CMSC 22600 **Compilers for Computer Languages** **Project 3**
Autumn 2016 **November 6, 2016**

The SOOL Normalizer

Due: November 20, 2016

1 Introduction

The third project is to transform the typed AST produced by the type checker into a lower-level representation that is more suitable for code generation and optimization. Specifically, you will transform the AST representation into *SOIR* (*Simple Object-oriented Intermediate Representation*), which has the following properties:

- arguments to operators and function calls are always variables
- object representations are explicitly represented
- method selection for both objects with class type and interface types is explicit
- class and interface meta-data is explicitly represented
- type coercions are explicitly represented
- `self` is explicitly passed to member functions
- Optional dispatch and variable access expressions are replaced with conditional tests.

2 SOIR

SOIR is a *normalized* representation, which means that we have restricted its syntactic structure to a canonical form. Specifically, SOIR expressions are all simple expressions involving a single operation that takes values (variables or constants) as arguments. We use such a representation for several reasons: it gives every subexpression a name (*i.e.*, the variable the expression is bound to), it makes data dependencies explicit, and it simplifies program analyses and optimizations. We give a "syntax" for SOIR in Figure 1

A SOOL program is represented in SOIR by a collection of class metadata, interface index tables, and function definitions. Figure 1 gives a "syntax" for SOIR function definitions that we use throughout this document. Note that SOIR has several different syntactic forms for extracting a field of a data object:

- $v\,.\,x$ for selecting a member-variable x from the object denoted by v.

Functions

fd ::= **fun** f **(**$x_1, \ldots, x_n$**)** b

Blocks

b ::= $\{\, s_1 \cdots s_n \,\}$

Statements

s ::=	**var** x**;**	Local-variable declaration
|	**var** x **=** e**;**	Local-variable initialization
|	x **:=** e**;**	Local-variable assignment
|	v**.**x **:=** e**;**	Member-variable assignment
|	v**(**$v_1, \ldots, v_n$**);**	Function-call statement
|	**if** v **then** b	If-then statement
|	**if** v **then** b_1 **else** b_2	If-then-else statement
|	**loop** b	Loop statement
|	**exit_if** v**;**	Loop exit statement
|	**return ;**	Void-return statement
|	**return** v**;**	Value-return statement

Expressions

e ::=	v	Value
|	v**.**x	Member-variable selection
|	v_1**::**v_2	Metadata/index-table selection
|	v_1**[**v_2**]**	Metadata indexing
|	**new** C	Object allocation
|	p**(**$v_1, \ldots, v_n$**)**	Primitive-operation application
|	v**(**$v_1, \ldots, v_n$**)**	Function application
|	$\langle\, v_1, \ldots, v_n \,\rangle$	Tuple allocation
|	**#i(**v**)**	Tuple-element projection

Values

v ::=	x	variables
|	*lit*	literals
|	**nil**	nil
|	$C.f$ | f	Functions
|	C	Metadata for class C
|	C@I	Index table for class C viewed as interface I

Figure 1: SOIR syntax

- v`::`f for selecting the member function f from a class or selecting the member-function offset for f from an interface's index table.
- v`::`I for selecting the offset for interface I's index table from an interface's index table.
- v_1`[`v_2`]` for selecting the function or index table from the class metadata specified by v_1 at the offset specified by v_2.
- `#i(`v`)` for selecting the ith component of the tuple v.

3 Runtime conventions

One of the main differences between the AST and SOIR representations is that we make various runtime conventions explicit in SOIR.

3.1 The main function

As discussed in the *Project Overview*, execution of a SOOL program occurs as if the statement "`main().run();`" is executed. As part of the translation to SOIR, we synthesize a function named `_sool_main` with no arguments whose body consists of creating a `main` object and then calling its `run` function.

3.2 Object creation

Creation of objects in SOIR is split into two steps. First we allocate space on the heap for the object using "`new` C" and then we call an initialization function $\texttt{init}_C$ for the object passing in the newly allocated object and the initialization arguments. If the class C has a superclass B, then its initialization function calls the initialization function for B (and so on) before initializing C's member variables.

3.3 Member functions

Member functions in the AST have an implicit `self` parameter; when we translate to SOIR, we make this parameter explicit in the representation. This parameter is bound to a pointer to the heap-allocated object and has the class type of the class where the member function was declared.

3.4 Runtime values

We use a *uniform representation* convention for SOOL values. By this term, we mean that every SOOL value is represented by a single machine word, which is either a tagged integer or a pointer into the heap. Booleans are represented as as tagged integers.[1] This representation allows us to distinguish the `nil` value, which is represented by the machine value zero, from all other SOOL values.

[1] The details of the tagging scheme for integers will be addressed in the next project.

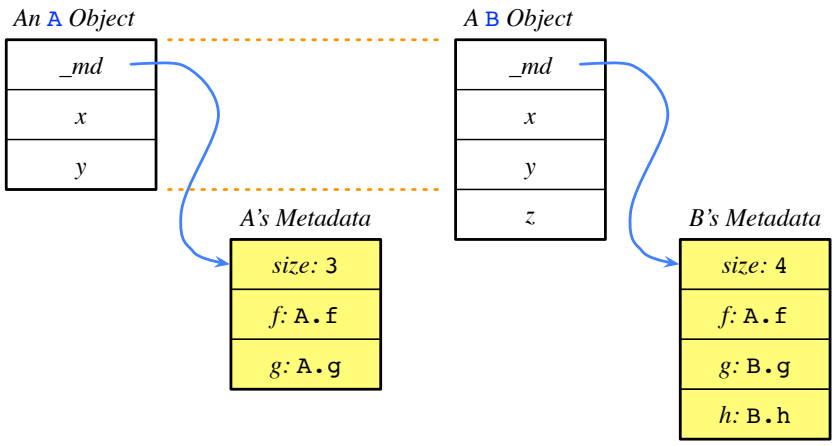
3.5 Object representation

We use two different runtime representations for objects, depending on whether they have a class or interface type.

