MP 4 (Extra Credit) – Simple Object-Oriented Programming Language
CS 422 – Spring 2017
Revision 1.0

Assigned April 25, 2017
Due May 10, 2017, 11:59pm
Extension None

1 Change Log
1.0 Initial Release.

2 Turn-In Procedure
Put your code as plain text for this MP in a file named mp4.thy, and submit your plain text file mp4.thy using (svn commit -m "submitting mp4" mp4.thy). Your file should contain your name, and netid in a comment at the top, and it should contain your solution. It should be named mp4.thy and committed in your assignments/mp4 directory.

3 Objectives
The purpose of this exercise is to familiarize you with some of the semantic consequences for dynamic dispatch in an object oriented programming language.

4 Background
In this assignment you will be giving an evaluation semantics in a Natural Semantics style a simple object-oriented programming language without cast (or super or self) (SOOPL). This language is a slight simplification of the language discussed in class (OOPL). As background, we will review the syntax, and Natural Semantics for the language discussed in
4.1 Syntax of OOPL

Recall the syntax of OOPL:

\[
I \in Identifier \\
N \in Numeral \\
E ::= N \mid true \mid false \mid unit \mid null \mid I \mid E + E \mid E - E \mid E \ast E \\
| E < E \mid E == E \mid E \& E \mid E \lor E \mid not E \\
| E ::= E \mid E.I \mid E.I.Args \mid new I Args \mid cast E \mid (E)
\]

\[
Exps ::= _ \mid E, Exps \\
Arguments ::= (Exps)
\]

\[
C ::= skip \mid C; C \mid \{Block\} \mid E \mid return E \mid if B then Block else Block fi \mid while B do Block od
\]

\[
ty ::= int \mid bool \mid unit \mid null \mid I
\]

\[
Ids ::= _ \mid ty I, Ids \\
Params ::= (Ids)
\]

\[
VDec ::= ty I = E; \\
Block ::= C \mid VDec; Block
\]

\[
MDec ::= ty I Params \{Block\} \\
Decs ::= _ \mid VDec Decs \mid MDec Decs
\]

\[
CBody ::= \{Decs\} \\
CDec ::= class I extends I CBody
\]

We will treat a program as partial function from class names to a pair of the name of its parent and is declarations: 

\[
Prog : ClassName \rightarrow (ClassName \times Decs)
\]

There are some restrictions on OOPL programs not captured by the above grammar:

- Every class must define a method of the same name.
- \(Prog\) must have a class named \texttt{main}\) whose constructor params are just ( )
- We support here only \texttt{instance variables} and \texttt{instance methods} in class definition, which will change the semantics of \texttt{main} some, and will simplify semantics of \texttt{new} some.
- We assume a pre-existing “class” called \texttt{Object} where implicitly \texttt{Object extends Object}, the body of the constructor \texttt{Object} just returns \texttt{unit}, and there are no instance variables or other instance methods
- We do not support overloading of methods or variables at different types; we do, however, support \texttt{dynamic dispatch}.

The syntax of the second language SOOPL differs only in not including \texttt{cast}.

4.2 Natural Semantics of OOPL

We assume a type of \texttt{Location}. We will use \texttt{VarName} and \texttt{ClassName} in place of \texttt{Identifier} to be more suggestive of the intended use. The values in OOPL is given by the types:

\[
Env ::= (VarName \times ClassName) \rightarrow Location \\
Class ::= ClassName \times Env \\
Value ::= N \mid true \mid false \mid unit \mid null \mid Class \\
Mem ::= Location \rightarrow Value
\]

Looking a Field Up in \texttt{Dec}:

\[
\texttt{field\_in\_decs} I [] = \text{False} \\
\texttt{field\_in\_decs} I (MDec#Decs) = \text{field\_in\_decs} I Decs \\
\texttt{field\_in\_decs} I ((I' = E)#Decs) = (I = I') \lor \text{field\_in\_decs} I Decs
\]
Field Class Lookup:

Locally Found Variable:
\[ \text{prog} (\text{Cl}) = (\text{Cl}', \text{Devs}) \]
\[ \text{field in devs} \ I \ Devs \]
\[ \text{Cl} \ sees\_\text{field} \ I \ at \ Cl' \ in \ prog \]

Field of Ancestor:
\[ \text{prog} (\text{Cl}) = (\text{Cl}', \text{Devs}) \]
\[ \{\text{field in devs} \ I \ Devs\} \]
\[ \text{Cl}' \ sees\_\text{field} \ I \ at \ Cl'' \ in \ prog \]

