Dynamic Semantics and Type-checking of Tuple

Jianbing Chen

TR #98-10
December 1998

This work was supported in part by grant CCR 9803843 from the US National Science Foundation and by a grant from Rockwell International.

Keywords: Tuple language, single dispatch, multiple dispatch, multimethods, subtype, type checking, semantic prototype, tuples.

1997 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.2 [*Programming Languages*] Languages Classifications — object-oriented languages; D.3.3 [*Programming Languages*] Languages Constructs and Features — abstract data types, control structures, procedures, functions, and subroutines; D.3.m [*Programming Languages*] Miscellaneous — multimethods, single dispatch, multiple dispatch, type systems; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — denotational semantics; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — control primitives, type structure.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Dynamic Semantics and Type-checking of Tuple

Jianbing Chen

December 15, 1998

Abstract

In this paper, we address some issues in the multiple dispatch language Tuple. Specifically, the purpose and basic ideas of the language Tuple, our implementation in Haskell of the dynamic semantics and type checking, and the soundness proof of it's static type system are presented. The program for the dynamic semantics looks like the denotional semantics of the language. And the program for type checking resembles the type inference rules in logic.

1 Introduction

An object-oriented programming language has *single dispatch* when only the dynamic class of the message's receiver, a single object, is used to select the method to execute in response to the message. Here, a *message* consists of the name of method being invoked and the actual arguments. The *receiver* of the message is the object instance that support the corresponding method. For example, in a function call $o.g(x)$, the message is $g(x)$, the receiver is o. *Multiple dispatch*, on the other hand, selects a method based on the dynamic class of any subset of the message's arguments. Many popular object-oriented programming languages, such as C++, Smalltalk-80, Java, and Eiffel, do not support multiple dispatch. However, programmers find it difficult to express binary methods and design patterns such as the "visitor" pattern [6, p.331-334] without multiple dispatch. We now give two examples to illustrate this point. Although examples are written in Java [1], the problem is common to all languages which do not support multiple dispatch.

1.1 The Problem with Single Dispatch

In Figure 1, two related classes have been defined. (These are inspired by Boyland and Castagna's article [3].) The meaning of the two classes are straightforward. Since we have much more efficient algorithms for merging two sorted lists than merging one sorted list and one unsorted list, method `union` in class `SortedNumberList` uses `instanceof` to distinguish the cases where the dynamic class of the actual parameter is `NumberList` or `SortedNumberList`. However, in object-oriented programming, this is not preferrable.

The second example (taken from the article [2]) presents a more general problem that is solved using a general technique known as "visitor" design pattern [6]. In Figure 2, abstract syntax trees and operations on these trees in the context of a compiler have been considered. Abstract syntax trees are built from nodes for assignments, variable references, expressions and so on. Operations on abstract syntax trees include type checking, code generation, flow analysis, etc.

In order to make the program easier to understand and maintain, the "visitor" pattern expects the node classes to have a single method called `Accept` that takes a visitor object as an argument. Then the pattern further relies on separate visitor classes corresponding to the operations. When the

```
public class NumberList {
  public NumberList union(NumberList) {
      // body for unioning two lists
  }
}

public class SortedNumberList extends NumberList {
  public SortedNumberList union(NumberList nl) {
    if ( nl instanceof SortedNumberList) {
        // merge two sorted lists
    } else {
        // merge a sorted list with a unsorted list
    }
  }
}
```

Figure 1: Merging two lists in Java. (has to be in two files)

```java
public abstract class Node {
  public abstract void Accept(NodeVisitor);
  //...
}

public class AssignmentNode extends Node {
  public void Accept(NodeVisitor v) { v.VisitAssignment(this);}
  //...
}

public class VariableRefNode extends Node {
  public void Accept(NodeVisitor v) { v.VisitVariableRef(this);}
  //...
}

public abstract class NodeVisitor {
  public abstract void VisitAssignment(Node);
  public abstract void VisitVariableRef(Node);
  //...
}

public class TypeCheckingVisitor extends NodeVisitor {
  public void VisitAssignment(Node) {// body for visiting}
  public void VisitVariableRef(Node) {// body for visiting}
  //...
}

public class CodeGeneratingVisitor extends NodeVisitor {
  public void VisitAssignment(Node) {// body for visiting}
  public void VisitVariableRef(Node) {// body for visiting}
  //...
}
```

Figure 2: Visitor pattern in Java. (has to be in several files)

`Accept` method is invoked, method selection is first based on the node class and then the visitation method is selected based on the actual visitor class. Hence the "visitor" pattern is implemented using double dispatching [8].

A couple of problems arise when using the "visitor" pattern. First, it is difficult to add new element classes. Each new element class `XNode` requires defining a method `VisitX` and corresponding method implementations in each visitor class to be added. This makes the visitor class library difficult to maintain. Furthermore, double dispatching is error-prone because it requires method selection to be implemented manually. This makes it hard to see which argument combination causes the execution of which method.

1.2 Tuple's Solution

To solve the above problems, Leavens and Millstein designed a language called Tuple [9]. The idea is to extend the single-dispatch language (e.g, Java) with tuple expressions, tuple classes, tuple types, and the ability to declare and send message to tuples, which gives multiple dispatch. To illustrate, we will give answers to the above two examples in the syntax of Tuple.

In Figure 3, three tuple classes have been defined. Each tuple class has different tuple formals. Based on the run time classes of the receiver tuple, the appropriate `union` method is selected from the three tuple classes. Note if snl is an instance of `SortedNumberList` and nl is an instance of `NumberList`, then $(snl, nl).union()$ will invoke the method defined in the last tuple class. However, $(nl, snl).union()$ will invoke the method defined in the first tuple class by the inheritance.

The solution for the second example is given in Figure 4. Here, for each different node and visitor pair, a tuple class is defined. The corresponding visiting method is defined in the tuple class. In this way, the semantics is clear and the code is easy to modify.

1.3 Contribution of this Paper

Based on the original Tuple paper [9], we have implemented the dynamic semantics and type checking of Tuple in the functional languag Haskell [11]. Using our prototype system, we have gained experience coding in Tuple and also proved the soundness of the type system. Also, we added keyword `super` to the original language to enable inheritance of methods to be mixed with overriding, as in Smalltalk.

```
tuple class(nl1:NumberList, nl2:NumberList) {
  method union():NumberList {
    --body for unioning two lists
  }
}

tuple class(snl1:SortedNumberList, snl2:SortedNumberList) {
  method union():SortedNumberList {
    --body for unioning two sorted lists
  }
}

tuple class(snl1:SortedNumberList, nl2:NumberList) {
  method union():SortedNumberList {
    --body for unioning one sorted list with one list
  }
}
```

Figure 3: Tuple solution for Merging lists in file `NumberList.tup`.

```
tuple class(n:AssignmentNode, v:TypeCheckingVisitor) {
  method visit():void {
    --body for visiting
  }
}

tuple class(n:AssignmentNode, v:CodeGeneratingVisitor) {
  method visit():void {
    --body for visiting
  }
}

tuple class(n:VariableRefNode, v:TypeCheckingVisitor) {
  method visit():void {
    --body for visiting
  }
}

tuple class(n:VariableRefNode, v:CodeGeneratingVisitor) {
  method visit():void {
    --body for visiting
  }
}
```

Figure 4: Tuple solution for Visitor pattern in file `Visitor.tup`.

The main difficulty we encountered in the semantics is the processing of recursive declaration sequences (see the next section). Since the declarations need not to be sequential, we need a mechanism to pass the correct environment in the dynamic semantics and type checking.

Thanks to Haskell, our code for dynamics semantics looks like the denotational semantics of the language. It is highly structural and clean. Based on the annotation tools developed by Leavens, we were able to make the implementation of the type checking rules look like inference rules used in the original paper by Leavens and Millstein [9].

1.4 Outline of the Rest Paper

In Section 2, we will discuss the abstract syntax and dynamic semantics of the language Tuple. In Section 3, we explain some issues in type checking Tuple programs. The soundness proof of the static type system with respect to the dynamic semantics is presented in Section 3. Then we give the conclusions and future work in Section 4. In the Appendix, we give the major program listings of our implementation.

2 Prototyping Tuple's Dynamic Semantics

In this section, we introduce the syntax and dynamic semantics of Tuple by our implementation. The abstract syntax of Tuple will be examined first. Then we will look into the run-time environments and the implementation of the dynamic semantics.

2.1 Abstract Syntax

The abstract syntax of Tuple written in the programming language Haskell is given in Figure 5.

The main components of the abstract syntax are declarations and expressions. The rest is for type checking. For example, `TypeContext` and components of `TypeContext` are used only in type checking.

There are two kinds of declarations: class declarations and tuple class declarations. Each class declaration has zero or more fields and zero or more methods. Tuple classes only have one or more methods. A method declaration includes the formals and return type, and an expression for the method body.

We will only explain the first three expressions. The rest of them should be clear by themselves. `New ClassName [E]` is the syntax for instantiating an object for the class `ClassName`. The expression `Dot E MethodName [E]` allows us to send messages to both class objects and tuple class intances. We use `Tuple [E]` to denote a tuple. So for example, the following Tuple program

```
class P {
     fields (x:int)
     method k(y:int):int {x*y}
}
(new P(4)).k(3)
```

would be represented by the following abstract syntax tree:

```
RDS [Class "P" []
      ["x" :< Aint]
      [Method "k" ["y" :< Aint] Aint
           (Ident "x" `Times` Ident "y")]
] `In`
(New "P" [Num 4] `Dot` "k") [Num 3]
```

```
data Program = RecursiveDeclarationSequence `In` Expression
data RecursiveDeclarationSequence = RDS [Declaration]
data Declaration = TupleClass [IdentifierTyping] [Method]
       | Class ClassName [ClassName] [IdentifierTyping] [Method]
data IdentifierTyping = Identifier :< Type
data Type = Aint | Abool | Product [Type] | Type :-> Type
       | Aname Identifier | Ameth (Type, Type)
       | Acls ClassInfo | Atup TupleClassInfo
       | Acon TypeContext | Api TypeAssignment
       | Aexp Type | Adec Type | Avar LogicalVar | Aerr
type TypeContext = ((TypeAssignment,ClassEnv,Tuples),Inheritance)
type Inheritance = FiniteRel Type Type
type ClassEnv = TypeAssignment
type ClassInfo = ([IdentifierTyping],TypeAssignment,ClassName)
type Tuples = FiniteRel Type Type
type TupleClassInfo = ([IdentifierTyping],TypeAssignment)
type TypeAssignment = FiniteFun Identifier Type
data Method = Method MethodName [IdentifierTyping] Type Expression
data Expression = New ClassName [E]
       | Dot E MethodName [E] | Tuple [E] | Self | Super
       | Ident Identifier | E `Plus` E | E `Minus` E
       | E `Times` E | E `Lt` E | E `Gt` E | E `Le` E
       | E `Ge` E | Not E | E `Equals` E
       | Num Integer | BoolLit Bool | E `And` E | E `Or` E
       | If E E E | E `Semi` E
type E = Expression
type ClassName = Identifier
type MethodName = Identifier
type Identifier = String
```

Figure 5: Abstract syntax of Tuple

2.2 Run-time Environments and Dynamic Semantics

In a top-down fashion, the run-time environments of our implementation are described in Figure 6. A run-time environment of type `EnvInh` consists of a `DEnvironment` and an `InheritanceRel`. A `DEnvironment` is a triple of an `Environment`, a `CEnvironment` and a `TEnvironment`; they represent run-time bindings, class environments and tuple environments respectively. Runtime bindings are the bindings of identifiers to `DenotableValue`s, which are integers, booleans, objects or tuple instances. Note that we use a `data` declaration to define `DEnvrionment` so we can print out a recursive environment by making an instance of the `show` class in Haskell. If this is not done, then printing will loop. Each `DVClassInfo` or `DVTupleInfo` contains a method dictionary `DVMethDict`, which corresponds a table of methods for a certain class or a tuple class. One thing which is worth mentioning is the definition of a method. We define it in Haskell as a pair of the formal list and a curried Haskell function which takes the actuals and the current environment and returns an `ExpressedValue`. In Tuple, expressed and denotable values are the same.