An object of class type C is represented by a heap-allocated sequence of word-sized values, where the first value is a pointer to the C's metadata and the remaining values are the member variables of the object. We use "_md" to label the metadata field of the object's representation. The order of member variables must respect the prefix ordering of the class heirarchy. For example, consider the following two classes:

```
class A() {
  var x : int = 1
  var y : int = 2
  meth f : () -> int { return self.x; }
  meth g : () -> int { return self.y; }
}
class B() extends A() {
  var z : int = 3
  override meth g : () -> int { return self.z; }
  meth h : () -> int { return self.y + self.z; }
}
```

The object and metadata layout for these classes is as follows:[2]



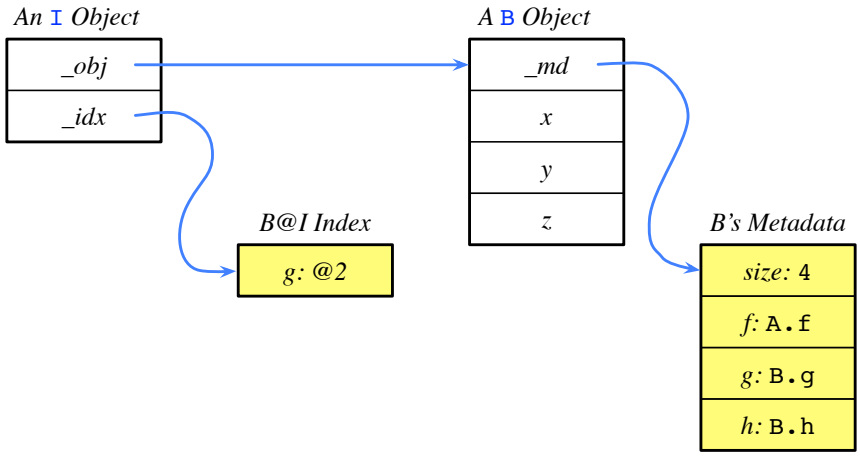
Notice that the layout of the A object is a prefix of the layout of the B object. This property means that the A.f function is able to access the x and y member variables of a class B object. Class metadata is allocated statically and is shared among all objects created from the class.

Objects with an interface type have a more complicated representation, since a given interface may be implemented by multiple classes. To represent an object of class type C at an interface type I, we pair the pointer to the object's representation with a pointer to an *index table*, which is a table of offsets used to lookup member functions in C's metadata table. For example, consider the previous example extended with the interface

[2]Note, we are assuming no interfaces in the program, so the metadata for this example does not have to account for index tables.

```
interface I {
  meth g : () -> int
}
```

which is a supertype of both class A and B. Consider a class B object viewed through the I interface; for this interface, we need an index table that tells us how to access the g member function from the object. The wrapped object will have the following runtime representation:



where $B@I$ is the name of the table that lets us view a B object at interface I.[3] Again, this is a simple representation that does not account for possible interface-to-interface coercions. We discuss the full complexity of class and interface metadata in the next section.

3.6 Object-type coercions and meta-data

Because of the prefix ordering on object representations, there is no runtime cost for coercing from a class to one of its superclasses. When coercing from a class type to an interface type, we can statically determine what index table must be used access the class's member functions and generate code to create the wrapped object representation described above. Things are more complicated, however, when coercing between two interface types. For example, we might have an object of class A, where A implements interface I, and we might be using it in a context where it is viewed as an I object. If in this context, we coerce it to a J-typed object, we do not know that we should be using $A@J$ to access member functions in the object. Therefore, we have to store pointers to the interface index tables in the class metadata and, furthermore, we must store offsets for these pointers in the interface index tables.

Consider a three level class heiarchy: $C \lessdot B \lessdot A$. The meta data for class C will consist an object-size field, followed by three sections: class A's information, class B's information, and class C's information. For each class, we have two subsections: the member-function pointers for the members declared in that class and the interface-index tables for class. Because we layout the metadata for a class from superclass to subclass, the metadata for B will have a layout that is a prefix for the metadata of class C (although the function-pointers might be different because of function overriding).

[3] Note that if we also view class A objects at interface I, then we can share this table between B and its superclass (and it would be called $A@I$).

For each class C in the program, we must define the *interfaces* of a class C (written $\mathbb{I}_C$). We start by defining the set of object-type coercions in the AST representation of the program as follows:

$$\begin{aligned}\mathbb{C} &= \{(I \succeq C) \mid \text{for each coercion } (I \succeq C)e \text{ in the program}\} \\ &\cup \{(J \succeq I) \mid \text{for each coercion } (J \succeq I)e \text{ in the program}\} \\ &\cup \{(I \succeq C) \mid \text{for each coercion } (I? \succeq C?)e \text{ in the program}\} \\ &\cup \{(J \succeq I) \mid \text{for each coercion } (J? \succeq I?)e \text{ in the program}\}\end{aligned}$$

Then, we define the interfaces of C to be the smallest set satisfying the following equation:

$$\begin{aligned}\mathbb{I}_C &= \{I \mid (I \succeq C) \in \mathbb{C}\} \\ &\cup \{I \mid I \in \mathbb{I}_B \wedge C \lessdot B\} \\ &\cup \{J \mid (J \succeq I) \in \mathbb{C} \wedge I \in \mathbb{I}_C\}\end{aligned}$$

This definition consists of three parts: the first accounts for direct coercions of the class C to interfaces, the second accounts for coercions from superclasses of C to interfaces (since C can be coerced to its superclasses), and the last closes the set over interface to interface coercions.