LVar Evaluation:

Variables:
\[ \text{Cl} \ sees\_\text{field} \ I \ at \ Cl' \ in \ prog \]
\[ \text{(prog, Cl)} \vdash (I, (\text{env, mem})) \downarrow (\text{mem}(I, Cl'), \text{mem}) \]

Field Expression:
\[ \text{(prog, Cl)} \vdash (E, (\text{env, mem})) \downarrow ((\text{objcl, objenv}), \text{mem}') \]
\[ \text{objcl sees\_\text{field} \ I \ at \ Cl' \ in \ prog} \]
\[ \text{(prog, Cl)} \vdash (E.I, (\text{env, mem})) \downarrow (\text{objenv}(I, Cl'), \text{mem}') \]

Evaluation of Expressions:

Constants:
\[ \text{(prog, Cl)} \vdash (c, (\text{env, mem})) \downarrow (c, \text{mem}) \]

Identifiers:
\[ \text{Cl} \ sees\_\text{field} \ I \ at \ Cl' \ in \ prog \]
\[ \text{(prog, Cl)} \vdash (I, (\text{env, mem})) \downarrow (\text{mem}(I, Cl'), \text{mem}) \]

Field Expression:
\[ \text{(prog, Cl)} \vdash (E, (\text{env, mem})) \downarrow ((\text{objcl}, \text{objenv}), \text{mem}') \]
\[ \text{objcl sees\_\text{field} \ I \ at \ Cl' \ in \ prog} \]
\[ \text{(prog, Cl)} \vdash (E.I, (\text{env, mem})) \downarrow (\text{objenv}(I, Cl'), \text{mem}') \]

Assignment:
\[ \text{(prog, Cl)} \vdash (E, (\text{env, mem})) \downarrow (i, \text{mem}') \]
\[ \text{(prog, Cl)} \vdash (E', (\text{env, mem}')) \downarrow (v, \text{mem}'') \]
\[ \text{(prog, Cl)} \vdash (E := E', (\text{env, mem})) \downarrow (v, \text{mem}'''(i \leftarrow v)) \]

Add a new layer to an object

\[
\begin{align*}
\text{add\_class\_decs\_to\_parent}(\text{prog, Class}) & : ([], \text{parenv}, \text{mem})(\text{parenv}, \text{mem}) \\
\text{add\_class\_decs\_to\_parent}(\text{prog, Class}) & : (\text{decs, parenv}, \text{m})(\text{thisenv}, \text{mem}) \\
\text{add\_class\_decs\_to\_parent}(\text{prog, Class}) & : (\text{((ty I Params\{Block\})#decs, parenv), mem}) \\
\end{align*}
\]

Add a new layer to an object
\[ \text{(prog, Cl)} \vdash (E, (\text{env, mem})) \downarrow (v, \text{mem}') \]
\[ i \notin (\text{dom mem}') \]
\[ \text{nextenv} = (\text{parenv}(\text{I, Class}) \leftarrow i) \]
\[ \text{mem}''' = \text{mem}'(i \leftarrow v) \]
\[ \text{add\_class\_decs\_to\_parent}(\text{prog, Class}) : ((\text{decs, nextenv}, \text{mem}'')(\text{thisenv}, \text{mem}'''')) \\
\text{add\_class\_decs\_to\_parent}(\text{prog, Class}) : (\text{((ty I = E)#decs, parenv), m})(\text{thisenv}, \text{m}'''') \]
Initializing an object

\[
\text{init\_object}(\text{prog, cl})("\text{Object}", (\text{env, mem}))(\{\}, \text{mem})
\]

\[
\text{Class} \neq \"\text{Object}\"
\]

\[
\text{prog Class = (Parent, ClassDecs)}
\]

\[
\text{init\_object}(\text{prog, cl})(\text{Class, (env, mem)})(\text{parenv, mem}')
\]

\[
\text{add\_class\_decs\_to\_parent}(\text{prog, Class})((\text{ClassDecs, parenv}, \text{mem}')
\]

\[
(\text{thisobjenv, m''})
\]

\[
\text{init\_object}(\text{prog, cl})(\text{Class, (env, mem)})(\text{thisobjenv, mem''})
\]