Now we are ready to discuss the implementation of dynamic semantics of Tuple. Based on the abstract syntax, we identified the following two phases of dynamic semantics processing:

(1) Processing declarations to build the run-time environment

(2) Use the environment to evaluate expressions.

These two phases are needed because we allow recursive declarations. Processing all the declaration sequences in one sweep would be difficult. In general, recursive declaration sequences requires that the run-time environment be built in a fixed point fashion, i.e., a continuing process until the environment does not change any more. In our case, since the only dependency is subclass dependency, we are able to get the final run-time environment in a second pass over the environment. This is done by first building a tree of relationships of classes with class "Top" being the root (in our type system, "Top" is super class of every other class) and getting a list of classes by preorder traversing the tree. Then the fields and method dictionaries can be updated in that order.

Furthermore, by using the laziness of Haskell, the actual code of processing recursive declaration squence is very similar to the code of processing

```
type EnvInh = (DEnvironment, InheritanceRel)
data DEnvironment = DEnv (Environment,CEnvironment,TEnvironment)
data InheritanceRel = FiniteRel ClassId ClassId
type Environment = FiniteFun Identifier DV
type CEnvironment = FiniteFun ClassId DVClassInfo
type TEnvironment = FiniteRel ClassIdTree DVTupleInfo
data DVClassInfo = DVCI (ClassId, Fields, DVMethDict)
data DVTupleInfo= DVTI (TupleFormals, DVMethDict)
type Fields = FormalList
type TupleFormals = FormalList
type FormalList = FiniteFun Identifier Type
type DVMethDict = FiniteFun Label DVMethod
data DVMethod = DVM (FormalList, ([DV] -> Environment -> EV))
type DV = DenotableValue
type EV = ExpressedValue
type ExpressedValue = DenotableValue
data DenotableValue =
          DVInt DInt | DVBool DBool
        | DVObject ClassId DVRecord
        | DVTuple [DV]
type DVRecord = FiniteFun Label DV
data ClassIdTree = ClassIdSingle ClassId
        | ClassIdTuple [ClassIdTree]
type Label = String
type ClassId = String
```

Figure 6: Run-time environments of Tuple

sequential declarations. The reason is that the laziness of Haskell enables us to pass a future computation around since the evaluation will occur only when needed. So as long as at the time of the evaluation we have all the information needed, the program is okay. Without this feature, we would have to pass a thunk instead. For example, if we have had chosen to implement the dynamic semantics in programming language ML [12], we would have to code the meaning function `meaningE` with the following (Haskell syntax) type:

```
meaningE :: Expression -> (() -> EnhInh)
                        -> (() -> ExpressedValue)
```

Instead, we have the following:

```
meaningE :: Expression -> EnhInh -> ExpressedValue
```

which is simpler and clearer.

After we obtain the class and tuple environments from the declaration processing phase, expression evaluation is relatively easy. The only tricky part is method dispatch on tuple classes. According to the number of most specific methods been found being one, zero, or more than one, there are three possible outcomes: correct message, method not found and ambiguous message. For two tuple types that both have the same length and a method of the same name, one tuple is more specific than the other if it is pointwise a subtype of the other. For example, suppose that we have defined two classes `Foo` and `Bar` and `Bar` is a subclass of `Foo`, then the tuple `(Foo,Bar)` is more specific than the tuple `(Foo,Foo)` if both tuple classed are defined. In fact, our type-checking makes sure that method not found and ambiguous message errors will never occur at run-time.

The complete listing of dynamic semantics is given in the Appendix.

3 Type Checking Tuple

In this section, our implementation of type checking Tuple is presented. This implementation is based on the framework laid in the original paper [9] by Leavens and Millstein.

The abstract syntax of types in Tuple is described in Figure 5. There, `Type` gives our primitive types and the skeleton of our type system. There are two kinds of types. The types in the one group are called *atomic* types, which include `Aint, Abool, Aname ClassName`. These are types for integers, booleans, class instances. For example, `Aname "foo"` would be the type of an instance of class "foo". The rest types are called *composite* types. `Product [Type]` is the type for tuples. In Haskell, `[a]` means a list of elements of type `a`. Such a type is polymorphic, as `a` can be replaced by any type. Thus, `Product [Aint, Abool]` is the type of a pair of an integer and a boolean. `Type :-> Type` represents a function's signature. Here, `:->` is type constructor in Haskell. `Ameth (Type,Type)` is method type, which contains the signature of a method and the type assignment of an object's fields. `Avar LogicalVar` is the type for logical variables. `Aerr` is the error type (bottom type). `Aexp Type` denotes an expression type [10]. `Acls ClassInfo` and `Atup TupleClassInfo` denote the types of declarations of classes and tuple classes. `Acon TypeContext` is the type for a type context that results from some declarations.

Our approach to type checking a Tuple program is similar to the one we used in the dynamic semantics. Once again, type checking hinges on the processing of recursive declaration sequence. This processing has two passes. During the first pass, the type information is collected and all methods declarations are assumed to be correct. During the second pass, fields of subclasses are updated and methods of classes or tuple classes are checked. Before going into the details of method type checking, we need to address the semantics of subtyping in Tuple. The subtype relationship of two types with respect to some type context can be described by the following algorithm:

```
> subTypes :: Type -> Type -> TypeContext -> Bool
> subTypes Aint Aint k = True
> subTypes Abool Abool k = True
> subTypes (Aname cn) (Aname cn') (_,inh) =
>   (Aname cn, Aname cn') `elem` inh
> subTypes (Product types) (Product types') k =
```

```
>   ((length types) == (length types')) &&
>      (and (map (\(x,y) -> subTypes x y k) pairs))
>      where pairs = zip types types'
> subTypes (paras :-> return) (paras' :-> return') k =
>   (subTypes return return' k) &&
>      (subTypes paras' paras k)
> subTypes (Ameth (type1,type2)) (Ameth (type1',type2')) k =
>      (subTypes type1 type1' k) && (subTypes type2 type2' k)
> subTypes (Api pi) (Api pi') k =
>      all (\(i,t') -> case (apply pi i) of
>                          (Just t) -> subTypes t t' k
>                          Nothing  -> False) pi
> subTypes _ _ k = False
```

The function `subTypes` takes two types and a type context and does pattern matching recursively to test the subtyping. Subtyping for class names involves the inheritance relation in the class environment which is assumed to be reflexive and transitively closed. Subtype checking for two arrow types corresponds to the contravariant rule [4]. We will use $t_2 \leq t_1$ to denote that t_2 is a subtype of t_1. Also, we use `Aname top` = `Aname "Top"` to denote the super class of all classes.

To type check a method, the method body (which is an expression) is checked under the current type environment to make sure it has the declared signature. Furthermore, there are two other kinds of checking that are related to type checking methods.

In type checking a class declaration, the following is checked to ensure that every subclass implements a subtype of its superclass: for every method m in class CN_1, a subclass of CN_0, m's type in CN_1 must be a subtype of m's type in CN_0 if it exists in class CN_0.

Implementation-side checking [5] is performed after we have complete information about the type environment. This is for methods defined in tuple classes. Two checks are made. The first check ensures *monotonicity*. Let $(S_1, \cdots, S_k) \to U$ and $(S'_1, \cdots, S'_k) \to U'$ be the types of the methods named I in tuple classes for the tuples $(T_1, \cdots, T_n)$ and $(T'_1, \cdots, T'_n)$ respectively. Suppose that $(T_1, \cdots, T_n)$ is a subtype of $(T'_1, \cdots, T'_n)$, then $(S_1, \cdots, S_k) \to U$ must be a subtype of $(S'_1, \cdots, S'_k) \to U'$. The second check ensures that two methods are not ambiguous. We define that two types T and T' are *related* if either T subtypes T' or vice versa. In this case, $\min(T, T')$ denotes the one of T and T' that subtypes the other. If two tuples $(T_1, \cdots, T_n)$ and $(T'_1, \cdots, T'_n)$ both have a method m and the T_i are related to T'_i for all $1 \leq i \leq n$, then the method m must be defined in a tuple class for the tuple $(min(T_1, T'_1), \cdots, min(T_n, T'_n))$. When there are no ambiguous tuple methods in the type context, each method sent to a tuple instance correspond to at most one method in some tuple class. We call this method the *most specific method*. Note here also that the implementation-side checking works because the language has single inheritance.

We call a type context, $k = ((\pi, cs, ts), inhr)$, *valid* if all the atomic types in π, ts and $inhr$ are either `Aint`, `Abool`, or a type in cs. This basically says that the type context is self-contained. Our type checking process ensures that we always have a valid type context if we start with a valid one. From now on, all type contexts in our discussion are assumed to be valid. In addition, if a valid type context contains no ambiguous tuple methods, we call it a *good* type context. Clearly, if a type context passes implementation-side checking, then it is a good type context.

4 Soundness of our Type System with respect to the Dynamic Semantics

We first give a brief overview on soundness of type systems. Then our proof in Schmidt's style [10] is presented.

4.1 Soundness, Why and How?

"For some applications, correctness is especially important. Examples include life-critical systems such as nuclear reactor controllers, car braking systems, fly-by-wire aircraft and software controlled medical equipment."... "Formal proof of correctness provides a way of establishing the absence of bugs when exhaustive testing is impossible (as it almost always is)." [7, p.14]

In general, there are two approaches to ensure the soundness of some type system:

(i) Define the language by the axioms and rules of the logic.

(ii) Prove that the logic fits the language.

"Approach (i) is called *axiomatic semantics*. The idea is to *define* the semantics of the language by requiring that it make the axioms and rules of inference true. It is then up to implementers to ensure that the logic matches the language. One snag with this approach is that most existing languages have already been defined in some other way (usually by informal and ambiguous natural language statements). An example of a language defined axiomatically is Euclid. The other snag with axiomatic semantics is that it is known to be impossible to devise complete Floyd-Hoare logics for certain constructs." "Approach (ii) requires that the axioms and rules of the logic be proved valid. To do this, a mathematical model of states is constructed and then a function, **Meaning** say, is defined which takes an arbitrary command C to a function **Meaning**(C) from states to states."... "It is then possible to attempt to prove rigorously that all the axioms are true and the inference rules lead from true premisses to true conclusions."[7, p.32] Since we have a prototype system with both dynamic semantics and type checking, we take the second approach in this paper.

4.2 Preliminary Definitions

We first give some preliminary definitions. Then we define the meaning of types, $[\![\cdot]\!]$, and the notion of consistency between a type context and a type environment; these are defined in terms of each other.