We must also define the interfaces of an interface I (written $\mathbb{I}_I$), which are necessary to implement interface-to-interface coercions. As before, we define this set as the smallest set satisfying the following equation:

$$\begin{aligned}\mathbb{I}_I &= \{J \mid (J \succeq I) \in \mathbb{C}\} \\ &\cup \{K \mid (K \succeq J) \in \mathbb{C} \wedge J \in \mathbb{I}_I\}\end{aligned}$$

These two definitions satisfy the following important properties:

- If $C \lessdot B$ then $\mathbb{I}_B \subseteq \mathbb{I}_C$ (the *prefix* property).
- If there is a coercion $(J \succeq I)e$ in the program, then $\mathbb{I}_J \subseteq \mathbb{I}_I$.
- If $I \in \mathbb{I}_C$, then $\mathbb{I}_I \subseteq \mathbb{I}_C$.

From these definitions, we can construct the metadata for each class, as well as the necessary index tables to implement interface-to-interface coercions. Before defining this construction, we need to define a mapping from member-function names in a class C to the SOIR functions that implement them.

$$\mathcal{F}_C : \textsc{MembFun} \xrightarrow{\text{fin}} \textsc{FunId}$$

This mapping must account for inheritance and for overriding of inherited member functions. For a base class defined as **class** $C(\cdots)$ { vd; fd } we have

$$\mathcal{F}_C = \{f \mapsto C.f \mid f \in fd\}$$

and for a derived class **class** $C(\cdots)$ **extends** $B(\cdots)$ { vd; fd } we have

$$\mathcal{F}_C = \mathcal{F}_B \pm \{f \mapsto C.f \mid f \in fd\}$$

The structure of class metadata has the following structure:

$$\textsc{Metadata} = \mathbb{N} \times (\textsc{Fun}^* \times \textsc{Ifc}^*)^*$$

The layout of class metadata is specified by the function

$$\mathcal{L}_{\mathcal{C}}[\![\cdot]\!] : \textsc{Cls} \xrightarrow{\text{fin}} (\textsc{Fun}^* \times \textsc{Ifc}^*)^*$$

which defines the layout of a prefix of class C's metadata. This function is defined inductively, as follows: For a base class defined as $\textbf{class}\ C(\cdots)\ \{\ vd;\ fd\ \}$ the metadata is defined by

$$\begin{aligned}
\mathcal{L}_{\mathcal{C}}[\![\, B \,]\!] \quad &= \quad \langle\, \mathcal{F}_C(f_1),\ \ldots,\ \mathcal{F}_C(f_n),\ B@I_1,\ \ldots,\ B@I_n \,\rangle \\
&\qquad \text{where } B \text{ is a base class } \textbf{class}\ B(\cdots)\ \{\ vd;\ fd\ \} \\
&\qquad f_i \in fd, \text{ and } I_i \in \mathbb{I}_B \\
\mathcal{L}_{\mathcal{C}}[\![\, B \,]\!] \quad &= \quad \mathcal{L}_{\mathcal{C}}[\![\, A \,]\!] \oplus \langle\, \mathcal{F}_C(f_1),\ \ldots,\ \mathcal{F}_C(f_n),\ B@I_1,\ \ldots,\ B@I_n \,\rangle \\
&\qquad \text{where } B \text{ is a derived class } \textbf{class}\ B(\cdots)\ \textbf{extends}\ A(\cdots)\ \{\ vd;\ fd\ \}, \\
&\qquad f_i \in fd, \text{ and } I_i \in \mathbb{I}_B \setminus \mathbb{I}_A
\end{aligned}$$

where $\oplus$ is tuple concatenation. We then define the metadata of a class C to be

$$\mathcal{M}[\![\, C \,]\!] = \langle\, |\hat{\mathcal{V}}| + 1 \,\rangle \oplus \mathcal{L}_{\mathcal{C}}[\![\, C \,]\!]$$

where C has the signature $(\cdots)\{\!|\, B; \hat{\mathcal{V}}; \hat{\mathcal{F}} \,|\!\}$.

We also need to define the contents of the interface index tables (the $C@I$ values) that appear in the class metadata. We model an index table as a finite map from function and index-table names to offsets. The actual layout of an index table does not matter, as long as all of the tables for a given interface have the same layout. Let $\mathcal{O}_{\mathcal{C}}$ be a function that maps functions and index-table names to their offset in class C's metadata and let interface I have the signature $\{\!|\, \hat{\mathcal{F}} \,|\!\}$, then the index table for class C viewed with type I is given by

$$\mathcal{X}[\![\, C@I \,]\!] = \{f_i \mapsto \mathcal{O}_{\mathcal{C}}(f_i) \mid f_i \in \mathrm{dom}(\hat{\mathcal{F}})\} \cup \{J_i \mapsto \mathcal{O}_{\mathcal{C}}(C@J_i) \mid J_i \in \mathrm{dom}(\mathbb{I}_I)\}$$

4 Normalization

The other main difference between the AST and SOIR representations is that SOIR is a normalized representation. In this section, we give a description of the *normalization* process that translates the AST to SOIR.