Send

\[
(\text{prog, Cl}) \vdash (E, (\text{env, mem})) \Downarrow ((\text{ObjCl, objenv}), \text{mem}')
\]

\[
(\text{prog, Cl}) \vdash (\text{Args, (env, mem'}) \Downarrow (\text{vs}, \text{mem''})
\]

\[
\text{ObjCl sees methods MMap in prog}
\]

\[
\text{MMap method = ((ty, Params, Body), MClass)}
\]

\[
\text{extend\_env\_and\_mem\_with\_params\_and\_values}
\]

\[
\text{objenv mem'' ObjCl Params vs = (methenv, mem'''')}
\]

\[
(\text{prog, ObjCl}) \Downarrow (\text{Body, (methenv, mem'''')})(\text{Some}(v), \text{mem''''})
\]

New:

\[
(\text{prog, Cl}) \vdash (E.\text{method(Args), (env, mem)}) \Downarrow (v, \text{mem''''})
\]

\[
\text{init\_object}(\text{prog, cl})(\text{Class, (env, mem)})(\text{objenv, mem'})
\]

\[
(\text{prog, Cl}) \vdash (\text{Args, (env, mem'}) \Downarrow (\text{vs}, \text{mem''})
\]

\[
\text{Class sees methods MMap in prog}
\]

\[
\text{MMap method = ((ty, Params, Body), MClass)}
\]

\[
\text{extend\_env\_and\_mem\_with\_params\_and\_values}
\]

\[
\text{objenv mem'' Class Params vs = (methenv, mem'''')}
\]

\[
(\text{prog, Class}) \Downarrow (\text{Body, (methenv, mem'''')})(\text{Some}(v), \text{mem''''})
\]

\[
(\text{prog, Cl}) \vdash (\text{new Class Args, (env, mem)}) \Downarrow ((\text{Class, objenv}), \text{mem''''})
\]

Cast:

\[
(\text{prog, Cl}) \vdash (e, (\text{env, mem})) \Downarrow ((\text{Class'}, \text{objenv}), \text{mem'})
\]

\[
(\text{prog, Cl}) \vdash (\text{cast Class e, (env, mem)}) \Downarrow ((\text{Class, objenv}), \text{mem'})
\]
Commands:

<table>
<thead>
<tr>
<th>Skip:</th>
<th>Block:</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{prog, cl}) \vdash (\text{skip, (env, mem)}) \Downarrow (\text{None, mem}))</td>
<td>((\text{prog, cl}) \vdash (\text{Blk, (env, mem)}) \Downarrow (\text{vo, mem'}))</td>
</tr>
<tr>
<td>((\text{prog, cl}) \vdash (\text{C, (mem)}) \Downarrow (\text{None, mem'}))</td>
<td>((\text{prog, cl}) \vdash ({\text{Blk}}, (\text{env, mem})) \Downarrow (\text{vo, mem'}))</td>
</tr>
<tr>
<td>((\text{prog, cl}) \vdash (\text{C', (mem')}) \Downarrow (\text{vo, mem''}))</td>
<td>((\text{prog, cl}) \vdash (\text{C, (mem)}) \Downarrow (\text{Some(v), mem'}))</td>
</tr>
<tr>
<td>((\text{prog, cl}) \vdash (\text{C; C', (env, mem)}) \Downarrow (\text{vo, mem''}))</td>
<td>((\text{prog, cl}) \vdash (\text{C; C', (env, mem)}) \Downarrow (\text{Some(v), mem''}))</td>
</tr>
</tbody>
</table>

If-true:

| \((\text{prog, cl}) \vdash (\text{B, (env, mem)}) \Downarrow (\text{true, mem'})\) | \((\text{prog, cl}) \vdash (\text{C, (env, mem'}) \Downarrow (\text{vo, mem''})\) |
| \((\text{prog, cl}) \vdash (\text{if B then C else C'} fi, (\text{env, mem})) \Downarrow (\text{vo, mem''})\) | \((\text{prog, cl}) \vdash (\text{if B then C else C'} fi, (\text{env, mem})) \Downarrow (\text{vo, mem''})\) |

If-false:

| \((\text{prog, cl}) \vdash (\text{B, (env, mem)}) \Downarrow (\text{false, mem'})\) | \((\text{prog, cl}) \vdash (\text{C', (env, mem'}) \Downarrow (\text{vo, mem''})\) |
| \((\text{prog, cl}) \vdash (\text{if B then C else C'} fi, (\text{env, mem})) \Downarrow (\text{vo, mem''})\) | \((\text{prog, cl}) \vdash (\text{if B then C else C'} fi, (\text{env, mem})) \Downarrow (\text{vo, mem''})\) |