Let $e = DEnv\ ((E, CE, TE), i)$ be a type environment. In the following, we use short-hand notation $((E, CE, TE), i)$ for e, and we use auxiliary function inh as follows:

$$inh(((E, CE, TE), i)) = i$$

We say a *DVRecord* r *corresponds* to a class c in e if $(c, DVCI\ (sid, \{i_j : t_j\}_{j \in J}, \pi)) \in CE$ and for any i_j, $j \in J$, $(i_j, v_{i_j}) \in r$ with $v_{i_j} \in [\![t_j]\!]e$. Let π be an instance of `TypeAssignment`. Let E' be an instance of `Environment`. We say E' *corresponds* to π in e if for any $(i, t_i) \in \pi$, there is a $(i, v_i) \in E'$ such that $v_i \in [\![t_i]\!]e$.

The following gives the meanings of the types. We use the notation *DVInt Int* for the subset of *Denotable Value* $\{DVInt\ i \mid i \in Int\}$ and similarly for *DVBool Bool*. We use the notation $(e : EnvInh) \rightarrow T(e)$ for dependent type, i.e., e is the element of *EnvInh* that is passed as an argument, and $T(e)$ is some type involving e. For example, the meaning of type $Product\ [t_1, \cdots, t_n]$ is a type that takes an element e of *EnvInh* and maps it to $\{DVTuple\ [v_1, \cdots, v_n] \mid v_1 \in [\![t_1]\!]e, \cdots, v_n \in [\![t_n]\!]e\}$, so that each component is drawn from the meaning of the corresponding type in e. Finally, we use subscript $\perp$ to denote that looping is included in the type.

$$[\![Aint]\!] = EnvInh \rightarrow DVInt\ Int$$

$$[\![Abool]\!] = EnvInh \rightarrow DVBool\ Bool$$

$$\begin{aligned}&[\![Product\ [t_1, \cdots, t_n]]\!] = (e : EnvInh) \rightarrow \\ &\quad \{DVTuple\ [v_1, \cdots, v_n] \mid v_1 \in [\![t_1]\!]e, \cdots, v_n \in [\![t_n]\!]e\}\end{aligned}$$

$$\begin{aligned}&[\![Aname\ I]\!] = (e : EnvInh) \rightarrow \textstyle\bigcup_{(S,I) \in inh(e)} \{o \mid e = ((E, CE, TE), i), \\ &\quad (id, o = DVObject\ S\ r) \in E\ \textit{for some id and r, r corresponds to}\ S\}\end{aligned}$$

$$[\![Aexp\ T]\!] = (e : EnvInh) \rightarrow [\![T]\!]e_\perp$$

$$\begin{aligned}&[\![Ameth\ (Product\ [t_1, \cdots, t_n] :\rightarrow t, Api\ \pi')]\!] = (e : EnvInh) \rightarrow \\ &\quad \{DVMD\ ([i_1 : t_1, \cdots, i_n : t_n], f) \mid f :: ([\![t_1]\!]e \times \cdots \times [\![t_n]\!]e) \rightarrow \\ &\quad (E' : Environment) \rightarrow [\![t]\!]e_\perp, \textit{where}\ E'\ \textit{corresponds to}\ \pi'\}\end{aligned}$$

$$[\![Api\ \{i : \theta_i\}_{i \in J}]\!] = (e : EnvInh) \rightarrow \{i : [\![\theta_i]\!]e\}_{i \in J}$$

$$[\![Acls\ (\{i :< t_i\}_{i \in I}, \varpi, cn)]\!] = (e : EnvInh) \rightarrow$$
$$\{c \mid c = DVCI\ (cn, \{i : t_i\}_{i \in I}, md), md \in [\![\varpi]\!]e\}$$

$$[\![Atup\ (\{i :< t_i\}_{i \in I}, \varpi)]\!] = (e : EnvInh) \rightarrow$$
$$\{t \mid t = DVTI\ (\{i : t_i\}_{i \in I}, md), md \in [\![\varpi]\!]e\}$$

$$[\![Adec\ (Acon\ k)]\!] = EnvInh \rightarrow \{e' \mid e' \textit{ is consistent with } k\}$$

We say that $e = ((E, CE, TE), i)$ is *consistent* with $k = ((\pi, cs, ts), inhr)$ if the following conditions are true:

(1). $\forall\ (o, t) \in \pi,\ \exists\ (o, v) \in E$ and $v \in [\![t]\!]e$;

(2). $\forall\ (c, cif) \in cs,\ \exists\ (c, dvcif) \in CE$ and $dvcif \in [\![cif]\!]e$;

(3). $\forall\ (t, tif) \in ts,\ \exists\ (t, dvtif) \in TE$ and $dvtif \in [\![tif]\!]e$;

(4). $i = inhr$.

From the definition of $[\![Aname\ I]\!]$, we can see that the following is true:

Corollary 4.0 Let e be any type environment. If c_2 is a subclass of c_1 in e, then $[\![Aname\ c_2]\!]e \subset [\![Aname\ c_1]\!]e$.

4.3 Soundness Proof

Our goal is the following:
Theorem 4.1 Let k be a type context and let e be a type environment. Let U be a syntactic phrase except a method. If $k \vdash U : t$ and e is *consistent* with k, then $[\![U]\!]e \in [\![t]\!]e_\perp$.

For methods, it turns out that we have to use different hypothesis, although we have similar result. The proofs for expressions, methods, declarations and recursive declaration sequences are given first. Then we prove the result for programs. All these proofs are using structural induction.

4.3.1 Expressions

Lemma 4.2 Let k be a good type context and let e be a type environment. Let *expr* be an expression. If $k \vdash$ *expr: Aexp* τ and e is consistent with k, then $[\![expr]\!]e \in [\![\tau]\!]e_\perp$.

Proof: For brevity, we only show the proofs for class instantiation and method invocation. The rest are straightforward.

Let $k = ((\pi, cs, ts), inhr)$ be a good type context. Let $e = ((E, CE, TE), i)$ be a type environment that is consistent with the k. Assume that $k \vdash$ *expr: Aexp* τ. We use the notation *MD* for a method dictionary. We use notation $dom(g)$ to represent the domain of a finite function g.

1. Suppose $expr = New\ cn\ [e_1, \cdots, e_n]$:
Suppose that $k \vdash expr : Aexp\ (Aname\ cn)$ for some class cn. We need to show that $[\![expr]\!]e \in [\![Aname\ cn]\!]e_\perp$. Since *expr* type checks, there exists some $cif = (\{i_1 :< t_1, \cdots, i_n :< t_n\}, \varpi, sid)$ such that $(cn, cif) \in cs$. By the consistency of e with k, there exists some $dvcif = (sid, \{i_1 : t_1, \cdots, i_n : t_n\}, MD)$ such that $(cn, DVCI\ cif) \in CE$ and $MD \in [\![\varpi]\!]e$. The type rule requires that $k \vdash e_j : t_j$ where $1 \leq j \leq n$. By the inductive hypothesis, $[\![e_j]\!]e \in [\![t_j]\!]e$. By the dynamic semantics, $[\![expr]\!]e = DVObject\ cn\ rec$, where $rec = zip\ [i_1, \cdots, i_n]\ [[\![e_1]\!]e, \cdots, [\![e_n]\!]e]$, which gives the result we need.

2. Suppose $expr = (Tuple\ es)$ *'Dot'* $mn\ [e_1, \cdots, e_n]$:
Then we need to prove that $[\![expr]\!]e \in [\![\tau]\!]e_\perp$. Assume $k \vdash Tuple\ es : \tau_0$ and $\tau_0 = Aexp\ (Product\ [t_1, \cdots, t_m])$ for some types $t_1, \cdots, t_m$. The type rule requires that:

(1) There are types $t'_1, \cdots, t'_m$ such that there exists the most specific method in the tuple class $(Product\ [t'_1, \cdots, t'_m], Atup\ tpflds, \varpi)) \in ts$ and $t_j \leq t'_j$ for $1 \leq j \leq m$

(2) $k \vdash e_j : s_j$ for some s_j, where $1 \leq j \leq n$

(3) $(mn, Ameth\ (Product\ [s_1, \cdots, s_n] :\rightarrow \tau, Api\ \pi')) \in \varpi$

By the consistency of e with k and the induction hypothesis, we have:

(4) $(ClassIdTuple\ [t'_1, \cdots, t'_m], DVTI\ (tpflds,\ MD)) \in TE$ such that $MD \in [\![\varpi]\!]e$

(5) $(mn,\ DVMD\ (flds,\ f)) \in methdict$ where $f :: ([\![s_1]\!]e \times \cdots \times [\![s_m]\!]e) \rightarrow (E' : Environment) \rightarrow [\![\tau]\!]e_\perp$ and E' corresponds to π'

(6) $[\![e_0]\!]e = DVTuple\ [dv_1, \cdots, dv_m]$ for $dv_i \in [\![t_i]\!]e$ and $1 \leq i \leq m$

(7) $[\![e_j]\!]e \in [\![s_j]\!]e$ for $1 \le j \le n$
Hence, by the definition of the semantics, we have
$[\![k \vdash E : s\ exp]\!]e = f\ actuals\ rec$, where
$actuals = [[\![e_1]\!]e, \cdots, [\![e_n]\!]e]$ and
$rec = zip\ (fst\ tpfls)\ [dv_1, \cdots, dv_m]$.

The bindings for all the tuple formals are contained in *rec*, so *rec* corresponds to π'. By the type of f, it follows that $[\![expr]\!]e \in [\![\tau]\!]e_\perp$.

3. Suppose $expr = e_0$ *'Dot'* $mn\ [e_1, \cdots, e_n]$:
Then we need to prove that $[\![expr]\!]e \in [\![\tau]\!]e_\perp$. Assume $k \vdash e_0 : \tau_0$ and $\tau_0 = Aexp\ (Aname\ cn)$ for some cn. By the type rule, there exists some $cif = (\{i_1 :< s_1, \cdots, i_m :< s_m\}, \varpi, sid)$ such that $(cn, cif) \in cs$ and $(mn : Ameth\ (Product[t_1, \cdots, t_n] :\rightarrow \tau, Api\ \pi')) \in MD$. Furthermore, the type rule requires that, $k \vdash e_j : t_j$ where $1 \le j \le n$. By the inductive hypothesis, $[\![e_0]\!]e = DVObject\ cn'\ rec$ for some *DVRecord rec* and class name cn' such that *rec* corresponds to cn', $(cn', cn) \in inhr$ and $(cn', dvcif') \in CE$ for some $dvcif' = DVCI\ (sid', flds, MD)$. Since cn' is a subclass of cn, $i_j \in dom(flds)$ and each element of $[\![\varpi]\!]e$ is a subset of MD. Therefore, $mn \in dom(MD)$ and has a subtype of $Ameth\ (Product\ [t_1, \cdots, t_n] :\rightarrow \tau, Api\ \pi')$. Let $(mn, DVMD\ ([p_1 : t'_1, \cdots, p_n : t'_n], f)) \in MD$. Then f has the type: $([\![t'_1]\!]e \times \cdots \times [\![t'_n]\!]e) \rightarrow (E' : Environment) \rightarrow [\![\tau']\!]e_\perp$, where E' corresponds to π', $t_j \le t'_j$ for $1 \le j \le n$ and $\tau' \le \tau$. Now by the definition of the dynamic semantics, $[\![expr]\!]e = f\ actuals\ rec'$, where
$actuals = [[\![e_1]\!]e, \cdots, [\![e_n]\!]e]$ and
$rec' = rec \cup \{(\text{“}self\text{”}, (DVOject\ cn'\ rec)\} \cup \{(\text{“}super\text{”}, (DVObject\ sid\ rec)\}$
Since *rec* corresponds to cn (*rec* corresponds to cn' and $(cn', cn) \in inhr$), *rec* contains the bindings for all the fields of cn. rec' then contains the bindings for all the fields of cn together with “self” and “super”. These are exactly what are contained in the domain of π'. Therefore, rec' corresponds to π'. By the type of f, we have $[\![expr]\!]e \in [\![\tau']\!]e_\perp$, so $[\![expr]\!]e \in [\![\tau]\!]e_\perp$. ∎

4.3.2 Methods

Lemma 4.3 Let $k = ((\pi, cs, ts), inhr)$ be a good type context and let e be a type environment. Let m be a method with name mn. Then $[\![m]\!]e \in [\![Api\ \pi_0]\!]e$ if the following are satisfied:
(a) $k \vdash$ *m:* $Api\ \pi_0$, where

$\pi_0 = \{(mn,\ Ameth\ (Product\ [t_1, \cdots, t_n] :\rightarrow t, Api\ \pi)\}$
for some types t_j's, $1 \leq j \leq n$

(b) e is consistent with $(([], cs, ts), inhr)$.