4.1 Classes, functions, and statements

For each class C in the program, we generate a SOIR function `C_init` to initialize the object and for each function f defined in C, we generate a SOIR function `C.f`. For void-valued functions, we introduce a **`return`** statement if there was not one in the AST. For example, the function `f` in the following SOOL code:

```
class C () { meth f () -> void { } }
```

is translated to the following SOIR function:

```
fun C.f (self) {
  return;
}
```

For statements, the translation from AST to SOIR is mostly a direct conversion of AST statement forms to SOIR statements. The main difference is that an expression that occurs in an AST statement is normalized to either a simple expression (if it is the right-hand-side of an assignment or local-variable definition) or to a value. The expression normalization process (described in the next section), generates additional statements that define intermediate results for subexpressions.

The one statement form that has a slightly more complicated translation is the **`while`** loop. Because we want to avoid duplicating the code that computes the loop's condition, we need to break up the loop into two statement forms: a **`loop`** statement and a conditional exit statement (**`exit_if`**). For example, the following SOOL while loop:

```
while (i <= n-1) {
  p := i * p;
  i := i + 1;
}
```

is translated to the following SOIR code:

```
loop {
    var t = n-1;
    var t2 = (i <= t);
  exit_if t2;
    p := i * p;
    i := i + 1;
}
```

The code before the **`exit_if`** statement is called the *loop header*.

4.2 Expressions

The normalization of AST expressions is the most involved aspect of the normalization process, so we formalize its description in this section. This translation is defined in terms of the following translation functions:

$$
\begin{aligned}
\mathcal{E}[\![\cdot]\!] &: \text{ASTExp} \to (\text{Value} \to \text{Stm}^*) \to \text{Stm}^* \\
\mathcal{A}[\![\cdot]\!] &: \text{ASTExp}^k \to (\text{Value}^k \to \text{Stm}^*) \to \text{Stm}^* \\
\mathcal{Q}_{T,T'}[\![\cdot \sim \cdot]\!] &: \text{Value} \times \text{Value} \to (\text{Value} \to \text{Stm}^*) \to \text{Stm}^* \\
\mathcal{N}[\![\cdot]\!] &: \text{ASTExp} \to (\text{Value} \to \text{Stm}^*) \to (\text{Value} \to \text{Stm}^*) \to \text{Stm}^* \\
\mathcal{C}_{(T' \succeq T)}[\![\cdot]\!] &: \text{Value} \to (\text{Value} \to \text{Stm}^*) \to \text{Stm}^*
\end{aligned}
$$

where ASTExp is the set of AST expressions, Value is the set of SOIR value terms, and Stm is the set of SOIR statement terms. The function $\mathcal{E}$ is used to normalize expressions, $\mathcal{A}$ normalizes tuples of expression (*e.g.*, the arguments of a function call), $\mathcal{Q}_{T,T'}$ is a helper for translating equality operators, $\mathcal{N}$ is a helper for handling the "**`?`**" operation, and $\mathcal{C}_{(T' \succeq T)}$ is used to normalize coercion expressions.

The $\mathcal{E}$ and $\mathcal{E}$ functions are given in Figure 2, while the other operations are defined in Figure 3. The definition of $\mathcal{Q}_{T,T'}$ is incomplete and is left to you to finish. Note that when coercing a boolean or integer value to its corresponding interface we create a new object using the internal class for the type. The **`string`** type is already represented by an object, so we do not have to allocate a wrapper for it.

4.3 Primops

Part of the normalization translation is to replace the AST operators with SOIR primitive operators (*primops*). The main difference between the `BinOp.t` datatype and the `PrimOp.t` datatype is that the latter supports additional operations that are either generated during translation to SOIR or may arise from optimizations. For example, SOOL has "`<`" and "`<=`" as builtin operators, whereas in SOIR, we also have "`>`" and "`>=`." The SOIR primops also include type-specialized versions of equality.

4.4 Options

As part of normalization, we make tests for **`nil`** explicit. For that purpose, we have a special primop `isNIL` for testing equality to **`nil`**. For example, consider the expression "$e?x$" and assume that normalization translates e to code that sets the variable t to the result of e. Then, normalization should produce the following SOIR code for the above expression

```
var t = ...
var t';
if isNIL(t)
  then t' := nil
  else t' := t.x
```

5 An example

In this section, we given an example how metadata is represented in a more complicated situation. Consider the class and interface declarations shown in Figure 4 (along the type hierarchy). The inherits relations are shown in green, the implements relations are in orange, and the interface-to-interface subtyping relations are in blue.

Let us also assume that the program contains the object-type coercions shown in Figure 5. On the right is the type-hierarchy graph restricted to the coercions in the program. The class-to-class coercions are in green, the class-to-interface coercions are in orange, and the interface-to-interface coercions are in blue (the dotted arrows represent subtyping relations that are not manifest in $\mathbb{C}$). From this diagram, we see that class `A` must support the `K` interface; class `B` must support the `I` interface and, because of transitivity, `B` must also support the `J` and `K` interfaces; and class `C` must

