Multiple Dispatch as Dispatch on Tuples

Gary T. Leavens

Department of Computer Science,
Iowa State University,
229 Atanasoff Hall, Ames, Iowa, 50011 USA
leavens@cs.iastate.edu
+1 515 294-1580

Todd D. Millstein

Department of Computer Science and Engineering,
University of Washington,
Box 352350, Seattle, WA 98195-2350 USA
todd@cs.washington.edu
+1 206 543-5118

ABSTRACT

Many popular object-oriented programming languages, such as C++, Smalltalk-80, Java, and Eiffel, do not support multiple dispatch. Yet without multiple dispatch, programmers find it difficult to express binary methods and design patterns such as the "visitor" pattern. We describe a new, simple, and orthogonal way to add multimethods to single-dispatch object-oriented languages, without affecting existing code. The new mechanism also clarifies many differences between single and multiple dispatch.

Keywords

Multiple dispatch, multimethods, generic functions, typing, single dispatch, binary methods, semantics, language design, Tuple.

1 INTRODUCTION

Single dispatch, as found in C++ [Stroustrup 97], Java [Arnold & Gosling 98, Gosling *et al.* 96], Smalltalk-80 [Goldberg & Robson 83], and Eiffel [Meyer 92, Meyer 97], selects a method using the dynamic class of one object, the message's receiver. *Multiple dispatch*, as found in CLOS [Chapter 28, Steele 90] [Paepcke 93], Dylan [Shalit 97, Feinberg *et al.* 97], and Cecil [Chambers 92, Chambers 95], generalizes this idea, selecting a method based on the dynamic class of any subset of the message's arguments. Multiple dispatch is in many ways more expressive and flexible than single dispatch in object-oriented (OO) programming [Bobrow *et al.* 86, Chambers 92, Castagna 97, Moon 86].

In this paper we propose a new, simple, and orthogonal way of adding multiple dispatch to existing languages with single dispatch. The idea is to add tuples as primitive expressions and to allow messages to be sent to tuples. Selecting a method based on the dynamic classes of the elements of the tuple gives multiple dispatch. To illustrate the idea, we have designed a simple class-based OO language called *Tuple*[*]. While perhaps not as elegant as a language built directly with multiple dispatch, we claim the following advantages for our mechanism:

1. It can be added to existing single dispatch languages, such as C++ and Java, without affecting either (a) the semantics or (b) the typing of existing programs written in these languages.
2. It retains the encapsulation mechanisms of single-dispatch languages.
3. It is simple enough to be easily understood and remembered, we believe, by programmers familiar with standard single dispatch.
4. It is more uniform than previous approaches of incorporating multiple dispatch within a single-dispatching framework.

To argue for the first two claims, we present the semantics of the language in two layers. The first layer is a small, class-based single-dispatching language, *SDCore*, of no interest in itself. The second layer, Tuple proper, includes SDCore and adds multiple dispatch by allowing messages to be sent to tuples.

Our support for the third claim is the simplicity of the mechanism itself. If, even now, you think the idea of sending messages to tuples is clearly equivalent to multiple dispatch, then you agree. To support the fourth claim, we argue below that our mechanism has advantages over others that solve the same problem of incorporating multiple dispatch into existing single-dispatching languages.

The rest of this paper is organized as follows. In Section 2 we describe the single-dispatching core of Tuple; this is needed only to show how our mechanism can be added to such a language. In Section 3 we describe the additions to the single- dispatching core that make up our mechanism and support our first three claims. Section 4 supports the fourth claim by comparison with related work. In Section 5 we offer

Appears in the *OOPSLA '98 Conference Proceedings, Conference on Object-Oriented Programming, Systems, and Applications*, Vancouver, British Columbia, Canada, October 18-22, 1998, pp. 374-387.

[*] With apologies to Bruce *et al.*, whose TOOPLE language [Bruce *et al.* 93] is pronounced the same way.

```
class Point {
    fields (xval:int, yval:int)
    method x():int { xval }
    method y():int { yval }
    method distanceFrom(l:Line):int
        { ... }
}

class ColorPoint inherits Point {
    fields (colorval:Color)
    method color():Color { colorval }
}
```

Figure 1: The classes `Point` and `ColorPoint`.

some conclusions. In Appendix A we give the concrete syntax of Tuple, and in Appendix B we give the language's formal typing rules.

2 SDCORE, THE SINGLE-DISPATCHING CORE OF TUPLE

In this section, we introduce the syntax and semantics of SDCore by example.

2.1 Syntax and Semantics

The single-dispatching core of Tuple is a class-based language that is similar in spirit to conventional OO languages like C++ and Java. Integer and boolean values, along with standard operations on these values, are built into the language. Figure 1 illustrates SDCore with the well-known Point/ColorPoint example. The `Point` class contains two fields, representing an x and y coordinate respectively. Each point instance contains its own values for these fields, supplied when the instance is created. For example, the expression **`new`** `Point(3,4)` returns a fresh point instance with `xval` and `yval` set to 3 and 4 respectively. The `Point` class also contains methods for retrieving the values of these two fields and for calculating the distance from the point to some line. (We assume the `Line` class is defined elsewhere.)

For ease of presentation, SDCore's encapsulation model is extremely simple. An instance's fields are only accessible in methods where the instance is the receiver argument. An instance may contain inherited fields, which this rule allows to be accessed directly; this is similar to the *protected* notion in C++ and Java.

The `ColorPoint` class is a simple subclass of the `Point` class, augmenting that definition with a `colorval` field and a method to retrieve the color. (We assume that a class `Color` is defined elsewhere.) The `ColorPoint` class inherits all of the `Point` class's fields and methods. To create an instance of a subclass, one gives initial values for inherited fields first and then the fields declared in the subclass. For example, one would write **`new`** `ColorPoint(3,5,red)`. As usual, subclasses can also override inherited methods. To simplify the language, SDCore has only single inheritance. (However, multiple inheritance causes no problems with respect to the new ideas we are advocating.)

We use a standard syntax for message sends. For example, if `p1` is an instance of `Point` or a subclass, then the expression `p1.distanceFrom(ln2)` invokes the instance's `distanceFrom` method with the argument `ln2`. Within the method body, the keyword **`self`** refers to the *receiver* of the message, in this case `p1`.

Dynamic dispatching is used to determine which method to invoke. In particular, the receiver's class is first checked for a method implementation of the right name and number of arguments. If no such method exists, then that class's immediate superclass is checked, and so on up the inheritance hierarchy until a valid implementation is found (or none is found and a *message-not-understood* error occurs).

For simplicity, and to ease theoretical analysis, we have not included assignment and mutation in SDCore. Again, these could be easily added.

2.2 Type Checking for SDCore

An overview of the static type system for SDCore is included here for completeness. The details (see Appendix B) are intended to be completely standard. For ease of presentation SDCore's type system is simpler than that found in more realistic languages, as this is peripheral to our contributions.

There are four kinds of type attributes. The types **`int`** and **`bool`** are the types of the built-in integer and boolean values, respectively. The *function type* $(T_1,...,T_n)\rightarrow T_{n+1}$ is the type of functions that accept as input n argument values of types T_1, ..., T_n respectively and return a result of type T_{n+1}. The *class type*, *CN*, where *CN* is a class name, is the type of instances of the class *CN*.