Proof: Let $k = ((\pi, cs, ts), inhr)$ be a type context. Let $e = ((E, CE, TE), i)$ be a type environment that is consistent with $k_0 = (([], cs, ts), inhr)$.
Suppose $m = Method\ mn\ [i_1 :< t_1, \cdots, i_n :< t_n]\ t\ expr$.
Assume that $k \vdash m : Api\ \pi_0$, where

$\pi_0 = \{(mn,\ Ameth\ (Product\ [t_1, \cdots, t_n] :\rightarrow t, Api\ \pi)\}$.

Then we need to show that $[\![m]\!]e \in [\![Api\ \pi_0]\!]e$. By the type rule, $k' \vdash expr : Aexp\ t$, where $k' = ((\pi', cs, ts), inhr)$ and $\pi' = \pi \uplus \{i_j : t_j\}_{1 \leq j \leq n}$. By Lemma 4.2, for any type environment e' that is consistent with k', $[\![expr]\!]e' \in [\![t]\!]e'_\perp$. By the dynamic semantics, we have

$[\![m]\!]e = \{(mn, DVMD\ ([i_1 : t_1, \cdots, i_n : t_n], f))\}$ where
$f\ actuals\ env = [\![expr]\!]e''$,
$e'' = ((E_1, CE, TE), i)$,
$E_1 = E \uplus env'$ and
$env' = env \uplus actualsenv$ and
$actualsenv = zip\ [i_1, \cdots, i_n]\ actuals$.

Here, $F \uplus F' = \{(x, y) \mid (x, y) \in F'\ or\ [(x, y) \in F\ and\ x\ is\ not\ in\ dom(F')]\}$ for two finite functions F and F'. Now, we need to show that e'' is consistent with k'. We only need to worry about E_1. By the induction hypothesis, $actualsenv$ corresponds to $\{i_j : t_j\}_{1 \leq j \leq n}$ and env corresponds to π. Hence E_1 corresponds to π'. By the consistency of e with k, we see that e'' is consistent with k'. Also, since e'' differs from e only by their components of `Environment` and $expr$ is an expression, $[\![expr]\!]e' \in [\![t]\!]e_\perp$. Then we have the following: $f :: ([\![t_1]\!]e \times \cdots \times [\![t_n]\!]e) \rightarrow (E'{:}Environment) \rightarrow [\![t]\!]e$ where E' corresponds to π, i.e., we have the result. ∎

4.3.3 Declarations

Lemma 4.4 Let d be a declaration. Let $k = (([], cs, ts), inhr)$ be a good type context and let $e = ((E, CE, TE), i)$ be a type environment. If e is consistent with k and $k \vdash d : Adec\ (Acon\ k_0)$, then $[\![d]\!]e \in [\![Adec\ (Acon\ k_0)]\!]e$.

Proof: Let $k = (([], cs, ts), inhr)$ be a good type context and let $e = ((E, CE, TE), i)$ be a type environment which is consistent with k. We use $fieldEnv(fields)$ to denote the type assignment generated by $fields$. We use $name(m)$ to denote the name of the method m. We use $getFieldTypes(fields)$

to denote the types extracted from the *fields*.
1. Suppose $d = Class\ cn\ [sid]\ fields\ [m_1, \cdots, m_r]$:
Suppose that $k \vdash d{:}Adec\ (Acon\ k_0)$ where
$k_0 = (([], [((cn, (Acls\ (fields, ms, sid)))], []), (cn, sid)))$ for some ms (will be described later). We need to show that $[\![d]\!]e \in [\![Adec\ (Acon\ k_0)]\!]e$. By the type rule, we have the following:

(1) $(sid, (Acls\ (sfields, sms, ssid)) \in cs$

(2) there exist types s_j for $1 \leq j \leq r$ and for each of these, there exist $s_{j_1}, \cdots, s_{j_{l_j}}$ such that

$k' \vdash m_j : Api[(name(m_j), Ameth\ (Product\ [s_{j_1}, \cdots, s_{j_{l_j}}] :\longrightarrow s_j, Api\ \pi))]$,
where

$k' = ((\pi, cs, ts), inhr)$
$\pi = flds \bigcup \pi_0$
$\pi_0 = \{(\text{“}self\text{”},\ Aname\ cn), (\text{“}super\text{”},\ Aname\ sid)\}$
$flds = (fieldEnv\ sfields) \uplus (fieldEnv\ fields)$

(3) For any m_j such that a method named $name(m_j)$ is also defined in class sid, then m_j must have a subtype of the type of the method in the super class sid.

Now, ms can be given as
$ms = \bigcup_{1 \leq j \leq r} \{(name(m_j), Ameth\ (Product\ [s_{j_1}, \cdots, s_{j_{l_j}}] :\longrightarrow s_j, Api\ \pi)\}$.
Using Lemma 4.3, by the consistency of e with k,
$[\![m_j]\!]e \in \{DVMD\ ([i_1 : t_1, \cdots, i_n : t_n], f) \mid f :: ([\![s_{j_1}]\!]e \times \cdots \times [\![s_{j_{l_j}}]\!]e)$
$\rightarrow (E' : Environment) \rightarrow [\![s_j]\!]e, where\ E' corresponds\ to\ \pi\}$
Therefore, by definition of the dynamic semantics,
$[\![d]\!]e = (([], [(cn, DVCI\ (sid, fields, \bigcup_{1 \leq j \leq r} md_j)], []), (cn, sid))$
where $md_j = [\![m_j]\!]e$.
Comparing the corresponding components of $[\![d]\!]e$ and k_0 and using the fact that $DVCI\ (sid, fields, \bigcup_{1 \leq j \leq r} md_j) = [\![Acls\ (fields, ms, sid)]\!]e$, we see that $[\![d]\!]e \in [\![Adec\ (Acon\ k_0)]\!]e$.

2. Suppose $d = TupleClass\ tfields\ [m_1, \cdots, m_r]$.
Suppose that $k \vdash d{:}Adec\ (Acon\ k_0)$ where
$k_0 = (([], [], [(Product\ (getFieldTypes\ tfields), (Atup\ (tfields, ms))]), [])$ for some ms. Then we need to show that $[\![d]\!]e \in [\![Adec\ (Acon\ k_0)]\!]e$. By the type rule, there exist types s_j for $1 \leq j \leq r$ and for each of these, there exist $s_{j_1}, \cdots, s_{j_{l_j}}$ such that

$k' \vdash m_j : Api[(name(m_j), Ameth\ (Product\ [s_{j_1}, \cdots, s_{j_{l_j}}] :\longrightarrow s_j, Api\ \pi))]$,
where

$k' = ((\pi, cs, ts), inhr)$

$\pi = fieldEnv\ tfields$

Now, ms can be given as

$ms = \bigcup_{1 \le j \le r} \{(name(m_j), Ameth\ (Product\ [s_{j_1}, \cdots, s_{j_{l_j}}] :\longrightarrow s_j, Api\ \pi)\}$.

Using Lemma 4.3, by the consistency of e with k,

$[\![m_j]\!]e \in \{DVMD\ ([i_1 : t_1, \cdots, i_n : t_n], f) \mid f :: ([\![s_{j_1}]\!]e \times \cdots \times [\![s_{j_{l_j}}]\!]e)$
$\rightarrow (E' : Environment) \rightarrow [\![s_j]\!]e, where\ E' corresponds\ to\ \pi\}$

By definition of the dynamic semantics,

$[\![d]\!]e = (([], [], [(ClassIdTuple\ (getFieldTypes\ tfields),$
$DVTI(fields, \bigcup_{1 \le j \le r} md_j))]), [])$ where $md_j = [\![m_j]\!]e$.

Using the fact that $DVTI(fields, \bigcup_{1 \le j \le r} md_j) = [\![Atup\ (tfields, ms)]\!]e$, we have $[\![d]\!]e \in [\![Adec\ (Acon\ k_0)]\!]$. ∎

4.3.4 Recursive Declaration Sequences

The following lemma uses the auxiliary function $contextShadow$:

$((\pi, cs, ts), inhr)$ ‘$contextShadow$‘ $((\pi', cs', ts'), inhr') =$
$((\pi \uplus \pi', cs \uplus cs', ts \bigcup ts'), reflexiveTransiveClosure(inhr \bigcup inhr')$.

Lemma 4.5 Let rds be a recursive declaration sequence. Let e_0 be a type environment. Let $k_0 = (([], cs, ts), inhr)$ be a good type context. If e_0 is consistent with k_0 and $k_0 \vdash rds{:}Adec\ (Acon\ k)$, then k_0‘$contextShadow$‘ k is a good type context and $[\![rds]\!]e_0 \in [\![Adec\ (Acon\ k)]\!]e_0$.

Proof: Let $k_0 = (([], cs, ts), inhr)$ be a good type context and let $e_0 = ((E, CE, TE), i)$ be a type environment which is consistent with k_0. We use the following additional auxiliary functions for processing type contexts: $getContext$, $implementationsSideCheck$, and $consolidateContext$. We use auxiliary functions $consolidate$, and $getDEnv$ for processing type environments. For details, please refer to the Appendix.

Suppose $rds = RDS\ [d_1, \cdots, d_n]$.

