$$\textsc{TRef}\ \frac{\vdash_r \textit{ref}}{\vdash \textit{ref}} \rightsquigarrow \textsc{LLPtr}\ \frac{\vdash_r t\ (= [\![\vdash_r \textit{ref}]\!])}{\vdash t*}$$

  - I.e., $[\![\vdash \textit{ref}]\!] = \vdash_s t*$, where $\vdash_r t = [\![\vdash_r \textit{ref}]\!]$

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.

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

Summary of type translation

- $[\![\vdash \texttt{int}]\!] = \vdash_s \texttt{i64}$
- $[\![\vdash \texttt{bool}]\!] = \vdash_s \texttt{i1}$
- $[\![\vdash \textit{ref}]\!] = \vdash_s t*$, 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 HW4)

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

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

Num

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

Add

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

Seq

$$\frac{T, \Gamma \vdash s_1 \Rightarrow \Gamma' \qquad T, \Gamma' \vdash s_2 \Rightarrow \Gamma''}{T, \Gamma \vdash s_1, s_2 \Rightarrow \Gamma''}$$

Base

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

...lots more

Well-typed expressions

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

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 $\Gamma'_{ll} \vdash \textit{opn} : t_{ll}$

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
- To compute $[\![\Gamma \vdash x : t]\!](\texttt{ctxt})$, first let $(id, t*) = \texttt{ctxt}(x)$, then:
  - Define $[\![\texttt{ctxt}]\!]$ to be the type environment associated with `ctxt`
  - Codestream: $[\![\texttt{ctxt}]\!] \vdash \%uid = \texttt{load}\ \ t*\ \ opn \Rightarrow [\![\texttt{ctxt}]\!]\{\%uid \mapsto t\}$
  - Operand: $[\![\texttt{ctxt}]\!]\{\%uid \mapsto t\} \vdash \%uid : t$

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

- Need a symbol table `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 type environment associated with `ctxt`
  - Codestream: $[\![\texttt{ctxt}]\!] \vdash \%uid = \texttt{load}\ \ t*\ \ opn \Rightarrow [\![\texttt{ctxt}]\!]\{\%uid \mapsto t\}$
  - Operand: $[\![\texttt{ctxt}]\!]\{\%uid \mapsto t\} \vdash \%uid : t$

How can 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 type environment associated with `ctxt`
  - Codestream: $[\![\texttt{ctxt}]\!] \vdash \%uid = \texttt{load}\ \ t*\ \ opn \Rightarrow [\![\texttt{ctxt}]\!]\{\%uid \mapsto t\}$
  - Operand: $[\![\texttt{ctxt}]\!]\{\%uid \mapsto t\} \vdash \%uid : t$

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

Global initializers

One would expect the following coherence property:

If $\Gamma \vdash e : t$ *translates to the codestream judgement* $\Gamma_{ll} \vdash s \Rightarrow \Gamma'_{ll}$ *and the operand judgement* $\Gamma'_{ll} \vdash opn : t_{ll}$*, then* $[\![t]\!] = t_{ll}$

- I.e., "compilation preserved types"

Global initializers

One would expect the following coherence property:

If $\Gamma \vdash e : t$ *translates to the codestream judgement* $\Gamma_{ll} \vdash s \Rightarrow \Gamma'_{ll}$ *and the operand judgement* $\Gamma'_{ll} \vdash opn : t_{ll}$*, then* $[\![t]\!] = t_{ll}$

- 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] }` ($\neq [\![$`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
    - *ref* <: *ref?*: non-null references are a subtype of nullable references
    - Record subtyping: width but not depth (why?)

Subtyping and type inference

$$\textsc{Subsumption}\quad \frac{\Gamma \vdash e : s \qquad \vdash s <: t}{\Gamma \vdash e : t}$$

- 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 `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.,

$$\textsc{Typ\_CArr}\quad \frac{H \vdash t \qquad H; G; L \vdash e_1 : t \qquad \ldots \qquad H; G; L \vdash e_n : t}{H; G; L \vdash \texttt{new t[]}\{e_1, \ldots e_n\}}$$

Subtyping and type inference

Subsumption

$$\frac{\Gamma \vdash e : s \qquad \vdash s <: t}{\Gamma \vdash e : t}$$

- 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 `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.,

Typ_CArr

$$\frac{H \vdash t \qquad H; G; L \vdash e_1 : t_1 \qquad \ldots \qquad H; G; L \vdash e_n : t_n \qquad H \vdash t_1 <: t \qquad \ldots \qquad H \vdash t_n <: t}{H; G; L \vdash \texttt{new t[]}\{e_1, \ldots e_n\}}$$