Types are related by a simple subtyping relation. The types **`int`** and **`bool`** are only subtypes of themselves. The ordinary contravariant subtyping rule is used for function types [Cardelli 88]. A class type CN_1 is a subtype of another class type CN_2 if the class CN_1 is a subclass of the class CN_2. To ensure the safety of this rule, the type system checks that, for every method name m in class CN_2, m's type in CN_2 is a supertype of m's type in CN_1.* Classes that do not meet this check will be flagged as errors. Thus every subclass that passes the type checker implements a subtype of its superclass.

To statically check a message send expressionof the form $E_0.I(E_1,...,E_n)$, we check that the static type of E_0 is a subtype of a class type *CN* whose associated class contains a method I of type $(T_1,...,T_n)\rightarrow T_{n+1}$, where the types of the expressions $E_1,...,E_n$ are subtypes of the $T_1,...,T_n$

* Unlike C++ and Java, SDCore does not allow static overloading of method names. However, since we add multiple dispatch to SDCore by a separate mechanism, and since dynamic dispatch can be seen as dynamic overloading, there is little reason to do so.

respectively; in this case, $E_0.I(E_1,...,E_n)$ has type T_{n+1}. For example, `p1.distanceFrom(ln2)` has type **int**, assuming that `p1` has type `Point` and `ln2` has type `Line`.

2.3 Problems with Single Dispatching

Single dispatching does not easily support some programming idioms. The best-known problem of this sort is the "binary method problem" [Bruce *et al.* 95]. For example, consider adding an equality test to the `Point` and `ColorPoint` classes above as follows. (For simplicity, in SDCore we have not included **super**, which would allow `ColorPoint`'s `equal` method to call `Point`'s `equal` method.)

```
class Point {
    ...
    method equal(p:Point):bool
        { xval = p.x() and yval = p.y() }
}
class ColorPoint inherits Point {
    ...
    method equal(p:ColorPoint):bool
        -- a type error in SDCore
        { (xval = p.x() and yval = p.y())
            and colorval.equal(p.color()) }
}
```

As is well-known, this makes it impossible for `ColorPoint` to be considered a subtype of `Point` [Cook *et al.* 90]. In other words, `ColorPoint` instances cannot be safely used wherever a `Point` is expected, so polymorphism on the point hierarchy is lost. (For this reason the example is ill-typed in SDCore.)

The problem is semantic, and not a fault of the SDCore type system. It stems from the asymmetry in the treatment of the two `Point` instances being tested for equality. In particular, one instance is the message receiver and is thereby dynamically dispatched upon, while the other is an ordinary argument to the message and plays no role in method selection [Castagna 95]. Multiple dispatch avoids this asymmetry by dynamically dispatching based on the run-time class of *both* arguments.

A more general problem is the "visitor" design pattern [Pages 331-344, Gamma *et al. 95*]. This pattern consists of a hierarchy of classes, typically representing the elements of a structure. In order to define new operations on these elements without modifying the element classes, a separate hierarchy of visitors is created, one per operation. The code to invoke when a "visit" occurs is dependent both on which visitor and which element is used. Multimethods easily express the required semantics [Section 7, Baumgartner *et al.* 96], while a singly-dispatched implementation must rely on unwieldy simulations of multiple dispatching.

```
tuple class (p1:Point, p2:Point) {
    method equal():bool
        {  p1.x() = p2.x()
           and p1.y() = p2.y() }
}
tuple class (cp1:ColorPoint,
             cp2:ColorPoint) {
    method equal():bool
        {  (cp1.x() = cp2.x()
           and cp1.y() = cp2.y())
           and (cp1.color(),
                cp2.color()).equal() }
}
```

Figure 2: Two tuple classes holding methods for testing equality of points.

3 TUPLE, THE MULTIPLE-DISPATCHING EXTENSION OF SDCORE

Tuple extends SDCore with tuple expressions, tuple classes, tuple types, and the ability to declare and send messages to tuples, which gives multiple dispatch. Nothing in the semantics or typing of SDCore is changed by this extension.

3.1 Syntax and Semantics

In Tuple the expression $(E_1, ..., E_n)$ creates a tuple of size n with components $v_1,...,v_n$, where each v_i is the value of the corresponding expression E_i.*

Figure 2 shows how one would solve the Point/ColorPoint problem in Tuple. Rather than defining `equal` methods within the `Point` and `ColorPoint` classes, we create two new *tuple classes* for the methods. In the first tuple class, a tuple of two `Point` instances is the receiver. The names `p1` and `p2` can be used within all methods of the tuple class to refer to the tuple components. However, tuple classes are client code and as such have no privileged access to the fields of such components. The second tuple class is similar, defining equality for a tuple of two `ColorPoint` instances. (We assume that there is a tuple class for the tuple (`Color`, `Color`) with an `equal` method.) There can be more than one tuple class for a given tuple of classes.†

Since no changes are made to the `Point` or `ColorPoint` classes when adding `equal` methods to tuple classes, the subtype relationship between `ColorPoint` and `Point` is unchanged. That is, by adding the `equal` method to a tuple class instead of to the original classes of Figure 1, `ColorPoint` remains a safe subtype of `Point`.

The syntax for sending a message to a tuple is analogous to that for sending a message to an instance. For example, `(p1,p2).equal()` sends the message "`equal()`" to the tuple `(p1,p2)`, which will invoke one of the two

* As in ML [Milner *et al.* 90], we do not allow tuples of length one. This prevents ambiguity in the syntax and semantics. For example, an expression such as `(x).g(y)` is interpreted as a message sent to an instance, not to a tuple. Tuples have either zero, or two or more elements.

† We allow the built-in types **int** and **boolean** to appear as components of a tuple class as well. Conceptually, one can think of corresponding built-in classes **int** and **boolean**, each of which has no non-trivial subclasses or superclasses.

`equal` methods. Just as method lookup in SDCore relies on the dynamic class of the receiver, method lookup in Tuple relies on the dynamic classes of the tuple components. Therefore, the appropriate `equal` method is selected from either the (`Point`, `Point`) or the (`ColorPoint`, `ColorPoint`) tuple class based on the dynamic classes of `p1` and `p2`. In particular, the method from the (`ColorPoint`, `ColorPoint`) tuple class is only invoked if both arguments are `ColorPoint` instances at run-time. The use of dynamic classes distinguishes multiple dispatch from static overloading (as found, for example, in Ada 83 [Ada 83]).