Suppose $k_0 \vdash RDS\ [d_1, \cdots, d_n] : Adec\ (Acon\ k)$ for some type context k, then we need to show that $[\![rds]\!]e_0 \in [\![Adec\ (Acon\ k)]\!]e_0$. By the type rule, $k^1 \vdash d_j : Adec\ (Acon\ k_j)$ for $1 \le j \le n$ where

$implementationSideCheck\ k^1 = true$

$k^1 = k_0$‘$contextShadow$‘ k

$k = consolidateContext\ k'$

$k' = getContext\ [d_1, \cdots, d_n]$

By definition of the dynamic semantics, we have
$[\![rds]\!]e_0 = (E_1, CE_2, TE_1), i_1)$ where
$\quad CE_2 = consolidate\ (CE \uplus CE_1)\ i_1$
$\quad (E_1, CE_1, TE_1), i_1) =$
$\qquad getDEnv\ [d_1, \cdots, d_n]\ (E \uplus E_1, CE \uplus CE_1, TE \bigcup TE_1), i)$
Now, it remains to argue that $[\![rds]\!]e_0$ is consistent with k. By Lemma 4.4, we have $\forall j,\ 1 \le j \le n,\ [\![d_j]\!]e_0 \in [\![Adec\ (Acon\ k_j)]\!]e_0$. Since *getContext* and *getDEnv* just accumulate the miniature type contexts and type environments created by each d_j, $1 \le j \le n$ and they are consistency-preserving, $(E_1, CE_1, TE_1), i_1)$ is consistent with k'. Moreover, the operations *consolidate* and *consolidateContext* also preserve the consistency since they essentially do the same processing on each corresponding component, so we see that $[\![rds]\!]e_0$ is consistent with k. Finally, the checking phase on k^1 *implementationSideCheck* ensures that there will be no ambiguous tuple methods, so k^1 is a good type context. Hence we have the result. ∎

4.3.5 Programs

Lemma 4.6 Let *prog* be a program. Let $k_0 = (([], cs, ts), inhr)$ be a good type context and let $e_0 = ((E, CE, TE), i)$ be a type environment such that the followings are true:
$\quad$(1) $(top, Acls\ ([], [], top)) \in cs$ and
$\quad$(2) $(top, DVCI\ (top, [], [])) \in CE$.
Then if e_0 is consistent with k_0 and $k_0 \vdash prog{:}Aexp\ \tau$, then $[\![prog]\!]e_0 \in [\![Aexp\ \tau]\!]e_0$.
Proof:
Let $k_0 = (([], cs, ts), inhr)$ be a type context and let $e_0 = ((E, CE, TE), i)$ be a type environment such that the conditions (1) and (2) are true. We use an auxiliary function *isPrimitive* to test if a type is primitive or not. It can be described by the following algorithm:

```
> isPrimitive :: Type -> Bool
> isPrimitive Aint = True
> isPrimitive Abool = True
> isPrimitive (Aname t) = (t == top)
> isPrimitive (Product ts) = all isPrimitive ts
> isPrimitive _ = False
```

Suppose $prog = rds$ *'In'* $expr$.
Assume that e_0 is consistent with k_0 and $k_0 \vdash rds$ *'In'* $expr : Aexp\ \tau$ for some τ. Then we need to show that $[\![P]\!]e_0 \in [\![Aexp\ \tau]\!]$. By the type rule,

$k_0 \vdash rds : Adec(Acon\ k)$

$k_1 \vdash expr : Aexp\ \tau$, where $k_1 = k_0$ *'contextShadow'* k

$isPrimitive\ \tau = true$

By Lemma 4.5, $[\![rds]\!]e_0 = e$ and e is consistent with k, By definition of the dynamic semantics, $[\![prog]\!]e_0 = [\![expr]\!]e_1$, where

$e_1 = ((E \uplus E', CE \uplus CE', TE \bigcup TE'), reflexiveTransitiveClosure\ (i \bigcup i'))$

$((E', CE', TE'),\ i') = e$

By the structure of e_1 and k_1 and the fact that e and e_0 are consistent with k and k_0, we see that e_1 is consistent with k_1. So, by Lemma 4.2, $[\![prog]\!]e_0 = [\![expr]\!]e_1 \in [\![\tau]\!]e_{1_\perp}$. Since τ is primitive, it then follows that $[\![prog]\!]e_0 \in [\![\tau]\!]e_{0_\perp}$ ∎

Now we have the soundness theorem.

5 Conclusions and future work

In this section, we will give the future directions of research and will draw some conclusions about our work.

Tuple, as it now, is a small language whose main purpose is to illustrate one way of adding multiple dispatch to a single-dispatched language. Our prototype system allows us to experiment with the ideas. Future work in this area is to apply these ideas to languages like C++ or Java. We believe that this will provide programers better ways to approach problems such as the visitor and binary method problems, as described in the introduction. Furthermore, other applications of tuple expressions and multiple dispatch may be exploited. For example, multiple dispatch could be used to describe the semantics of obtaining multiple locks at the same time.

Our experience in programming in Tuple suggests that it is easy for programers to code and that Tuple programs have good readability. Since we separate the methods that need to be multiple dispatched from the ones that don't by coding them in tuple classes, the meaning of Tuple programs is easy to convey and the code is easy to maintain. Take the visitor example we gave in Section 1 as an example, the semantics of the tuple program is clear and it is easy to add new node or visitor classes.

The contribution of this work is that it gives a semantic prototype and a type checker for the language Tuple. Moreover, based on our prototype system, we have proved that our revised static type system of Tuple is sound with respect to the dynamic semantics. Besides that the dynamic semantics has been studied in more detail, several changes from the original Tuple paper [9] have been made during the process of soundness proof. They are as follows:

1. It turns out that the original type for methods does not contain enough information, so we have changed it from

$$\mathit{Product}\ [t_1, \cdots, t_n] :\rightarrow t$$

to

$$\mathit{Ameth}\ (\mathit{Product}\ [t_1, \cdots, t_n] :\rightarrow t, \mathit{Api}\ \pi)$$

where π contains the bindings of the run-time class or tuple class objects.

2. There should be one more check in type checking programs, namely, the type of the program's expression should be a primitive type.

3. The keyword `super` has been added to enable the corresponding super class method to be called.

4. Our approach to type checking recursive declaration sequences is different from the original one. That is, instead of type checking by the way of a fix-point, we first assume the correctness in the information gathering pass, then type check it in a second pass.

In addition, we find that a rigorous soundness proof of the type system not only gives us the result we want, but also helps in finding coding errors.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, second edition, 1998.

[2] G. Baumgartner, K. Läufer, and V. F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Department of Computer Science, Purdue University, Feb. 1998.

[3] J. Boyland and G. Castagna. An implementation of multi-methods for java. *ACM*, 1997. *OOPSLA '97 Proceedings*.

[4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[5] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *TOPLAS*, 17(6):805–843, Nov. 1995.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[7] M. J. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 1988.

[8] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. 21(11):347–349, Nov. 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[9] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. *ACM*, 1998. *OOPSLA '98 Proceedings*.

[10] D. A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing Series. MIT Press, Cambridge, Mass., 1994.

[11] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Harlow, England, 1996.

[12] J. D. Ullman. *Elements of ML Programming.* Prentice Hall, Englewood Cliffs, NJ, 1994.

A Appendix

Dynamic semantics for the Tuple.

AUTHOR: Jianbing Chen.

```
> module Semantics where
> import Parser
> import Environments
> import FinFun
> import Helpers
> import List
> import Tree
```

SEMANTIC FUNCTIONS

The semantic functions are meaningP, meaningRDS, meaningDS, meaningD, meaningMDS, meaningMD, meaningES, meaningE.

The presentation of them is top down. There are cases that we use Haskell function "error" to print out some error messages. This is because that we implemented the semantics first. In fact, all these erros will be caught in type checking, so they won't show up at run-time. On the other hand, this allows the semantics of Tuple to be used without its' type system, as a dynamically type language.

PROGRAMS

```
> meaningP :: Program -> EnvInh -> ExpressedValue
> meaningP (( rds `In` e )) (d_env, inh) =
>   meaningE ((e)) (d_env1, inh1)
>    where
>      (d_env1, inh1) = (DEnv (env `unionMinus` env2,
```

```
>         classEnv `unionMinus` classEnv2,
>         tupleEnv ++ tupleEnv2),
>         getInheritance (inh `union` inh2))
>      (DEnv (env2,classEnv2,tupleEnv2), inh2) =
>         meaningRDS((rds)) (d_env, inh)
>      (DEnv (env,classEnv,tupleEnv), _) = (d_env, inh)
```

RECURSIVE_DECLARATION_SEQUENCES

```
> meaningRDS :: RDS -> EnvInh -> EnvInh
> meaningRDS ((RDS rds)) (DEnv (env,cEnv,tEnv),inh) =
>    let
>       getDEnv = meaningDS
>       (DEnv (newEnv,newClassEnv,newTupleEnv),inh1) =
>          getDEnv rds (DEnv (env `unionMinus` newEnv,
>            cEnv `unionMinus` cEnv1,
>            tEnv ++ newTupleEnv), inh)
>       cEnv1 =
>          consolidate (cEnv `unionMinus` newClassEnv) inh1
>    in
>       (DEnv (newEnv, cEnv1, newTupleEnv), inh1)
```

DECLARATION LISTS

```
> meaningDS :: [Declaration] -> EnvInh -> EnvInh
> meaningDS (( [] )) (d_env, inh) = (emptyDEnv, [])
> meaningDS (( d:ds )) (d_env, inh) =
>   let
>     (DEnv d_env1, inh1) = meaningD (( d )) (d_env,inh)
>     (DEnv d_env2, inh2) = meaningDS (( ds )) (d_env,inh)
>     d_env3 = case (d_env1 `envUnion` d_env2) of
>          Nothing -> error "meaningDS: dup declarations"
>          Just d_env3 -> DEnv d_env3
>     inh3 =  (inh1 `union` inh2)
>   in
```

```
>     (d_env3, inh3)
```

DECLARATIONS

```
> meaningD :: Declaration -> EnvInh -> EnvInh
> meaningD ((Class cid [] flds meths)) (d_env,inh) =
>   meaningD ((Class cid [top] flds meths)) (d_env,inh)

> meaningD ((Class cid [sid] flds ms)) (d_env,inh) =
>   let
>       flds1 = map (\(i :< t) -> (i, t)) flds
>       dict1 = meaningMDS ms (d_env, inh)
>   in
>       (DEnv (emptyEnv, [(cid, DVCI (sid,flds1,dict1))],
>          emptyTupleEnv),  [(cid, sid)])

> meaningD (( TupleClass flds ms )) (d_env, inh) =
>   let
>      tupleid =
>        ClassIdTuple (map (\(i :< (Aname t)) ->
>                          (ClassIdSingle t)) flds)
>      tupleformals = map (\(i :< t) -> (i, t)) flds
>      dict = meaningMDS ms (d_env, inh)
>   in
>      (DEnv (emptyEnv, emptyClassEnv,
>            [(tupleid, DVTI (tupleformals, dict))]), [])
```

METHOD LISTS

```
> meaningMDS :: [Method] -> EnvInh -> DVMethDict
> meaningMDS (( [] )) (d_env, inh) = []
> meaningMDS (( m:ms )) (d_env, inh) =
>    let
>       dict = meaningMD m (d_env, inh)
>       dict1 = meaningMDS ms (d_env, inh)
```

```
>    in
>       case (dict `unionDot` dict1) of
>         Just dict2  ->  dict2
>         Nothing ->  error "meaningMDS: dup meth names"
```

METHODS

```
> meaningMD :: Method -> EnvInh -> DVMethDict
> meaningMD ((Method m ps rt expr)) (DEnv d_env,inh) =
>       [(m, DVMD (formals,f))]
>   where
>      formals = map (\(i :< t) -> (i, t)) ps
>      f actuals objenv = meaningE ((expr)) (d_env1,inh)
>        where
>           d_env1 =
>              DEnv ((tripleFst d_env) `unionMinus` env1,
>              tripleSnd d_env, tripleTrd d_env)
>           env1 = objenv `unionMinus` actualsenv
>           actualsenv =
>       zip (map (\(i,t) -> i) formals) actuals
```

EXPRESSION LISTS

```
> meaningES :: [Expression] -> EnvInh -> [ExpressedValue]
> meaningES (( exprs )) (d_env, inh) =
>    map (\expr -> meaningE expr (d_env, inh)) exprs
```