$$
\begin{array}{rcl}
\mathcal{E}[\![\,(e_1 \texttt{ \&\& } e_2)\,]\!]\,\kappa & = & \mathcal{E}[\![\,e_1\,]\!]\,\lambda(t_1).\,\texttt{var } t\texttt{;} \\
& & \quad \texttt{if } t_1 \texttt{ then } \{\,\mathcal{E}[\![\,e_2\,]\!]\,\lambda(t_2).\,t \texttt{ := } t_2\texttt{;}\,\} \\
& & \quad \texttt{else } \{\,t \texttt{ := false;}\,\} \\
& & \quad \kappa(t) \\
\mathcal{E}[\![\,(e_1 \texttt{ || } e_2)\,]\!]\,\kappa & = & \mathcal{E}[\![\,e_1\,]\!]\,\lambda(t_1).\,\texttt{var } t\texttt{;} \\
& & \quad \texttt{if } t_1 \texttt{ then } \{\,t \texttt{ := true;}\,\} \\
& & \quad \texttt{else } \{\,\mathcal{E}[\![\,e_2\,]\!]\,\lambda(t_2).\,t \texttt{ := } t_2\texttt{;}\,\} \\
& & \quad \kappa(t) \\
\mathcal{E}[\![\,(e_1 \texttt{ == } e_2)\,]\!]\,\kappa & = & \mathcal{E}[\![\,e_1\,]\!]\,\lambda(t_1).\,\mathcal{E}[\![\,e_2\,]\!]\,\lambda(t_2).\,\mathcal{Q}_{\text{typeOf}(e_1),\text{typeOf}(e_2)}[\![\,t_1 \texttt{==} t_2\,]\!]\,\kappa \\
\mathcal{E}[\![\,(e_1 \texttt{ != } e_2)\,]\!]\,\kappa & = & \mathcal{E}[\![\,e_1\,]\!]\,\lambda(t_1).\,\mathcal{E}[\![\,e_2\,]\!]\,\lambda(t_2).\,\mathcal{Q}_{\text{typeOf}(e_1),\text{typeOf}(e_2)}[\![\,t_1 \texttt{!=} t_2\,]\!]\,\kappa \\
\mathcal{E}[\![\,(e_1 \odot e_2)\,]\!]\,\kappa & = & \mathcal{E}[\![\,e_1\,]\!]\,\lambda(t_1).\,\mathcal{E}[\![\,e_2\,]\!]\,\lambda(t_2).\,\texttt{var } t \texttt{ = } \odot\ \texttt{(}t_1\texttt{, } t_2\texttt{);}\ \kappa(t) \\
\mathcal{E}[\![\,C(e_1, \ldots, e_n)\,]\!]\,\kappa & = & \mathcal{A}[\![\,e_1, \ldots, e_n\,]\!]\,\lambda(t_1, \ldots, t_n).\,\texttt{var } t \texttt{ = new } C\texttt{; init}_C\texttt{(}t\texttt{, } t_1\texttt{, } \ldots\texttt{, } t_n\texttt{);}\ \kappa(t) \\
\mathcal{E}[\![\,e.f(e_1, \ldots, e_n)\,]\!]\,\kappa & = & \begin{cases}
\mathcal{A}[\![\,e, e_1, \ldots, e_n\,]\!]\,\lambda(obj, t_1, \ldots, t_n). & \text{if typeOf}(e) = C \\
\quad \texttt{var } md \texttt{ = } obj\texttt{._md;} \\
\quad \texttt{var } f \texttt{ = } md\texttt{::}f\texttt{;} \\
\quad \texttt{var } t \texttt{ = } f\texttt{(}obj\texttt{, } t_1\texttt{, } \ldots\texttt{, } t_n\texttt{);} \\
\quad \kappa(t) \\
\mathcal{A}[\![\,e, e_1, \ldots, e_n\,]\!]\,\lambda(iobj, t_1, \ldots, t_n). & \text{if typeOf}(e) = I \\
\quad \texttt{var } obj \texttt{ = \#1(}iobj\texttt{);} \\
\quad \texttt{var } index \texttt{ = \#2(}iobj\texttt{);} \\
\quad \texttt{var } ix \texttt{ = } index\texttt{::}f\texttt{;} \\
\quad \texttt{var } md \texttt{ = } obj\texttt{._md;} \\
\quad \texttt{var } f \texttt{ = } md\texttt{[}ix\texttt{];} \\
\quad \texttt{var } t \texttt{ = } f\texttt{(}obj\texttt{, } t_1\texttt{, } \ldots\texttt{, } t_n\texttt{);} \\
\quad \kappa(t)
\end{cases} \\
\mathcal{E}[\![\,e?f(e_1, \ldots, e_n)\,]\!]\,\kappa & = & \texttt{var } t\texttt{;} \\
& & \mathcal{N}[\![\,e\,]\!]\,(\lambda().\,t \texttt{ := nil;})(\lambda(t_1).\,\mathcal{E}[\![\,t_1.f(e_1, \ldots, e_n)\,]\!]\,\lambda(t_3).\,t \texttt{ := } t_3\texttt{;}) \\
& & \kappa(t) \\
\mathcal{E}[\![\,e.x\,]\!]\,\kappa & = & \mathcal{E}[\![\,e\,]\!]\,\lambda(t_1).\,\texttt{var } t \texttt{ = } t\texttt{.}x\texttt{;}\ \kappa(t) \\
\mathcal{E}[\![\,e?x\,]\!]\,\kappa & = & \texttt{var } t\texttt{;} \\
& & \mathcal{N}[\![\,e\,]\!]\,(\lambda().\,t \texttt{ := nil;})(\lambda(t_1).\,t \texttt{ := } t_1\texttt{.}x\texttt{;}) \\
& & \kappa(t) \\
\mathcal{E}[\![\,e!\,]\!]\,\kappa & = & \mathcal{N}[\![\,e\,]\!]\,(\lambda().\,\texttt{FAIL})(\kappa(t_1)) \\
\mathcal{E}[\![\,x\,]\!]\,\kappa & = & \kappa(x) \\
\mathcal{E}[\![\,lit\,]\!]\,\kappa & = & \kappa(lit) \\
\mathcal{E}[\![\,\textbf{nil } T\,]\!]\,\kappa & = & \kappa(\texttt{nil}) \\
\mathcal{E}[\![\,(T' \succeq T)e\,]\!]\,\kappa & = & \mathcal{E}[\![\,e\,]\!]\,\lambda(t_1).\,\mathcal{C}_{(T' \succeq T)}[\![\,t\,]\!]\,\lambda(t_2).\,\kappa(t_2) \\
\\
\mathcal{A}[\![\,e_1, \ldots, e_n\,]\!]\,\kappa & = & \mathcal{E}[\![\,e_1\,]\!]\,\lambda(t_1).\ \cdots\ \mathcal{E}[\![\,e_n\,]\!]\,\lambda(t_n).\,\kappa(t_1, \ldots, t_n)
\end{array}
$$