The semantics of sending messages to tuples, multiple dispatch, is similar to that in Cecil [Chambers 95]. Consider the expression $(E_1,...,E_n).I(E_{n+1},...,E_m)$, where each E_i has value v_i, and where $C_{d,i}$ is the minimal dynamic class of v_i. A method in a tuple class $(C_1,...,C_n)$ is *applicable* to this expression if the method is named I, if for each $1 \le i \le n$ the dynamic class $C_{d,i}$ is a subclass of C_i, and if the method takes $m-n$ additional arguments. (The classes of the additional arguments are not involved in determining applicability, but their number does matter.) Among the applicable methods (from various tuple classes), a unique most-specific method is chosen. A method M_1 in a tuple class $(C_{1,1},...,C_{1,n})$ is *more specific than* a method M_2 in a tuple class $(C_{2,1},...,C_{2,n})$ if for each $1 \le i \le n$, $C_{1,i}$ is a subclass of $C_{2,i}$. (The other arguments in the methods are not involved in determining specificity.) If no applicable methods exist, a *message-not-understood* error occurs. If there are applicable methods but no most-specific one, a *message-ambiguous* error occurs. Algorithms for efficiently implementing multiple dispatch exist (see, e.g., [Kiczales & Rodriguez 93]).

This semantics immediately justifies part 1(a) of our claim for the orthogonality of the multiple dispatch mechanism. An SDCore expression cannot send a message to a tuple. Furthermore, the semantics of message sends has two cases: one for sending messages to instances and one for sending messages to tuples. Hence the meaning of an expression in SDCore is unaffected by the presence of multiple dispatch.

The semantics for tuple classes also justifies our second claim. That is, since tuple classes have no special privileges to access the fields of their component instances, the encapsulation properties of classes are unaffected. However, because of this property, Tuple, like other multimethod languages, does not solve the "privileged access" aspect of the binary methods problem [Bruce *et al.* 95]. It may be that a mechanism such as C++ friendship grants would solve most of this in practice. We avoided giving methods in tuple classes default privileged access to the fields of the instances in a tuple because that would violate information hiding. In particular, any client could access the fields of instances of a class C simply by creating a tuple class with C as a component.

```
tuple class (p:Point, out:Terminal) {
    method print():()
        {   -- prints Points to the terminal
            ... }
}

tuple class (p:Point, out:Printer) {
    method print():()
        {   -- prints Points to the printer
            ... }
}

tuple class (cp:ColorPoint,
             out:ColorPrinter) {
    method print():()
        {   -- print ColorPoints to the printer in color
            ... }
}
```

Figure 3: Multimethods in tuple classes for printing. The unit tuple type, `()`, is like C's void type.

3.2 Multiple Dispatch is not just for Binary Methods

Multimethods are useful in many common situations other than binary methods [Chambers 92, Baumgarter *et al.* 96]. In particular, any time the piece of code to invoke depends on more than one argument to the message, a multimethod easily provides the desired semantics.

For example, suppose one wants to print points to output devices. Consider a class `Output` with three subclasses: `Terminal`, an ordinary `Printer`, and a `ColorPrinter`. We assume that `ColorPrinter` is a subclass of `Printer`. Printing a point to the terminal requires different code than printing a point to either kind of printer. In addition, color printing requires different code than black-and-white printing. Figure 3 shows how this situation is programmed in Tuple.

In this example, there is no binary method problem. In particular, the addition of print methods to the `Point` and `ColorPoint` classes will not upset the fact that `ColorPoint` is a subtype of `Point`. The problem is that we need to invoke the appropriate method based on *both* whether the first argument is a `Point` or `ColorPoint` and whether the second argument is a `Terminal`, `Printer`, or `ColorPrinter`. In a singly-dispatched language, an unnatural work-around such as Ingalls's "double dispatching" technique [Ingalls 86, Bruce *et al.* 95] is required to encode the desired behavior.

3.3 Tuples vs. Classes

The ability to express multiple dispatching via dispatching on tuples is not easy to simulate in a single-dispatching language, as is well-known [Bruce *et al.* 95]. The Ingalls double-dispatching technique mentioned above is a faithful simulation but often requires exponentially (in the size of the tuple) more methods than a multimethod-based solution.

A second attempt to simulate multiple dispatch in single-dispatching languages is based on product classes [Section

3.2, Bruce *et al.* 95]. This simulation is not faithful, as it loses dynamic dispatch. However, it is instructive to look at how this simulation fails, since it reveals the essential capability that Tuple adds to SDCore. Consider the following classes in SDCore (adapted from the Bruce *et al.* paper).

```
class TwoPoints {
    fields(p1:Point, p2:Point)
    method equal()
        {  p1.x() = p2.x()
           and p1.y() = p2.y() }
}
class TwoColorPoints {
    fields(cp1:ColorPoint, cp2:ColorPoint)
    method equal()
        {  (cp1.x() = cp2.x()
           and cp1.y() = cp2.y())
           and (new TwoColors(
                  cp1.color(),
                  cp2.color())).equal() }
}
```

With these classes, one could create instances that simulate tuples via the **new** expression of SDCore. For example, an instance that simulates a tuple containing two `Point` instances is created by the expression **new** `TwoPoints(my_p1,my_p2)`. However, this loses dynamic dispatching. The problem is that the **new** expression requires the name of the associated class to be given statically. In particular, when the following message send expression is executed

```
(new TwoPoints(my_p1,my_p2)).equal()
```

the method in the class `TwoPoints` will always be invoked, even if both `my_p1` and `my_p2` denote `ColorPoint` instances at run-time.

By contrast, a tuple expression does not statically determine what tuple classes are applicable. This is because messages sent to tuples use the dynamic classes of the values in the tuple instead of the static classes of the expressions used to construct the tuple. For example, even if the static classes of `my_p1` and `my_p2` are both `Point`, if `my_p1` and `my_p2` denote `ColorPoint` instances, then the message send expression `(my_p1,my_p2).equal()` will invoke the method in the tuple class for (`ColorPoint`, `ColorPoint`) given in Figure 2. Hence sending messages to tuples is not static overloading but dynamic overloading. It is precisely multiple dispatch.

Of course, one can also simulate multiple dispatch by using a variant of the *typecase* statement to determine the dynamic types of the arguments and then dispatching appropriately. (For example, in Java one can use the `getClass` method provided by the class `Object`.) However, writing such code by hand will be more error-prone than automatic dispatch by the language. Such dispatch code will also need to be duplicated in each method that requires multiple dispatch, causing code maintenance problems. Every time a new class enters the system, the dispatch code will need to be appropriately rewritten in each of these places, while in Tuple no existing code need be modified. Further problems can arise if the intent of the dispatch code (to do dispatch) is not clear to maintenance programmers. By contrast, when writing tuple classes it is clear that multiple dispatch is desired. The semantics of Tuple ensures that each dispatch is handled consistently, and the static type system ensures that this dispatching is complete and unambiguous.

3.4 Type Checking for Tuple

We add to the type attributes of SDCore *product types* of the form $(T_1,...,T_n)$; these are the types of tuples containing elements of types $T_1,...,T_n$. A product type $(T_1',...,T_n')$ is a subtype of $(T_1,...,T_n)$ when each T_i' is a subtype of T_i.

Because of the multiple dispatching involved, type checking messages sent to tuples is a bit more complex than checking messages sent to instances (see Appendix B for the formal typing rules). We divide the additions to SDCore's type system into *client-side* and *implementation-side* rules [Chambers & Leavens 95]. The client-side rules check messages sent to tuples, while the implementation-side rules check tuple class declarations and their methods. The aim of these rules is to ensure statically that at run-time no type mismatches occur and that no *message-not-understood* or *message-ambiguous* error will occur in the execution of messages sent to tuples.