EXPRESSIONS

```
> meaningE :: Expression -> EnvInh -> ExpressedValue
> meaningE (( New cid exprs )) (DEnv d_env, inh) =
```

```
>  let
>     d_values = meaningES exprs (DEnv d_env, inh)
>  in
>     case (lookup cid (tripleSnd d_env)) of
>       Just (DVCI (super_cid, fds, dict)) ->
>            DVObject cid (zip (dom fds) d_values)
>       Nothing -> error ("No such class! " ++ cid)

> meaningE ((Dot (Tuple es) m exprs)) (DEnv d_env, inh) =
>  let
>    obj = meaningE (Tuple es) (DEnv d_env, inh)
>    actuals_dvs = meaningES exprs (DEnv d_env, inh)
>  in
>    case (obj) of
>      DVTuple dvs ->
>        let
>          tupleType =
>             map (\(DVObject acid ard) -> acid) dvs
>          validtype (ClassIdTuple cids, DVTI (tfs,dict))
>            = and [length cids == length dvs,
>                   md `elem` (dom dict),
>                   subtypes tupleType
>                     (map (\(ClassIdSingle tcid)->tcid)
>                       cids) inh]
>          candidateTuples =
>             filter validtype (tripleTrd d_env)
>          tupleTypes =
>             map (map (\(ClassIdSingle cid) -> cid))
>              (map (\(ClassIdTuple ttype,_) -> ttype)
>                candidateTuples)
>          sortedTypes = typesqueeze tupleTypes inh []
>          DVTI (targetFls, targetDict) =
>           case (length sortedTypes) of
>            1         ->
>             findTuple (sortedTypes!!0) candidateTuples
>            0         ->
>             error "tuple not implemented"
>            otherwise ->
```

```
>             error "ambiguous tuple implementation"
>          Just (DVMD (fl,f)) = lookup d targetDict
>          rec = zip (dom targetFls) dvs
>        in
>          f actuals_dvs rec
>      _     -> error "invalid message receiver!"

> meaningE ((Dot expr m exprs)) (DEnv d_env, inh) =
>  let
>    obj = meaningE expr (DEnv d_env, inh)
>    actuals_dvs = meaningES exprs (DEnv d_env, inh)
>  in
>    case (obj) of
>      DVObject cid rec ->
>        let
>          Just (DVCI (scid, fds, mdict)) =
>              (lookup cid (tripleSnd d_env))
>          rec1 = rec `union` [("self", obj)]
>          rec2 = rec1 `union`
>              [("super", (DVObject scid rec))]
>        in
>          case (lookup d mdict) of
>            Just (DVMD (fl, f)) ->
>              f actuals_dvs rec2
>            _                  ->
>              error "method app error"
>      _     -> error "invalid message receiver!"

> meaningE (( Tuple exprs )) (d_env, inh) =
>   DVTuple (meaningES exprs (d_env, inh))

> meaningE (( Ident iden )) (DEnv d_env, inh) =
>   case (lookup iden (tripleFst d_env)) of
>       Just dv -> dv
>       Nothing -> error (iden ++ " not defined")

> meaningE (( e1 `Plus` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
```

```
>       (DVInt i1, DVInt i2) -> DVInt (i1 + i2)
>       otherwise            -> error "Plus error"

> meaningE (( e1 `Minus` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env, inh),
>            meaningE e2 (d_env, inh)) of
>       (DVInt i1, DVInt i2) -> DVInt (i1 - i2)
>       otherwise            -> error "Plus error"

> meaningE (( e1 `Times` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVInt i1, DVInt i2) -> DVInt (i1 * i2)
>       otherwise            -> error "Plus error"

> meaningE (( e1 `Lt` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVInt i1, DVInt i2) -> DVBool (i1 < i2)
>       otherwise            -> error "Plus error"

> meaningE (( e1 `Gt` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVInt i1, DVInt i2) -> DVBool (i1 > i2)
>       otherwise            -> error "Plus error"

> meaningE (( e1 `Gt` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVInt i1, DVInt i2) -> DVBool (i1 > i2)
>       otherwise            -> error "Plus error"

> meaningE (( e1 `Le` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVInt i1, DVInt i2) -> DVBool (i1 <= i2)
>       otherwise            -> error "Plus error"

> meaningE (( e1 `Ge` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVInt i1, DVInt i2) -> DVBool (i1 >= i2)
>       otherwise            -> error "Plus type error"
```

```
> meaningE (( Not expr )) (d_env, inh) =
>   case (meaningE expr (d_env, inh)) of
>       DVBool b  -> DVBool (not b)
>       otherwise -> error "Not type error"

> meaningE (( e1 `Equals` e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVInt i1, DVInt i2)   -> DVBool (i1 == i2)
>       (DVBool b1, DVBool b2) -> DVBool (b1 == b2)
>       otherwise              -> error "Equals type error"

> meaningE (( Num n )) (d_env, inh) = DVInt n

> meaningE (( BoolLit b )) (d_env, inh) = DVBool b

> meaningE (( And e1 e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVBool b1, DVBool b2) -> DVBool (and [b1, b2])
>       otherwise              -> error "Equals type error"

> meaningE (( Or e1 e2 )) (d_env, inh) =
>   case (meaningE e1 (d_env,inh),meaningE e2 (d_env,inh)) of
>       (DVBool b1, DVBool b2) -> DVBool (or [b1, b2])
>        otherwise             -> error "Equals type error"

> meaningE (( If e1 e2 e3 )) (d_env, inh) =
>   case (meaningE e1 (d_env, inh)) of
>       DVBool b -> if b
>                   then meaningE e2 (d_env, inh)
>                   else meaningE e3 (d_env, inh)
>       otherwise -> error "If type error"
```

SOME AUXILIARY FUNCTIONS

```
> consolidate :: CEnvironment -> InheritanceRel
>                             -> CEnvironment
```

```
> consolidate classEnv inh =
>   let
>     typeTree = buildTypeTree top inh
>     sortedlist = (traverse typeTree) \\ [top]
>   in
>     updateClassEnv classEnv sortedlist
>   where
>     updateClassEnv cEnv [] = cEnv
>     updateClassEnv cEnv (c:cs) = updateClassEnv
>       (update cEnv c (DVCI (sid,flds,dict))) cs
>         where
>           Just (DVCI (sid, newflds, newdict)) =
>               lookup c cEnv
>           Just flds = (sflds `unionDot2` newflds)
>           dict = (sdict `unionMinus` newdict)
>           (sflds, sdict) = case (lookup sid cEnv) of
>             Just (DVCI (_, sflds, sdict)) -> (sflds, sdict)
>             Nothing -> error (sid ++ " undefined in "
>                                   ++ (show cEnv))

> buildTypeTree :: ClassId -> InheritanceRel -> Tree ClassId
> buildTypeTree cid inh =
>   Node cid (map (\id -> buildTypeTree id inh) subtrees)
>      where
>        subtrees = [ cls | (cls, super_cls) <- inh,
>                           super_cls == cid]
```

```
Type checking for the language Tuple.
Based on the paper of Leavens and Millstein:
"Multiple Dispatch as Dispatch on Tuples".

AUTHOR: Jianbing Chen & Gary T. Leavens.

> module Typing where
> import Parser
> import Unparser
> import TypeAttributes
> import TypeHelpers
> import Subst
> import FinFun
> import List
```

SYNTAX DOMAIN

```
To allow heterogenity of types of syntax in the lists
of hypotheses, we use the data type Syntax defined
below.

> data Syntax =
>           Prog Program
>         | Rds RecursiveDeclarationSequence
>         | Decl Declaration
>         | Meth Method
>         | Expr Expression
>         | SynList [Syntax]
>         | DeclList [Declaration]
>         | MethList [Method]

we make an instance for the type Syntax also.

> instance Annotatable Syntax where
>   annotate k (Prog p) =
```

```
>         case annotate k p of
>              (p:::a) -> Prog p ::: a
>   annotate k (Rds rds) =
>         case annotate k rds of
>              (rds:::a) -> Rds rds ::: a
>   annotate k (Decl d) =
>         case annotate k d of
>              (d:::a) -> Decl d ::: a
>   annotate k (Meth m) =
>         case annotate k m of
>              (m:::a) -> Meth m ::: a
>   annotate k (Expr e) =
>         case annotate k e of
>              (e:::a) -> Expr e ::: a
>   annotate k (SynList []) =
>     axiom (k :- SynList [] ::: Product [])
>   annotate k (SynList (x:xs)) =
>     ruleIf [(k :- x ::: Avar tau),
>             (k :- (SynList xs) ::: Avar theta)]
>             (\_ -> True)
>             (\[t1, Product t2] ->
>                k1 ->- Product (t1:t2))
>             -----------------------------------
>             (k :- (SynList (x:xs)) ::: Avar k1)
>   annotate k (DeclList []) =
>     axiom (k :- DeclList [] :::
>                Adec (Acon (([],[],[]),[])))
>   annotate k (DeclList (d:ds)) =
>     ruleIf [(k :- (Decl d) ::: Avar tau),
>             (k :- (DeclList ds) ::: Avar theta)]
>             (\[Adec (Acon ((_,cs1,_),_)),
>                Adec (Acon ((_,cs2,_),_))]
>               -> disjoint cs1 cs2)
>             (\[Adec (Acon k1),Adec (Acon k2)] ->
>               k3 ->- Adec (Acon (k1 `contextDot` k2)))
>             ------------------------------------------
>             (k :- (DeclList (d:ds)) ::: Avar k3)
>   annotate k (MethList []) =
```

```
>     axiom (k :- MethList [] ::: Api [])
>   annotate k (MethList (m:ms)) =
>     ruleIf [(k :- (Meth m) ::: Avar pi1),
>             (k :- (MethList ms) ::: Avar pis)]
>             (\[Api dict1, Api dict2] ->
>                disjoint dict1 dict2)
>             (\[Api dict1, Api dict2] ->
>                let
>                  Just dict = (dict1 `unionDot` dict2)
>                in
>                  k3 ->- Api dict)
>             -----------------------------------------
>             (k :- (MethList (m:ms)) ::: Avar k3)
```

To help in printing error messages, it is useful to make Syntax also an instance of the class Show.

```
> instance Show Syntax where
>   showsPrec p (Prog pr) = showsPrec p pr
>   showsPrec p (Rds rds) = showsPrec p rds
>   showsPrec p (Decl d) = showsPrec p d
>   showsPrec p (Meth m) = showsPrec p m
>   showsPrec p (Expr e) = showsPrec p e
>   showsPrec p (SynList ls) = showList ls
>   showsPrec p (DeclList ds) = showList ds
>   showsPrec p (MethList ms) = showList ms
```

Declaration of the initial type context.

```
> initialTypeContext :: TypeContext
> initialTypeContext =
>    (([],[(top, Acls ([],[],top))],[]),[])
```

Declarations of logical variables.

```
> tau = (0, "tau")
```

```
> tau1 = (1, "tau")
> tau2 = (2, "tau")
> theta = (0, "theta")
> pi1 = (1, "pi")
> pi2 = (2, "pi")
> pis = (0, "pis")
> k1 = (1, "k")
> k2 = (2, "k")
> k3 = (3, "k")
```