While-false:

| \((\text{prog, cl}) \vdash (\text{B, (env, mem)}) \Downarrow (\text{false, mem'})\) | \((\text{prog, cl}) \vdash (\text{while B do C od , (env, mem)}) \Downarrow (\text{None, mem'})\) |

While-true:

| \((\text{prog, cl}) \vdash (\text{B, (env, mem)}) \Downarrow (\text{true, mem'})\) | \((\text{prog, cl}) \vdash (\text{C, (env, mem'}) \Downarrow (\text{None, mem''})\) |
| \((\text{prog, cl}) \vdash (\text{while B do C od , (env, mem'}) \Downarrow (\text{vo, mem''})\) | \((\text{prog, cl}) \vdash (\text{while B do C od , (env, mem'}) \Downarrow (\text{vo, mem''})\) |

While-abort:

| \((\text{prog, cl}) \vdash (\text{B, (env, mem)}) \Downarrow (\text{true, mem'})\) | \((\text{prog, cl}) \vdash (\text{C, (env, mem'}) \Downarrow (\text{Some(v), mem''})\) |
| \((\text{prog, cl}) \vdash (\text{while B do C od , (env, mem'}) \Downarrow (\text{Some(v), mem''})\) | \((\text{prog, cl}) \vdash (\text{while B do C od , (env, mem'}) \Downarrow (\text{Some(v), mem''})\) |

Variable Declarations and Blocks:

Variable Declaration:

| \((\text{prog, Cl}) \vdash (\text{E, (env, mem)}) \Downarrow (\text{v, mem'})\) | \((\text{prog, cl}) \vdash (\text{ty I = E, (env, mem)}) \Downarrow (\text{env[I \leftarrow i], mem'[i \leftarrow v]})\) |
| \((\text{prog, cl}) \vdash (\text{i \notin \text{rng(env)} \cup \text{dom(mem)})}\) | \((\text{prog, cl}) \vdash (\text{ty I = E, (env, mem)}) \Downarrow (\text{env[I \leftarrow i], mem'[i \leftarrow v]})\) |

Blocks - No Decs

| \((\text{prog, cl}) \vdash (\text{C, (env, mem)}) \Downarrow (\text{vo, mem'})\) | \((\text{prog, cl}) \vdash (\text{C, (env, mem)}) \Downarrow (\text{vo, mem'})\) |

Block - Dec. Seq.

| \((\text{prog, cl}) \vdash (\text{D, (env, mem)}) \Downarrow (\text{env', mem'})\) | \((\text{prog, cl}) \vdash (\text{Blk, (env', mem'}) \Downarrow (\text{vo, mem''})\) |
| \((\text{D; Blk, (env, mem)}) \Downarrow (\text{vo, mem''})\) | 

5
5 Problem

1. (20 pts) In shared_syntax_and_semantics.thy are the types and functions shared in common between the variant of our simple object-oriented programming language (OOPL) supporting cast, and the even simpler variant you are asked to work with (SOOPL), which is identical except that it does not support cast. In the file oopl.thy you are given a datatype describing the abstract syntax for expressions in OOPL, the mutually inductive relations describing the Natural Semantics for OOPL evaluation. In the file mp4.thy, you are given the type for expressions in SOOPL (hence with cast missing), and you are given a skeleton, including types and a few clauses for mutually inductive relations describing the Natural Semantics for SOOPL evaluation.

Complete the definition of the Natural Semantics of SOOPL, putting your results in mp4.thy. Do not change the names or types of the relations given, but you may right auxiliary functions, and add additional relations to the mutual induction if you feel it is appropriate. Besides adhering to the types given in mp4.thy, your definition of evaluation in SOOPL should satisfy the following property: If you take an OOPL program that does not use cast and evaluate it as a SOOPL program, you will get the same result as if you evaluate it as an OOPL program and view the result as a SOOPL program. (We also want to know if a program evaluates in SOOPL, then it also evaluates in OOPL.)

In the file correctness.thy is a statement of correctness of the relation between OOPL and SOOPL evaluation, and the background definitions supporting it. I could not find a proof that did not use the specific names and precise statements of the clauses for SOOPL evaluation making it not practical for use of testing your work, and given that the proof is quite long, I have omitted it. The file is included for your general information only.