The client-side rule is the analog of the method application rule for ordinary classes described above. In particular, given an application $(E_1,...,E_n).I(E_{n+1},...,E_m)$ we ensure that there is some tuple class for the product type $(T_1,...,T_n)$ such that the static type of $(E_1,...,E_n)$ is a subtype of $(T_1,...,T_n)$. Further, that tuple class must contain a method implementation named I with m-n additional arguments such that the static types of $E_{n+1},...,E_m$ are subtypes of the method's additional argument types. Because the rule explicitly checks for the existence of an appropriate method implementation, this eliminates the possibility of *message-not-understood* errors.

However, the generalization to multiple dispatching can easily cause method-lookup ambiguities. For example, consider again the Point/ColorPoint example from Section 2. Suppose that, rather than defining equality for the tuple classes `(Point,Point)` and `(ColorPoint,ColorPoint)`, we had defined equality instead for the tuple classes `(Point,ColorPoint)` and `(ColorPoint,Point)`. According to the client-side rule given above, an `equal` message sent to two `ColorPoint` expressions is legal, since there exists a tuple class of the right type that contains an appropriate method implementation. The problem is that there exist two such tuple classes, and neither is more specific than the other. Therefore, at run-time such a method invocation will cause a *method-ambiguous* error to occur.

Our solution is based on prior work on type checking for multimethods [Castagna *et al.* 92, Castagna 95]. For each pair of tuple classes $(T_1,...,T_n)$ and $(T_1',...,T_n')$ that have a

method named I that accepts k additional arguments, we check two conditions.

The first check ensures *monotonicity* [Castagna *et al.* 92, Castagna 95, Goguen & Meseguer 87, Reynolds 80]. Let $(S_1,...,S_k)\rightarrow U$ and $(S_1',...,S_k')\rightarrow U'$ be the types of the methods named I in the tuple classes $(T_1,...,T_n)$ and $(T_1',...,T_n')$, respectively. Suppose that $(T_1,...,T_n)$ is a subtype of $(T_1',...,T_n')$. Then $(S_1,...,S_k)\rightarrow U$ must be a subtype of $(S_1',...,S_k')\rightarrow U'$. By the contravariant rule for function types, this means that for each j, S_j' must be a subtype of S_j, and U must be a subtype of U'.

The second check ensures that the two methods are not ambiguous. We define two types S and T to be *related* if either S subtypes T or vice versa. In this case, $min(S,T)$ denotes the one of S and T that subtypes the other. It must be the case that $(T_1,...,T_n)$ and $(T_1',...,T_n')$ are not the same tuple. Further, if for each j, T_j and T_j' are related, then there must be a tuple class $(min(T_1,T_1'),...,min(T_n,T_n'))$ that has a method named I with k additional arguments. The existence of this method is necessary and sufficient to disambiguate between the two methods being checked.

The type rules for tuple classes and message sends to tuples validate part (b) of our first claim. That is, Tuple's extensions to the SDCore type system are orthogonal. The typing rules in Tuple are a superset of the typing rules in SDCore. Hence, if an SDCore program or expression has type T, it will also have type T in Tuple.

3.5 Discussion

In Tuple we chose a by-name typing discipline, whereby there is a one-to-one correspondence between classes and types. This unification of classes with types and subclasses with subtypes allows for a very simple static type system. It also reflects the common practice in popular object-oriented languages. Indeed, this approach is a variant of that used by C++ and Java. (Java's interfaces allow a form of separation of types and classes.) Although the type system is simplistic, the addition of multimethods to the language greatly increases its expressiveness, allowing safe covariant overriding while preserving the equivalence between subclassing and subtyping.

There are several other ways in which we could design the type system. For example, a purely structural subtyping paradigm could be used, with classes being assigned to record types based on the types of their methods. Another possibility would be to maintain by-name typing but keep this typing and the associated subtyping relation completely orthogonal to the class and inheritance mechanisms. This is the approach taken in Cecil [Chambers 95]. We ruled out these designs for the sake of clarity and simplicity.

Another design choice is whether to dispatch on classes or on types. In Tuple, this choice does not arise because of the strong correlation between classes and types. In particular, the Tuple dispatching semantics can be viewed equivalently as dispatching on classes or on types. However, in the two alternate designs presented above, the dispatching semantics could be designed either way. Although both options are feasible, it is conceptually simpler to dispatch on classes, as this nicely generalizes the single-dispatching semantics and keeps the dynamic semantics of the language completely independent of the static type system.

The names of the tuple formals in a tuple class are, in a sense, a generalization of `self` for a tuple. They also allow a very simple form of the pattern matching found in functional languages such as ML [Milner *et al.* 90]. Having the tuple formals be bound to the elements of the tuple allows Tuple, like ML, to include tuple values without needing to build into the language primitive operations to extract the components of a tuple. It is interesting to speculate about the advantages that might be obtained by adding algebraic datatypes and more extensive pattern-matching features to object-oriented languages (see also [Bourdoncle & Merz 97, Ernst *et al.* 98]).

4 RELATED WORK

In this section we discuss two kinds of related work. The first concerns generic-function languages; while these do not solve the problem we address in this paper, using such a language is a way to obtain multiple dispatch. The second, more closely-related work, addresses the same problem that we do: how to add support for multiple dispatch to languages with single dispatch.

An inspirational idea for our work is the technique for avoiding binary methods by using product classes described by Bruce *et al.* [Section 3.2, Bruce *et al.* 95]. We discussed this in detail in Section 3.3 above.

Another source of inspiration for this work was Castagna's paper on covariance and contravariance [Castagna 95]. This makes clear the key idea that covariance is used for all arguments that are involved in method lookup and contravariance for all arguments that are not involved in lookup. In Tuple these two sets of arguments are cleanly separated, since in a tuple class the arguments in the tuple are used in method lookup, and any additional arguments are not used in method lookup. The covariance and contravariance conditions are reflected in the type rules for Tuple by the monotonicity condition.

4.1 Generic-Function Languages

Our approach provides a subset of the expressiveness of CLOS, Dylan, and Cecil multimethods, which are based on generic functions. Methods in tuple classes do not allow generic functions to have methods that dynamically dispatch on different subsets of their arguments. That is, in Tuple the arguments that may be dynamically dispatched upon must be decided on in advance, since the distinction is made by client code when sending messages to tuples. In CLOS, Dylan, and Cecil, this information is not visible to clients. On the other hand, a Tuple programmer can hide this information by

always including all arguments as part of the tuple in a tuple class. (This suggests that a useful syntactic sugar for Tuple might be to use $f(E_1,...,E_n)$ as sugar for $(E_1,...,E_n).f()$ when n is at least 2.)

Second, generic function languages are more uniform, since they only have one dispatching mechanism and can treat single dispatching as a degenerate case of multiple dispatching rather than differentiating between them.

Although we believe that these advantages make CLOS-style multimethods a better design for a new language, the approach illustrated by Tuple has some key advantages for adapting existing singly-dispatched languages to multimethods. First, our design can be used by languages like C++ and Java simply by adding tuple expressions, tuple types, tuple classes, and the ability to send messages to tuples. As we have shown, existing code need not be modified and will execute and type check as before. This is in contrast to the generic function model, which causes a major shift in the way programs are structured.