AUXILIARY FUNCTIONS FOR TYPING

```
> contextShadow :: TypeContext -> TypeContext -> TypeContext
> ((pi1,cs1,ts1),i1) `contextShadow` ((pi2,cs2,ts2),i2) =
>    updateInheritance ((pi1 `unionMinus` pi2,
>         cs1 `unionMinus` cs2, ts1 `union` ts2), [])

> contextDot :: TypeContext -> TypeContext -> TypeContext
> ((pi1,cs1,ts1),i1) `contextDot` ((pi2,cs2,ts2),i2) =
>   case (pi1 `unionDot` pi2, cs1 `unionDot` cs2,
>         ts1 `union` ts2) of
>      (Just pi3, Just cs3, ts3)
>         -> ((pi3,cs3,ts3), (inh1 `union` inh2))
>      otherwise
>         -> error "duplicate declaration"

> field_types :: [IdentifierTyping] -> [Type]
> field_types its = map (\(_:<tau) -> tau) its

> field_env :: [IdentifierTyping] ->
>              FiniteFun Identifier Type
> field_env its = toFun (map (\(i:<t) -> (i,t)) its)
```

TYPING RULES FOR PROGRAMS

```
> instance Annotatable Program where
>  annotate k p@(rds `In` e) =
>   ruleSeq
>     [Hyp (k :- Rds rds ::: Adec (Avar k1)),
>      Def (\[Adec (Acon k1)] -> (k2 ->- Acon
>           (k `contextShadow` k1))),
>      Hyp (k2 :?- Expr e ::: Aexp (Avar tau)),
>      When (\[_, Aexp t] -> isPrimitive t)]
>     -----------------------------------------
>     (k :- p ::: Aexp (Avar tau))
```

RECURSIVE DECLARATION SEQUENCE

```
> instance Annotatable RecursiveDeclarationSequence where
>  annotate k rds@(RDS ds) =
>    ruleIf [(k2 :- DeclList ds ::: Avar tau)]
>            (\_ -> implementationSideCheck k2)
>            (\_ -> nullSubst)
>           ----------------------------------
>           (k :- rds ::: Adec (Acon k1))
>    where
>            k2 = k `contextShadow` k1
>            k1 = consolidateContext k0 k
>            k0 = foldl1 contextDot
>                   (map (\d -> getDeclContext d) ds)
```

DECLARATIONS

```
> instance Annotatable Declaration where
>  annotate k d@(TupleClass its ms) =
>    ruleIf [(k' :- MethList ms ::: Avar pi1)]
>            (\_ -> True)
>            (\[(Api pi1)] -> k1 ->-  Acon (([],[],
>                [(Product (field_types its),
>                  Atup (its,pi1))]),[]))
>    --------------------------------------
```

```
>     (k :- d ::: Adec (Avar k1))
>    where
>            k' = ((pi',cs,ts),inh)
>            pi' = pi2
>            pi2 = pi `unionMinus` (field_env its)
>            ((pi,cs,ts),inh) = k

>  annotate k d@(Class i [] its ms) =
>    annotate k (Class i [top] its ms)

>  annotate k d@(Class i [sid] its ms) =
>    ruleIf [(k' :- MethList ms ::: Avar pis)]
>            (\[(Api mdict)] -> (sid `elem` (dom cs))
>               && (checkClassMethDict mdict sid k'))
>            (\[(Api mdict)] ->  k1 ->- Acon (([],
>                     [(i,Acls (its,mdict,sid))],[]),
>                     [(Aname i,Aname sid)]))
>     ---------------------------------------
>            (k :- d ::: Adec (Avar k1))
>    where
>             k' = ((pi2,cs,ts),inh)
>             pi2 = pi1 `union` pi0
>             pi1 = pi `unionMinus` fields
>             pi0 = [("self", Aname i),
>                     ("super", Aname sid)]
>             fields =
>               case ((field_env its) `unionDot`
>                    (getFldsEnv sid k)) of
>                  (Just flds) -> flds
>                  Nothing -> error "duplicate fields"
>            ((pi,cs,ts),inh) = k
```

METHODS

```
> instance Annotatable Method where
>  annotate k m@(Method mn its t e) =
>    ruleIf [k' :- e ::: Aexp t]
```

```
>            (\[Aexp t'] -> subTypes t' t k)
>            (\_ -> pi1 ->- (Api [(mn, Ameth
>               ((Product (field_types its) :-> t),
>                Api pi))]))
>           --------------------------------------
>           (k :- m ::: (Avar pi1))
>    where
>            k' = ((pi',cs,ts),inh)
>            pi' = pi `unionMinus` (field_env its)
>           ((pi,cs,ts),inh) = k
```

EXPRESSIONS

```
> justApply pi i =
>   case (lookup i pi) of
>     (Just theta) -> theta
>     Nothing -> Aerr

> extractExListType :: Type -> Type
> extractExListType (Product ts) =
>    Product (map (\(Aexp t) -> t) ts)


> instance Annotatable Expression where
>  annotate k e@(New cn es) =
>    ruleIf [k :- SynList (map Expr es) ::: Avar tau]
>            (\[ts] -> (cn `elem` (dom cEnv)) &&
>               (subTypes (extractExListType ts)
>                               formalT k))
>            (\_ -> nullSubst)
>           -----------------------------------------
>            (k :- e ::: Aexp (Aname cn))
>    where
>            ((_,cEnv,_),_) = k
>            formalT = Product (getFldsType cn k)
```

```
> annotate k e@(Dot (Tuple es) md es1) =
>   ruleIf [k :- SynList (map Expr es) ::: Avar tau1,
>           k :- SynList (map Expr es1) ::: Avar tau2]
>           (\[esT, exprsT] -> case (esT, exprsT) of
>             (Product _, Product _) -> hasMostSpecific
>                       ((extractExListType esT),mn)
>                       (extractExListType exprsT) k)
>           (\[esT, es1T] -> case (esT, es1T) of
>             (Product _, Product _)  ->
>                  tau ->- (Aexp return)
>                where
>                    Ameth (_:->return,_) = getMdType
>                      ((extractExListType esT), md)
>                      (extractExListType exprsT) k)
>          ------------------------------------------
>           (k :- e ::: (Avar tau))

> annotate k e@(Dot expr md es) =
>   ruleIf [k :- Expr expr ::: Avar tau,
>           k :- SynList (map Expr es) ::: Avar tau1]
>           (\[exprT, esT] -> case (exprT, esT) of
>               (Aexp (Aname cn), ts) ->
>                     (cn `elem` (dom cEnv)) &&
>                     (mn `elem` (dom (getMethDict cn k)))
>                     && (subTypes actualT para k)
>                  where
>                    (Ameth (para:->return,_)) =
>                     getMdType (Aname cn,md) actualT k
>                    actualT = extractExListType ts
>               (_,_)                 -> False)
>           (\[exprT, esT] -> case (exprT, esT) of
>               (Aexp (Aname cn), ts) ->
>                 tau2 ->- (Aexp return)
>                  where
>                    Ameth (_:->return,_) =
>                     getMdType (Aname cn,md) actualT k
>                    actualT = extractExListType ts)
>           ------------------------------------------
```

```
>            (k :- e ::: (Avar tau1))
>   where
>            ((_,cEnv,_),_) = k

> annotate k e@(Tuple es) =
>   rule [k :- SynList (map Expr es) ::: Avar tau]
>        ----------------------------------------
>        (k :- e ::: (Aexp (Avar tau)))

> annotate k e@(Ident id) =
>   ruleIf nohyp (\_ -> id `elem` (dom env))
>                (\_ -> (theta ->- justApply env id))
>          ------------------------------------------
>          (k :- e ::: Aexp (Avar theta))
>   where
>      nohyp :: [Judgement (AttribTree
>                     Expression PrimitiveAttrib)]
>      nohyp = []
>      ((env,_,_),_) = k

> annotate k e@(Num n) =
>   axiom (k :- e ::: (Aexp Aint))

> annotate k e@(BoolLit b) =
>   axiom (k :- e ::: (Aexp Abool))

> annotate k e@(e1 `Plus` e2) =
>   rule [k :- Expr e1 ::: Aexp Aint,
>         k :- Expr e2 ::: Aexp Aint]
>        ----------------------------
>        (k :- e ::: (Aexp Aint))

> annotate k e@(e1 `Minus` e2) =
>   rule [k :- Expr e1 ::: Aexp Aint,
>         k :- Expr e2 ::: Aexp Aint]
>        ----------------------------
>        (k :- e ::: (Aexp Aint))
```

```
> annotate k e@(e1 `Times` e2) =
>   rule [k :- Expr e1 ::: Aexp Aint,
>         k :- Expr e2 ::: Aexp Aint]
>        ----------------------------
>        (k :- e ::: (Aexp Aint))

> annotate k e@(e1 `Lt` e2) =
>   rule [k :- Expr e1 ::: Aexp Aint,
>         k :- Expr e2 ::: Aexp Aint]
>        ----------------------------
>        (k :- e ::: (Aexp Abool))

> annotate k e@(e1 `Gt` e2) =
>   rule [k :- Expr e1 ::: Aexp Aint,
>         k :- Expr e2 ::: Aexp Aint]
>        ----------------------------
>        (k :- e ::: (Aexp Abool))

> annotate k e@(e1 `Le` e2) =
>   rule [k :- Expr e1 ::: Aexp Aint,
>         k :- Expr e2 ::: Aexp Aint]
>        ----------------------------
>        (k :- e ::: (Aexp Abool))

> annotate k e@(e1 `Ge` e2) =
>   rule [k :- Expr e1 ::: Aexp Aint,
>         k :- Expr e2 ::: Aexp Aint]
>        ----------------------------
>        (k :- e ::: (Aexp Abool))

> annotate k e@(Not e1) =
>   rule [k :- Expr e1 ::: Aexp Abool]
>        -----------------------------
>        (k :- e ::: (Aexp Abool))

> annotate k e@(e1 `Equals` e2) =
>    (rule [k :- e1 ::: Aexp Aint,
>           k :- e2 ::: Aexp Aint]
```

```
>           -------------------------
>           (k :- e ::: (Aexp Abool)))
>   `ruleOr`
>     (rule [k :- e1 ::: Aexp Abool,
>            k :- e2 ::: Aexp Abool]
>           -------------------------
>           (k :- e ::: (Aexp Abool)))

>  annotate k e@(e1 `And` e2) =
>    rule [k :- Expr e1 ::: Aexp Abool,
>          k :- Expr e2 ::: Aexp Abool]
>         ----------------------------
>         (k :- e ::: (Aexp Abool))

>  annotate k e@(e1 `Or` e2) =
>    rule [k :- Expr e1 ::: Aexp Abool,
>          k :- Expr e2 ::: Aexp Abool]
>         ----------------------------
>         (k :- e ::: (Aexp Abool))

>  annotate k e@(If e1 e2 e3) =
>    ruleIf [k :- Expr e1 ::: Aexp Abool,
>     k :- Expr e2 ::: Aexp (Avar tau1),
>     k :- Expr e3 ::: Aexp (Avar tau2)]
>            (\[t1,t2,t3] -> (t2 == t3))
>            (\_  -> nullSubst)
>           ------------------------------
>           (k :- e ::: (Aexp (Avar tau1)))
```