Figure 2: Normalization of AST expressions

$$
\begin{array}{rcl}
\mathcal{Q}_{\iota,\iota}[\![\, v_1 \sim v_2 \,]\!] & = & \texttt{var}\ t = \sim_\iota (v_1, v_2);\ \kappa(t) \\
\mathcal{Q}_{\iota?,\iota?}[\![\, v_1 \sim v_2 \,]\!] & = & \texttt{var}\ t = \sim_\iota (v_1, v_2);\ \kappa(t) \\
\mathcal{Q}_{C,C}[\![\, v_1 \sim v_2 \,]\!] & = & \texttt{var}\ t = \sim_{obj} (v_1, v_2);\ \kappa(t) \\
\mathcal{Q}_{C?,C?}[\![\, v_1 \sim v_2 \,]\!] & = & \texttt{var}\ t = \sim_{obj} (v_1, v_2);\ \kappa(t) \\
\mathcal{Q}_{I,C}[\![\, v_1 \sim v_2 \,]\!] & = & \texttt{var}\ obj = \texttt{\#1}(v_1); \\
 & & \texttt{var}\ t = \sim_{obj} (obj, v_2); \\
 & & \kappa(t) \\
\mathcal{Q}_{I?,C}[\![\, v_1 \sim v_2 \,]\!] & = & \texttt{var}\ t; \\
 & & \mathcal{N}[\![\, v_1 \,]\!] \\
 & & \quad (\lambda().\, t := \texttt{false};) \\
 & & \quad (\texttt{var}\ obj = \texttt{\#1}(v_1);\ t := \sim_{obj} (obj, v_2);) \\
 & & \kappa(t) \\
\mathcal{Q}_{I,J}[\![\, v_1 \sim v_2 \,]\!] & = & \texttt{var}\ obj_1 = \texttt{\#1}(v_1); \\
 & & \texttt{var}\ obj_2 = \texttt{\#1}(v_2); \\
 & & \texttt{var}\ t = \sim_{obj} (obj_1, obj_2); \\
 & & \kappa(t) \\
\cdots & & \\
\\
\mathcal{N}[\![\, e \,]\!]\, \kappa_1 \kappa_2 & = & \mathcal{E}[\![\, e \,]\!]\, \lambda(t_1).\, \texttt{var}\ t_2 = \texttt{isNil}(t_1); \\
 & & \quad \texttt{if}\ t_2\ \texttt{then}\ \kappa_1()\ \texttt{else}\ \kappa_2(t_1) \\
\mathcal{C}_{(T?\succeq T)}[\![\, v \,]\!]\, \kappa & = & \kappa(v) \\
\mathcal{C}_{(C'\succeq C)}[\![\, v \,]\!]\, \kappa & = & \kappa(v) \\
\mathcal{C}_{(C'?\succeq C?)}[\![\, v \,]\!]\, \kappa & = & \kappa(v) \\
\mathcal{C}_{(T_1?\succeq T_2?)}[\![\, v \,]\!]\, \kappa & = & \texttt{var}\ t; \\
 & & \mathcal{N}[\![\, v \,]\!]\, (\lambda().\, t := \texttt{nil};)(\lambda(\_).\, \mathcal{C}_{(T_1\succeq T_2)}[\![\, v \,]\!]\, \lambda(t_1).\, t := t_1;) \\
 & & \kappa(t) \\
\mathcal{C}_{(I\succeq C)}[\![\, v \,]\!]\, \kappa & = & \texttt{var}\ t = \langle\, v, C@I \,\rangle;\ \kappa(t) \\
\mathcal{C}_{(I\succeq \iota)}[\![\, v \,]\!]\, \kappa & = & \texttt{var}\ t_1 = \texttt{new}\ \iota; \\
 & & \texttt{init}_\iota(t_1, v); \\
 & & \texttt{var}\ t = \langle\, t_1, \iota@I \,\rangle; \\
 & & \kappa(t) \\
\mathcal{C}_{(J\succeq I)}[\![\, v \,]\!]\, \kappa & = & \texttt{var}\ t_1 = \texttt{\#1}(v); \\
 & & \texttt{var}\ t_2 = \texttt{\#2}(v); \\
 & & \texttt{var}\ t_3 = t_2 :: J; \\
 & & \texttt{var}\ t_4 = t_1.\texttt{\_md}; \\
 & & \texttt{var}\ t_5 = t_4[t_3]; \\
 & & \texttt{var}\ t = \langle\, t_1, t_5 \,\rangle; \\
 & & \kappa(t)
\end{array}
$$

Figure 3: Additional normalization functions

```
class A() {
  meth f : () -> void { ··· }
}
class B() extends A() {
  override meth f : () -> void { ··· }
  meth g : () -> void { ··· }
}
class C() {
  meth h : () -> void { ··· }
  meth f : () -> void { ··· }
}
interface I {
  meth f : () -> void
  meth g : () -> void
}
interface J {
  meth g : () -> void
}
interface K {
  meth f : () -> void
}
```