Second, our model maintains class-based encapsulation, keeping the semantics of objects as self-interpreting records. The generic function model gives this up and must base encapsulation on scoping constructs, such as packages [Chapter 11, Steele 90] or local declarations [Chambers & Leavens 97].

4.2 Encapsulated and Parasitic Multimethods

Encapsulated multimethods [Section 4.2.2, Bruce *et al.* 95] [Section 3.1.11, Castagna 97] are similar in spirit to our work in their attempt to integrate multimethods into existing singly-dispatched languages. The following example uses this technique to program equality tests for points in an extension to SDCore.

```
class Point {
    ...
    method equal(p:Point):bool
        {  xval = p.x() and yval = p.y() }
}
class ColorPoint inherits Point {
    ...
    method equal(p:Point):bool
        {  xval = p.x() and yval = p.y() }
    method equal(p:ColorPoint):bool
        {  (xval = p.x() and yval = p.y())
           and colorval.equal(p.color()) }
}
```

With encapsulated multi-methods, each message send results in two dispatches (in general). The first is the usual dispatch on the class of the receiving instance (messages cannot be sent to tuples). This dispatch is followed by a second, multimethod dispatch, to select a multimethod from within the class found by the first dispatch. In the example above, the message `p1.equal(p2)` first finds the dynamic class of the object denoted by `p1`. If `p1` denotes a `ColorPoint`, then a second multimethod dispatch is used to select between the two multimethods for `equal` in the class `ColorPoint`. In essence, the first dispatch selects a generic function formed from the multimethods in the class of the receiver, and the second dispatch is the usual generic function dispatch on the remaining arguments in the message.

One seeming advantage of encapsulated multimethods is that they have access to the private data of the receiver object, whereas in Tuple, a method in a tuple class has no privileged access to any of the elements in the tuple. In languages like C++ and Java, where private data of the instances of a class are accessible by any method in the class, this privileged access will be useful for binary methods. However, this advantage is illusory for multimethods in general, as no special access is granted to private data of classes other than that of the receiver. This means that access must be provided to all clients, in the form of accessor methods, or that some other mechanism, such as C++ friendship grants, must provide such access to the other arguments' private data.

Two problems with encapsulated multimethods arise because the multimethod dispatch is preceded by a standard dispatch to a class. The first problem is the common need to duplicate methods or to use stub methods that simply forward the message to a more appropriate method. For example, since `ColorPoint` overrides the `equal` generic function in `Point`, it must duplicate the `equal` method declared within the `Point` class. As observed in the Bruce *et al.* paper, this is akin to the Ingalls technique for multiple polymorphism [Ingalls 86]. Parasitic multimethods [Boyland & Castagna 97], a variant of encapsulated multimethods, remove this disadvantage by allowing parasitic methods to be inherited.

The second problem caused by the two dispatches is that existing classes sometimes need to be modified when new subclasses are added to the system. For example, in order to program special behavior for the equality method accepting one `Point` and one `ColorPoint` (in that order), it is necessary to modify the `Point` class, adding the new encapsulated multimethod. This kind of change to existing code is not needed in Tuple, as the programmer simply creates a new tuple class. Indeed, Tuple even allows more than one tuple class with the same tuple of component classes, allowing new multimethods that dispatch on the same tuple as existing multimethods to enter the system without requiring the modification of existing code.

Encapsulated and parasitic multimethods have an advantage in terms of modularity over both generic-function languages and Tuple. The modularity problem of generic-function languages, noted by Cook [Cook 90], is that independently-developed program modules, each of which is free of the possibility of *message-ambiguous* errors, may cause *message-ambiguous* errors at run-time. For example, consider defining the method `equal` in three modules: module `A` defines it in a tuple class (`Point`, `Point`), module `B` in a tuple class (`Point`, `ColorPoint`), and module `C` in a tuple class (`ColorPoint`, `Point`). By

themselves these do not cause any possibility of message-ambiguous errors, and a program that uses either `A` and `B` or `A` and `C` will type check. However, a program that includes all three modules may have message-ambiguous errors, since a message sent to a tuple of two `ColorPoint` instances will not be able to find a unique most-specific method. Therefore, a link-time check is necessary to ensure type safety. Research is underway to resolve this problem for generic function languages [Chambers & Leavens 95], which would also resolve it for Tuple. However, to date no completely satisfactory solution has emerged.

The design choices of encapsulated and parasitic multimethods were largely motivated by the goal of avoiding this loss of modularity. Encapsulated multimethods do not suffer from this problem because they essentially define generic functions within classes, and each class must override such a generic function as a whole. (However, this causes the duplication described above.) Parasitic multimethods do not have this problem because they use textual ordering within a class to resolve ambiguities in the inheritance ordering. However, this ordering is hard to understand and reason about. In particular, if there is no single, most-specific parasite for a function call, control of the call gets passed among the applicable parasites in a manner dependent on both the specificity and the textual ordering of the parasites, and determining at which parasite this ping-ponging of control terminates is difficult. Boyland and Castagna also say that, compared with their textual ordering, ordering methods by specificity as we do in Tuple "is very intuitive and clear" [Page 73, Boyland & Castagna 97]. Finally, they note that textual ordering causes a small run-time penalty in dispatching, since the dispatch takes linear instead of logarithmic time, on the average.

5 CONCLUSIONS

The key contribution of this work is that it describes a simple, orthogonal way to add multiple dispatch to existing single-dispatch languages. We showed that the introduction of tuples, tuple types, tuple classes for the declaration of multimethods, and the ability to send messages to tuples is orthogonal to the base language. This is true in both execution and typing. All that tuple classes do is allow the programmer to group several multimethods together and send a tuple a message using multimethod dispatching rules. Since existing code in single-dispatching languages cannot send messages to tuples, its execution is unaffected by this new capability. Hence our mechanism provides an extra layer, superimposed on top of a singly-dispatched core. Design decisions in this core do not affect the role or functionality of tuples and tuple classes.

Tuple also compares well against related attempts to add multiple dispatch to singly-dispatched languages. We have shown that Tuple's uniform dispatching semantics avoids several problems with these approaches, notably the need to "plan ahead" for multimethods or be forced to modify existing code as new classes enter the system. On the other hand, this uniformity also causes Tuple to suffer from the modularity problem of generic-function languages, which currently precludes the important software engineering benefits of separate type checking.

The Tuple language itself is simply a vehicle for illustrating the idea of multiple dispatching via dispatching on tuples. Although it would complicate the theoretical analysis of the mechanism, C++ or Java could be used as the singly-dispatched core language.

ACKNOWLEDGMENTS

Thanks to John Boyland for discussion and clarification about parasitic multimethods. Thanks to the anonymous referees for helpful comments. Thanks to Olga Antropova, John Boyland, Giuseppe Castagna, Sevtap Karakoy, Clyde Ruby, and R. C. Sekar for comments on an earlier draft. Thanks to Craig Chambers for many discussions about multimethods. Thanks to Olga Antropova for the syntactic sugar idea mentioned in Section 4.1. Thanks to Vassily Litvinov for pointing us to [Baumgartner *et al.* 96] and to Craig Kaplan for an idea for an example. Leavens's work was supported in part by NSF Grants CCR 9593168 and CCR-9803843.