Helper functions for TypeContext, etc

```
> updateInheritance :: TypeContext -> TypeContext
> updateInheritance ((pi, cs, ts), inh) =
>   let
>      directInh = map (\(cid, Acls (its,mds,scid))
```

```
>           -> (Aname cid, Aname scid)) cs
>      inh1 = (reflexiveTransitiveClosure directInh)
>          \\ [(Aname top, Aname top)]
>   in
>      if (cyclic
>          (inh1 \\ map (\a -> (a,a)) (dom inh1)))
>        then error ("Classes def in cycle")
>        else ((pi, cs, ts), inh1)

> getMethDict :: ClassId -> TypeContext -> TypeAssignment
> getMethDict id ((_,cs,_),_) =
>    case (lookup id cs) of
>      Just (Acls (_, mdict, _)) -> mdict
>      Nothing -> error ("class "++id++" undefined")

> getFldsEnv :: ClassId -> TypeContext -> TypeAssignment
> getFldsEnv cn ((_,cs,_),_) =
>    case (lookup cn cs) of
>      Just (Acls (its, mds, scid))
>         -> map (\(i :< t) -> (i,t)) its
>      _
>         -> error ("class "++cn++" undefined")

> getFldsType :: ClassName -> TypeContext -> [Type]
> getFldsType cn ((pi, cs, ts), inh) =
>    case (lookup cn cs) of
>      Just (Acls (its, mds, scid))
>         -> map (\(i :< t) -> t) its
>      _
>         -> error ("class "++cn++" undefined")

> hasMostSpecific :: (Type, MethodName) -> Type
>                    -> TypeContext -> Bool
> hasMostSpecific (tfls, md) actualstype
>           ((pi, cs, ts), inh) =
>  case (tfls) of
>   (Product tts) ->
>    let
```

```
>      hasMethod mn dict =
>        if (mn `elem` (dom dict))
>             then let
>                    Just (Ameth (type1:->rtype,_))
>                      = lookup mn dict
>                  in
>                    subTypes actualstype type1
>                      ((pi,cs,ts),inh)
>             else False
>      validType (tfls1, Atup (ifs, mdict)) =
>         (hasMethod md mdict) &&
>         (subTypes tfls tfls1 ((pi,cs,ts), inh))
>      candidates = (filter validType ts)
>      sorted = tupleSqueeze (dom candidates)
>         ((pi,cs,ts),inh) []
>    in case (length sorted) of
>       1 -> True
>       _ -> False
>   _             -> False

> getMdType :: (Type, MethodName) -> Type
>               -> TypeContext -> Type
> getMdType (Aname cn, md) actualstype
>           ((pi, cs, ts), inh) =
>    let
>      Just (Acls (its,mds,scid)) = (lookup cn cs)
>    in
>      case (lookup md mds) of
>        Just t -> t
>        _      -> error (md++" not implemented")
> getMdType (tfls, md) actualstype
>           ((pi, cs, ts), inh) =
>    let
>      hasMethod mn dict =
>        if (mn `elem` (dom dict))
>             then let
>                    Just (Ameth (type1:->rtype,_))
>                      = lookup mn dict
```

```
>                 in
>                   subTypes actualstype type1
>                     ((pi,cs,ts),inh)
>            else False
>     validType (tfls1, Atup (ifs, mdict)) =
>        (hasMethod md mdict) &&
>        (subTypes tfls tfls1 ((pi,cs,ts), inh))
>     candidates = (filter validType ts)
>     sorted = tupleSqueeze (dom candidates)
>        ((pi,cs,ts),inh) []
>   in case (length sorted) of
>      1 -> let
>             Just (Atup (its,mds)) =
>                  (lookup (sorted!!0) candidates)
>             Just t = (lookup md mds)
>           in t
>      0 -> error "tuple not implemented"
>      _ -> error "ambiguous tuple implementation"

> tupleSqueeze :: [Type] -> TypeContext -> [Type] -> [Type]
> tupleSqueeze [] k ts = ts
> tupleSqueeze (c:cs) k ts =
>   if (or (map (\x -> subTypes x c k) ts))
>     then tupleSqueeze cs k ts
>     else
>        let
>          ts1 = [ x | x <- ts, subTypes c x k]
>        in
>          tupleSqueeze cs k ((ts \\ ts1) ++ [c])

> isPrimitive :: Type -> Bool
> isPrimitive Aint = True
> isPrimitive Abool = True
> isPrimitive (Aname t) = (t == top)
> isPrimitive (Product ts) = all isPrimitive ts
> isPrimitive _ = False

> subTypes :: Type -> Type -> TypeContext -> Bool
```

```
> subTypes Aint Aint k = True
> subTypes Abool Abool k = True
> subTypes (Aname cn1) (Aname cn2) ((pi,cs,ts),i) =
>    (Aname cn1, Aname cn2) `elem` i
> subTypes (Product types1) (Product types2) k =
>    ((length types1) == (length types2)) &&
>      (all (\(x,y) -> subTypes x y k) pairs)
>    where pairs = zip types1 types2
> subTypes (ps1 :-> return1) (ps2 :-> return2) k =
>    (subTypes return1 return2 k) &&
>      (subTypes ps2 ps1 k)
> subTypes (Ameth (t1,t2)) (Ameth (t1',t2')) k =
>    (subTypes t1 t1' k) && (subTypes t2 t2' k)
> subTypes (Api pi) (Api pi') k =
>    all (\(i,t') -> case (apply pi i) of
>            (Just t) -> subTypes t t' k
>            Nothing  -> False) pi
> subTypes _ _ k = False


> getDeclContext :: Declaration -> TypeContext
> getDeclContext (Class cid [] fds meths) =
>   getDeclContext (Class cid [top] fds meths)

> getDeclContext (Class c [s] fds meths) =
>   let
>     (Api methEnv) = getMethEnv meths c
>     cls = (fds, methEnv, s)
>   in
>     (([],[(c, Acls cls)],[]),[(Aname c, Aname s)])

> getDeclContext (TupleClass fds meths) =
>   let
>     tpfl = Product (map (\(i :< t) -> t) fds)
>     (Api methEnv) = getMethEnv meths "tuple"
>     tup = (fds, methEnv)
>   in
>     (([],[],[(tpfl, Atup tup)]),[])
```

```
> getMethEnv :: [Method] -> ClassName -> Type
> getMethEnv [] cid = Api []
> getMethEnv (m:ms) cid =
>   let
>     (Api dict1) = getMethInfo m cid
>     (Api dict2) = getMethEnv ms cid
>   in
>     case (dict1 `unionDot` dict2) of
>       Just dict
>          ->   Api dict
>       Nothing
>          ->   error "duplicate method declaration"

> getMethInfo :: Method -> ClassName -> Type
> getMethInfo (Method mn its rt e) cid =
>    Api [(mn, (Ameth (Product (field_types its) :-> rt,
>                      Api [])))]

> consolidateContext :: TypeContext -> TypeContext
>                                   -> TypeContext
> consolidateContext k1 k =
>   let
>     ((pi, classes, ts),inh) = k1
>     k2 = k `contextShadow` k1
>     ((pi1, cs1, ts1), inh1) = updateInheritance k2
>     classTree = buildClassTree (Aname top) inh
>     sortedlist = (traverse classTree) \\ [(Aname top)]
>     newclasses = updateClassEnv classes sortedlist cs1
>   in
>     ((pi,newclasses,ts),inh1)
>   where
>     updateClassEnv classes [] hypoclasses = classes
>     updateClassEnv classes ((Aname c):cs) hcs =
>     updateClassEnv
>      (update classes c (Acls (flds, dict,sid))) cs hcs
>        where
>           flds = map (\(i,t) -> (i:<t)) flds1
```

```
>           Just flds1 = (map (\(i:<t) -> (i,t)) newflds)
>                 `unionDot`
>                    (map (\(i :< t) -> (i,t)) sflds)
>           dict = (sdict `unionMinus` newdict)
>           Just (Acls (newflds, newdict, sid)) =
>                 lookup c classes
>           (sflds, sdict) =  case (lookup sid hcs) of
>                    Just (Acls (sflds, sdict,_))
>                            -> (sflds, sdict)
>                    Nothing -> error (sid ++
>                                    " undefined")

> buildClassTree :: Type -> Inheritance -> Tree Type
> buildClassTree cid inh =
>   Node cid (map (\id -> buildClassTree id inh) subtrees)
>     where subtrees = [ cls | (cls, super_cls) <- inh,
>                              super_cls == cid]

> implementationSideCheck :: TypeContext -> Bool
> implementationSideCheck k = checkTuples ts k
>    where ((_,_,ts),_) = k

> checkTuples :: Tuples -> TypeContext -> Bool
> checkTuples [] k = True
> checkTuples ((t,Atup (its,mds)):ts) k =
>   (all (\m -> checkTupleMethod m t k) mds) &&
>   (checkTuples ts k)

> checkTupleMethod :: (MethodName, Type) -> Type ->
>                     TypeContext -> Bool
> checkTupleMethod m t k =
>   (checkTupleMethodMono m t k) &&
>   (checkTupleMethodAmbi m t k)

> checkTupleMethodMono :: (MethodName, Type) -> Type ->
>                         TypeContext -> Bool
> checkTupleMethodMono (mn,mt) t k =
>   let
```

```
>      methodtypes = [ mt' | (t',Atup (its',mds')) <- ts,
>                          subTypes t t' k,
>                          (mn, mt') <- mds',
>                          arity mt' == arity mt]
>   in
>      all (\s -> subTypes mt s k) methodtypes
>   where
>      ((_,_,ts),_) = k
>      arity (Ameth (Product ts:->rt,_)) = length ts

> checkTupleMethodAmbi :: (MethodName, Type) -> Type
>                           -> TypeContext -> Bool
> checkTupleMethodAmbi (mn,mt) (Product t) k =
>   let
>      tuples = [t'|(Product t', Atup (its',mds')) <- ts,
>                     length t' == length t,
>                     mds' `has` mn,
>                     all (\(a,a') -> areRelated a a' k)
>                         (zip t t')]
>      pairs = map (\t' -> (t',t)) tuples
>   in
>      all (\(a,a') ->
>         ((minTypes a a' k) `elem` tuples)) pairs
>   where
>      ((_,_,ts),_) = k
>      has mds mn = case (lookup mn mds) of
>              (Just _) -> True
>              Nothing  -> False

> areRelated :: Type -> Type -> TypeContext -> Bool
> areRelated t1 t2 k =
>    (subTypes t1 t2 k) || (subTypes t2 t1 k)

> minType :: Type -> Type -> TypeContext -> Type
> minType t1 t2 k = if (areRelated t1 t2 k)
>         then
>            if (subTypes t1 t2 k)
>               then t1
```

```
>               else t2
>         else
>            error "incomparable types"

> minTypes :: [Type] -> [Type] -> TypeContext -> [Type]
> minTypes ts1 ts2 k =
>   if ((length ts1) == (length ts2))
>      then let
>             pairs = zip ts1 ts2
>           in
>             map (\(a,a') -> minType a a' k) pairs
>      else error "different lengths of types"

> checkClassMethDict :: FiniteFun Identifier Type
>                       -> ClassName -> TypeContext -> Bool
> checkClassMethDict pi sid k = all
>    (\(mn,mtype) -> checkClassMeth (mn,mtype) sid k) pi

> checkClassMeth :: (MethodName, Type) -> ClassName
>                    -> TypeContext -> Bool
> checkClassMeth (mn,mtype) sid ((pi,cs,ts),inh) =
>   let
>     sdict = case (lookup sid cs) of
>         Just (Acls (_,dict,_)) -> dict
>         Nothing -> error (sid++" undefined")
>   in
>     case (lookup mn sdict) of
>         (Just mtype')
>             -> subTypes mtype mtype' ((pi,cs,ts),inh)
>         Nothing
>             -> True
```