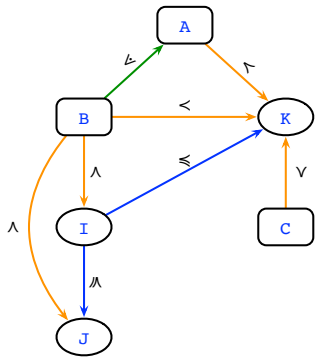


Figure 4: A simple example and its type heirarchy

$$\mathbb{C} = \left\{ \begin{array}{l} (\texttt{A} \succeq \texttt{B}) \\ (\texttt{K} \succeq \texttt{A}) \\ (\texttt{I} \succeq \texttt{B}) \\ (\texttt{K} \succeq \texttt{C}) \\ (\texttt{J} \succeq \texttt{I}) \\ (\texttt{K} \succeq \texttt{I}) \end{array} \right\}$$

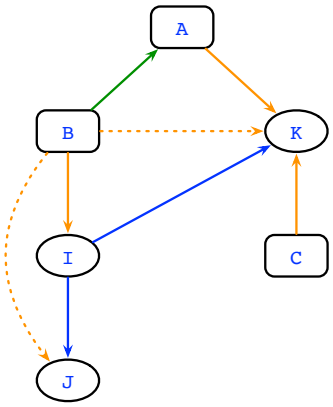


Figure 5: Coercions and the type hierarchy restricted to the program's coercions (the dotted edges represent subtyping relations that are not manifest in $\mathbb{C}$)

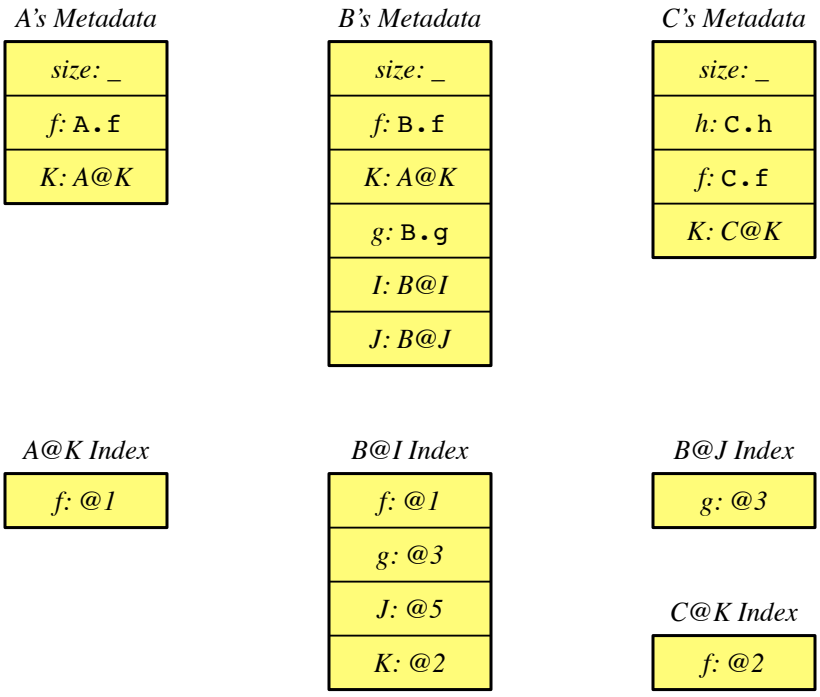


Figure 6: Metadata tables for example in Figure 4

support the K interface. Using the notation from the previous section, we have

$$\begin{aligned}
\mathbb{I}_A &= \{\texttt{K}\} \\
\mathbb{I}_B &= \{\texttt{I}, \texttt{K}, \texttt{J}\} \\
\mathbb{I}_C &= \{\texttt{K}\} \\
\mathbb{I}_I &= \{\texttt{K}, \texttt{J}\} \\
\mathbb{I}_J &= \{\} \\
\mathbb{I}_K &= \{\}
\end{aligned}$$

These requirements are captured in the metadata layout shown in Figure 6. In this layout, we use the notation "$A@K$" to name the index table for viewing objects from class A as having interface type K. Because of the prefix property of classes, this index table will also work for any subclass of A (*e.g.*, B). With this information, we can then translate the various coercions that were described above. This translation is given in Figure 7.

6 Implementation hints

There are two main parts to this project: an analysis of the AST, which determines the runtime representations of objects, and of class and interface metadata; and the normalization phase, which translates the AST to SOIR.

Coercion		SOIR code
`(A ≥ B)obj`	⇒	no code required
`(K ≥ A)obj`	⇒	`var iobj = ⟨obj, A@K⟩;`
`(I ≥ B)obj`	⇒	`var iobj = ⟨obj, B@I⟩;`
`(K ≥ C)obj`	⇒	`var iobj = ⟨obj, C@K⟩;`

`(J ≥ I)iobj`	⇒	`var obj = #1(iobj);`
		`var index = #2(iobj);`
		`var ix = index::J;`
		`var md = obj._md;`
		`var index′ = md[ix];`
		`var iobj′ = ⟨obj, index′⟩;`

`(K ≥ I)iobj`	⇒	`var obj = #1(iobj);`
		`var index = #2(iobj);`
		`var ix = index::K;`
		`var md = obj._md;`
		`var index′ = md[ix];`
		`var iobj′ = ⟨obj, index′⟩;`

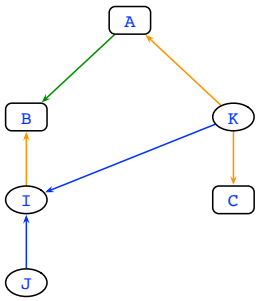
Figure 7: SOIR code for coercions using the metadata from Figure 6

6.1 AST analysis

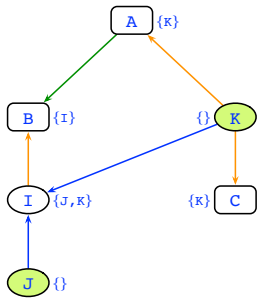
The main job of the AST analysis is to determine the representation of class and interface metadata. It also involves defining a mapping from class and interface members in the AST to their representation in SOIR.