REFERENCES

[Ada 83] American National Standards Institute. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD 1815A, February, 1983.

[Amadio & Cardelli 93] Roberto M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems,* 15(4):575-631, 1993.

[Arnold & Gosling 98] Ken Arnold and James Gosling. *The Java Programming Language*. Second Edition, Addison-Wesley, Reading, Mass., 1998.

[Baumgartner *et al.* 96] Gerald Baumgartner, Konstantin Läufer, Vincent 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, February 1996.

[Bobrow *et al.* 86] Daniel G. Bobrow, Kenneth Kahn, George Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In Norman Meyrowitz (editor), *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21, number 11 of *ACM SIGPLAN Notices*, pp. 17-29. ACM, New York, November, 1986.

[Bourdoncle & Merz 97] François Bourdoncle and Stephan Merz. Type Checking Higher-Order Polymorphic Multi-Methods. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 302-315. ACM, New York, January 1997.

[Boyland & Castagna 97] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97).* Volume 32, number 10 of *ACM SIGPLAN Notices*, pp. 66-76. ACM, New York, November 1997.

[Bruce *et al.* 93] Kim B. Bruce and Jon Crabtree and Thomas P. Murtagh and Robert van Gent and Allyn Dimock and Robert Muller. Safe and decidable type checking in an object-oriented language. In Andreas Paepcke (editor), *OOPSLA '93 Conference Proceedings.* Volume 28, number 10 of *ACM SIGPLAN Notices*, pp. 29-46. ACM, New York, October, 1993.

[Bruce *et al.* 95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, **1**(3):221-242, 1995.

[Cardelli 88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, **76**(2/3):138-164, February/March, 1988. An earlier version appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pp. 51-66, Springer-Verlag, 1984.

[Castagna *et al.* 92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, *San Francisco, June, 1992*, pp. 182-192, volume 5, number 1 of *LISP Pointers*. ACM, New York, January-March, 1992.

[Castagna 95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems,* **17**(3):431-447, 1995. See also [Chapter 5, Castagna 97].

[Castagna 97] Giuseppe Castagna. Object-Oriented Programming A Unified Foundation. Birkhäuser, Boston, 1997.

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann-Madsen, editor, *ECOOP '92 Conference Proceedings, Utrecht, the Netherlands, June/July, 1992*, volume 615 of *Lecture Notes in Computer Science*, pp. 33-56. Springer-Verlag, Berlin, 1992.

[Chambers 95] Craig Chambers. The Cecil Language: Specification and Rationale: Version 2.0. Department of Computer Science and Engineering, University of Washington, December, 1995. http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html

[Chambers & Leavens 95] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. *ACM Transactions on Programming Languages and Systems*, **17**(6):805-843. November, 1995.

[Chambers & Leavens 97] Craig Chambers and Gary T. Leavens. BeCecil, A Core Object-Oriented Language with Block Structure and Multimethods: Semantics and Typing. Department of Computer Science, Iowa State University, TR #96-17a, April 1997. ftp://ftp.cs.iastate.edu/pub/techreports/TR96-17/TR.ps.Z; the appendix sections only are in ftp://ftp.cs.iastate.edu/pub/techreports/TR96-17/appendix.ps.Z.

[Cook *et al.* 90] William Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not Subtyping. In C*onference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pp 125-135. ACM, New York, 1990.

[Cook 90] William Cook. Object-Oriented Programming versus Abstract Data Types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop Proceedings, Noordwijkerhout, the Netherlands, May/June, 1990*, volume 489 of *Lecture Notes in Computer Science*, pp. 151-178. Springer-Verlag, New York, 1991.

[Ernst *et al.* 98] Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. To appear in *ECOOP '98, the 12th European Conference on Object-Oriented Programming, Brussels, Belgium*, July, 1998. Also http://www.cs.washington.edu/research/projects/cecil/www/www/Papers/gud.html.

[Feinberg *et al*. 97] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

[Gamma *et al.* 95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[Goguen & Meseguer 87] Joseph A. Goguen and Jose Meseguer. Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems. In *Symposium on Logic in Computer Science, Ithaca, NY*, pp. 18-29. IEEE Press, NY, June, 1987.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[Gosling *et al*. 96] James Gosling, Bill Joy, Guy Steele, Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1996.

[Ingalls 86] D. H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings*, *Portland, Oregon, November, 1986*, volume 21, number 11 of *ACM SIGPLAN Notices*, pp. 347-349. ACM, New York, October, 1986.

[Kiczales & Rodriguez 93] Gregor Kiczales and Luis H. Rodriguez Jr. Efficient Method Dispatch in PCL. In [Paepcke 93], Chapter 14.

[Meyer 92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[Meyer 97] Bertrand Meyer. *Object-Oriented Software Construction*. Second Edition, Prentice Hall, New York, 1997.

[Milner *et al.* 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Moon 86] David A. Moon. Object-Oriented Programming with Flavors. In Norman Meyrowitz (editor), *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21, number 11 of *ACM SIGPLAN Notices*, pp. 1-8. ACM, New York, November, 1986.

[Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Reynolds 80] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In Neil D. Jones (editor), *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, pp. 211-258. Volume 94 of Lecture Notes in Computer Science, Springer-Verlag, NY, 1980.

[Schmidt 94] David A. Schmidt. *The Structure of Typed Programming Languages.* MIT Press, Cambridge, Mass., 1994.

[Shalit 97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[Steele 90] Guy L. Steele Jr. *Common Lisp: The Language (second edition).* Digital Press, Bedford, MA, 1990.

[Stroustrup 97] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley, Reading, Mass., 1997.

APPENDIX A FORMAL SYNTAX

The following is the concrete syntax for Tuple. The *italic* font notations to the right are the names of the syntactic domains.

PROG	: : =		*Program*
		RDS E	
RDS	: : =		*Recursive-Declaration-Seq*
		D | *D RDS*	
D	: : =		*Declaration*
		tuple class *TF* **{** *MS* **}**	
	|	**class** I_1 **{ fields** *F MS* **}**	
	|	**class** I_1 **inherits** I_2 **{ fields** *F MS* **}**	
F	: : =		*Formals*
		() | **(***FS***)**	
FS	: : =		*Formal-List*
		*I***:***T* | *I***:***T***,** *FS*	
TF	: : =		*Tuple-Formals*
		() | **(***TFS***)**	
TFS	: : =		*Tuple-Formal-List*
		I_1**:**T_1**,** I_2**:**T_2	
	|	*I***:***T***,** *TFS*	
T	::=		*Type*
		int | **bool**	
	|	*P* | $T_1 \rightarrow T_2$ | *CN*	
P	: : =		*Product-Type*
		() | **(***TTS***)**	
TTS	: : =		*Tuple-Type-List*
		T_1**,** T_2 | *T***,** *TTS*	
MS	: : =		*Method-List*
		M | *M MS*	
M	: : =		*Method*
		method *I F* **:** *T* **{** *E* **}**	
E	: : =		*Expression*
		new *CN***(***ES***)** | *E***.***MN E* | *TUP*	
	|	**self** | *I*	
	|	*<built-in operations on ints and bools>*	
ES	: : =		*Expression-List*
		E_1 | *E***,** *ES*	
TUP	: : =		*Tuple*
		() | **(***TES***)**	
TES	: : =		*Tuple-Expression-List*
		E_1**,** E_2 | *E***,** *ES*	
CN	: : =		*Class-Name*
		I	
MN	: : =		*Method-Name*
		I	
I	: : =		*Identifier*
		"strings of alphanumerics, starting with an alphabetic"	

APPENDIX B FORMAL TYPING RULES

B.1 Type attributes

For type checking, we extend the type attributes found in the concrete syntax to a larger domain, which we also call *Type*, as follows. (The symbols *P*, *CN*, and *FS* mean the same syntactic domains as above, but we interpret the *T*s that occur in product types and formals as meaning the domain *Type* below.)

T	: : =		*Type*
		int | **bool** | *P* | $T_1 \rightarrow T_2$ | *CN*	
	|	*T* **exp** | *k* **dec** | *c* **cls** | *t* **tup**	
p	: : =	**{}** | **{***FS***}**	*TypeEnv*
k	: : =	**(***p***,** *cs***,** *ts***)**	*TypeContext*
cs	: : =	*p*	*ClassEnv*
c	: : =	**(***F***,** *p***,** *CN***)**	*ClassInfo*
ts	: : =	**[]** | **[***TIS***]**	*Tuples*
TIS	: : =		*TupleInfoRelationships*
		(*P***,** *T***)** | **(***P***,** *T***)** **,** *TIS*	
t	: : =	**(***F***,** *p***)**	*TupleClassInfo*

Following Schmidt [Schmidt 94], type attributes of the form *T* **exp** stand for expressions that return type *T*. Type attributes of the form *k* **dec** stand for declarations that have the effect of making a context *k*. Type attributes of the form *c* **cls** record information, *c*, about a class; these are used as the types of classes (as opposed to their instances). Type attributes of the form *t* **tup** record information, *t*, about a tuple class.

The domains *TypeEnv* and *ClassEnv* are regarded as finite functions from identifiers to types; hence we make sure that there is at most one association for a given identifier in its domain. Similarly we regard the domain *Tuples* as a finite relation between product types and type attributes of the form *t* **tup**.

B.2 Preliminary definitions and functions

We define the following helper functions on finite functions, relations, and lists of formals:

$$dom(\{I_1{:}T_1,...,I_n{:}T_n\}) = \{I_1,...,I_n\}$$
$$range(\{I_1{:}T_1,...,I_n{:}T_n\}) = \{T_1,...,T_n\}$$

The function + denotes a shadowing union of two relations (or functions), favoring the second one:

$$r_1 + r_2 = \{(x,y) \mid (x,y) \in r_2 \textbf{ or } [(x,y) \in r_1 \textbf{ and } x \notin dom(r_2)]\}$$

The function + is also extended to a shadowing union of two *TypeContexts*, favoring the second:

$(p_1,cs_1,ts_1) + (p_2,cs_2,ts_2) = (p_1 + p_2,\ cs_1 + cs_2,\ ts_1 \cup ts_2)$

We define the following helper functions on the domain *ClassInfo*:

field-types$((I_1{:}T_1,...,I_n{:}T_n), p, I') = (T_1,...,T_n)$
meth-env$(F, p, I') = p$

We define the helper function *meth-env* also on the domain *TupleClassInfo*:

meth-env$(F, p) = p$

We define a helper function on the syntax of methods:

name(**method** $I\ F : T$ **{** E **}**) $= I$

We define a helper function on types:

length(T) = if $\exists\ n\ \exists\ T_1\ ...\ \exists\ T_n.\ (n \geq 0$ and $T = (T_1,...,T_n))$ then n else 1

We define a helper function on pairs of types:

min$(S,T,$cs$)$ = if cs $\vdash S \leq T$ then S else T

We define a helper function to check the disjointness of a list of identifiers:

disjoint$(I_1,...,I_n) = \forall\ i.\forall\ j.\ (1 \leq i \leq n$ and $1 \leq j \leq n$
and $I_i = I_j) \Rightarrow (i = j)$

B.3 Type Checking Rules

Besides T, we also use S, U, and V for types. Recall that P is used for product types.

B.3.1 Type Checking Programs

[program]

$$\frac{k_0{+}k \vdash RDS{:}\ k\ \textbf{dec},\ \ k_0{+}k \vdash \textit{impl-side-type-check},\quad k_0{+}k \vdash E : T\ \textbf{exp}}{\vdash RDS\ E : T\ \textbf{exp}}$$

where $k_0 =$
$(\{\},\{Top{:}((),\{\},Top)\ \textbf{cls}\},\{\})$

Figure B-1: Rule of the form $\vdash$ *Program* : *Type*. The class name *Top* is used to avoid special cases that would result from some classes not having a superclass.

B.3.2 Type Checking Recursive Declaration Sequences

A recursive declaration sequence allows mutual recursion among its declarations.

[rds]

$$\frac{k + (k_1 + k_2) \vdash D : k_1\ \textbf{dec},\quad k + (k_1 + k_2) \vdash RDS : k_2\ \textbf{dec}}{k \vdash D\textbf{;}\ RDS : (k_1 + k_2)\ \textbf{dec}}$$

where $(p_1,cs_1,ts_1) = k_1$,
$(p_2,cs_2,ts_2) = k_2$,
$dom(p_1) \cap dom(p_2) = \{\}$,
$dom(cs_1) \cap dom(cs_2) = \{\}$

Figure B-2: Rule of the form *TypeContext* $\vdash$ *Recursive-Declaration-Sequence* : *TypeContext* **dec**.

B.3.3 Type Checking Declarations

[tuple-dec]
$$\frac{k + k_{args} \vdash M_1 : S_1 \rightarrow T_1,\ \ldots,\ k + k_{args} \vdash M_n : S_n \rightarrow T_n}{k \vdash \textbf{tuple class}\ (I_1{:}U_1,\ldots,I_m{:}U_m)\ \{\ M_1 \ldots M_n\ \} : (\{\},\{\},\{((U_1,\ldots,U_m),((I_1{:}U_1,\ldots,I_m{:}U_m),p)\textbf{tup})\})\ \textbf{dec}}$$
where $m \geq 0$, $n \geq 0$, $disjoint(I_1,\ldots,I_m)$, $k_{args} = (\{I_1{:}U_1,\ldots,I_m{:}U_m\},\{\},\{\})$, $p = \{MN_1{:}S_1 \rightarrow T_1,\ldots,MN_n{:}\ S_n \rightarrow T_n\}$, $\forall\ i.\ 1 \leq i \leq n \Rightarrow MN_i = name(M_i)$