The first step of the analysis is to walk over the AST representation of the program and to collect the object-type coercions (*i.e.*, the set $\mathbb{C}$). From these, the equations of Section 3.6 can be computed using in a brute-force fixed-point iteration, but it is also possible to compute them more efficiently using the coercion graph to guide the computation. The coercion graph is a graph with a node for each class and interface in the program, and an edge from the node for type T to T' if there is a coercion $(T \succeq T') \in \mathbb{C}$. We augment these edges with a edge from B to C when $C \lessdot B$ and there is no coercion from C to B in the program. Figure 8(a) shows the coercion graph for the example from Figure 4. Note that the direction of the edges in the coercion graph go from supertype to subtype! I.e., the **opposite** direction as the edges in the type-hierarchy graph. Recall the three properties of the interface sets given on Page 6; These imply that if there is an edge from T to T', then $\mathbb{I}_T \subseteq \mathbb{I}_{T'}$. Thus, by propagating the interface sets from the roots to the leaves of the coercion graph we can compute the information for the metadata layout. Figures 8(b)-(d) illustrate this process on the example graph. Note that the only real propagation of information occurs at the last step, when the interfaces `J` and `K` are propagated to class `B`.

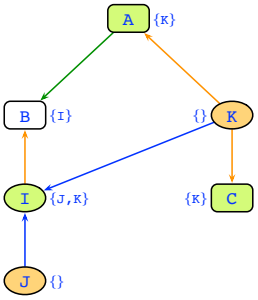
In your implementation, you will also have to introduce nodes for the Basis interfaces (*e.g.*, `boolI`). In addition, there are internal classes that implement the methods of the `boolI`, `intI`, and `stringI` interfaces. Coercions from primitive types to these interfaces should be treated as coercions from the corresponding internal classes to the interfaces.



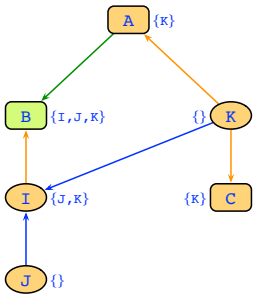
(a) Initial coercion graph



(b) Step 1 of analysis



(c) Step 2 of analysis



(d) Step 3 of analysis

Figure 8: Computing metadata for example in Figure 4. Each node is annotated with its interfaces computed so far; the green nodes are the frontier of the graph traversal and the orange nodes are the nodes that have already been computed.

Once the interface sets have been computed, the layout of class metadata and the creation of interface index tables can proceed by a pre-order traversal of the class hierarchy.

Another detail that must be addressed are *orphan interfaces*, which are interfaces for which there is no class that coerces to them (but there is code that uses them). For example, consider the following SOOL program:

```
interface I { meth f : () -> void }
interface J extends I { meth g : () -> void }
class main {
  meth h (x : J) -> I { return x; }
  meth run () -> void { }
}
```

In the body of `main.h`, we need to generate code for a coercion from `J` to `I`, but since there is no class that implements either `I` or `J`, we will not have generated a layout for the interfaces. In this case, it is safe to map all offsets for the interface to `0`. Thus, the SOIR code for `main.h` will be

```
fun main.h (self, x) {
  var obj = #1(x);
  var md = obj._md;
  var index = md[0];
  var iobj = ⟨obj, index⟩
```

```
  return;
}
```

6.2 Normalization

The most direct way to implement the normalization of expressions is using a higher-order, context-passing, approach. In this case, a context is a function from a SOIR variable to a list of SOIR statements (essentially, it is the κ in Figure 2). For example, consider the expression "$e_1 + e_2$." We are going to generate code that evaluates e_1 and binds the result to some variable (say t_1), then evaluates e_2 and binds the result to t_2, and then computes the sum and binds the result to the variable t. In the normalizer, this would be implemented as:

```
fun expToVar (env, exp, cxt) = (case exp
       of ...
        | AST.EXP_PrimOp(e1, BinOp.^+^, e2) =>
            expToVar (env, e1, fn v1 =>
              expToVar (env, e2, fn v2 => let
                val t = freshTmp()
                in
                  SOIR.LocalVarStm(t,
                    SOME(SOIR.PrimExp(PrimOp.IntAdd, [v1, v2])))
                  :: cxt (SOIR.VarVal t))
                end))
        | ...
      (* end case *))
```

Of course, in practice, we want to handle all of the non-conditional operators as a single case.

7 Submission

We will collect the projects at **11pm** on **Sunday November 20** from the SVN repositories, so make sure that you have committed your final version before then.

> **Important note:** You are expected to submit code that **compiles** and that is well documented. Remember that points for project code are assigned 30% for coding style (documentation, choice of variable names, and program structure), and 70% for correctness. Code that does not compile will **not** receive any points for correctness.

8 Document history

November 17, 2016 Removed rule for $\mathcal{Q}_{I,C?}$ from Figure 3, since that case cannot occur (neither argument is a subtype of the other) and replaced it with the rule for $\mathcal{Q}_{I?,C}$.

November 16, 2016 Fixed typo in rule for `&&` operator.

November 12, 2016 Changed the treatment of `string` to `stringI` coercions to agree with the other primitive types.

November 6, 2016 Original version.