[class-dec]
$$\frac{k \vdash \textbf{class}\ I\ \textbf{inherits}\ Top\ \{\ \textbf{fields}\ (I_1{:}U_1,\ldots,I_m{:}U_m)\ M_1 \ldots M_n\ \} : k'\ \textbf{dec}}{k \vdash \textbf{class}\ I\ \{\ \textbf{fields}\ (I_1{:}U_1,\ldots,I_m{:}U_m)\ M_1 \ldots M_n\ \} : k'\ \textbf{dec}}$$
where $m \geq 0$, $n \geq 0$

[class-inh-dec]
$$\frac{\begin{array}{l}(p,cs,ts) + (\{(\textbf{self},I)\},\{\},\{\}) + k_{flds} \vdash M_1 : S_1 \rightarrow T_1,\\ \ldots,\\ (p,cs,ts) + (\{(\textbf{self},I)\},\{\},\{\}) + k_{flds} \vdash M_n : S_n \rightarrow T_n,\\ cs \vdash S_{\pi(1)} \rightarrow T_{\pi(1)} \leq S_{\pi(1)}' \rightarrow T_{\pi(1)}',\\ \ldots,\\ cs \vdash S_{\pi(i)} \rightarrow T_{\pi(i)} \leq S_{\pi(i)}' \rightarrow T_{\pi(i)}',\end{array}}{\begin{array}{l}(p,cs,ts) \vdash \textbf{class}\ I\ \textbf{inherits}\ I'\{\\ \qquad\qquad \textbf{fields}\ (I_1{:}U_1,\ldots,I_m{:}U_m)\ M_1 \ldots M_n\ \}\\ : (\{\},\ \{I{:}((I_{c,1}{:}V_1,\ldots,I_{c,l}{:}V_l,I_1{:}U_1,\ldots,I_m{:}U_m),p,I')\textbf{cls}\},\\ \qquad \{\})\ \textbf{dec}\end{array}}$$
where $m \geq 0$, $n \geq 0$, $l \geq 0$, $i \geq 0$, $(I'{:}c\ \textbf{cls}) \in cs$, $p' = meth\text{-}env(c)$, $(I_{c,1}{:}V_1,\ldots,I_{c,l}{:}V_l) = fields(c)$, $disjoint(I_{c,1},\ldots,I_{c,l},I_1,\ldots,I_m)$, $k_{flds} = (\{I_{c,1}{:}V_1,\ldots,I_{c,l}{:}V_l\},\{\},\{\}) + (\{I_1{:}U_1,\ldots,I_m{:}U_m\},\{\},\{\})$, $\forall\ j.\ 1 \leq j \leq n \Rightarrow MN_j = name(M_j)$, $p = p' + \{MN_1{:}S_1 \rightarrow T_1,\ldots,MN_n{:}\ S_n \rightarrow T_n\}$, $\pi{:}\{1,..,n\} \rightarrow \{1,..,n\}$, π is a permutation, $\forall j.\ 1 \leq j \leq i \Rightarrow (MN_{\pi(j)}{:}S_{\pi(j)}' \rightarrow T_{\pi(j)}') \in p'$, $\forall\ j.\ i < j \leq n \Rightarrow MN_{\pi(j)} \notin dom(p')$

Figure B-3: Rules of the form *TypeContext* ⊢ *Declaration* : *TypeContext* **dec**.

B.3.4 Type Checking Methods

[method]
$$\frac{k + (\{(I_1,T_1),\ldots,(I_n,T_n)\},\{\},\{\}) \vdash E : T\ \textbf{exp}}{k \vdash \textbf{method}\ I\ (I_1{:}T_1,\ldots,I_n{:}T_n) : T\ \{\ E\ \} : (T_1,\ldots,T_n) \rightarrow T}$$
where $n \geq 0$, $disjoint(I_1,\ldots,I_n)$

Figure B-4: Rule of the form *TypeContext* ⊢ *Method* : *Type*.

B.3.5 Type Checking Expressions

[new]
$$\frac{(p,cs,ts) \vdash E_1 : T_1\ \mathbf{exp},\ \ldots,\ (p,cs,ts) \vdash E_n : T_n\ \mathbf{exp}}{(p,cs,ts) \vdash \mathbf{new}\ CN\ (E_1,\ldots,E_n) : CN\ \mathbf{exp}}$$
where $n \geq 0$, $(CN,c) \in cs$, $\mathit{field\text{-}types}(c) = (T_1,\ldots,T_n)$

[app-rec]
$$\frac{(p,cs,ts) \vdash E_0 : I_0\ \mathbf{exp},\quad (p,cs,ts) \vdash E : S\ \mathbf{exp}}{(p,cs,ts) \vdash E_0.MN\ E : T\ \mathbf{exp}}$$
where $(I_0{:}c\ \mathbf{cls}) \in cs$, $(MN{:}S{\rightarrow}T) \in \mathit{meth\text{-}env}(cs)$

[app-prod]
$$\frac{(p,cs,ts) \vdash E_0 : (T_1,\ldots,T_n)\ \mathbf{exp},\ \ (p,cs,ts) \vdash E : S\ \mathbf{exp}}{(p,cs,ts) \vdash E_0.MN\ E : T\ \mathbf{exp}}$$
where $n \geq 0$, $n \neq 1$, $((T_1,\ldots,T_n), t\ \mathbf{tup}) \in ts$, $(MN{:}S{\rightarrow}T) \in \mathit{meth\text{-}env}(t)$

[tuple]
$$\frac{k \vdash E_1 : T_1\ \mathbf{exp},\ \ldots,\quad k \vdash E_n : T_n\ \mathbf{exp}}{k \vdash (E_1,\ldots,E_n) : (T_1,\ldots,T_n)\ \mathbf{exp}}$$
where $n \geq 0$, $n \neq 1$

[self]
$$(p,cs,ts) \vdash \mathbf{self} : T\ \mathbf{exp}$$
where $(\mathbf{self}{:}T) \in p$

[id]
$$(p,cs,ts) \vdash I : T\ \mathbf{exp}$$
where $(I{:}T) \in p$

[sub-sumption]
$$\frac{(p,cs,ts) \vdash E : S\ \mathbf{exp},\quad cs \vdash S \leq T}{(p,cs,ts) \vdash E : T\ \mathbf{exp}}$$

Figure B-5: Rules of the form *TypeContext* ⊢ *Expression* : *Type*. We omit rules for the built-in operators on integers and booleans.

B.3.6 Subtyping Rules

[sub-eq]
$$\mathrm{cs} \vdash T \leq T$$

[sub-trans]
$$\frac{\mathrm{cs} \vdash S \leq T,\quad \mathrm{cs} \vdash T \leq U}{\mathrm{cs} \vdash S \leq U}$$

[sub-CN]
$$\mathrm{cs} \vdash CN \leq CN_2$$
where $(CN{:}(F,p,CN_2)\ \mathbf{cls}) \in cs$

[sub-prod]
$$\frac{\mathrm{cs} \vdash S_1 \leq T_1,\ \ldots,\quad \mathrm{cs} \vdash S_n \leq T_n}{\mathrm{cs} \vdash (S_1,\ldots,S_n) \leq (T_1,\ldots,T_n)}$$
where $n \geq 0$, $n \neq 1$

[sub-fun]
$$\frac{\mathrm{cs} \vdash T_1 \leq S_1,\quad \mathrm{cs} \vdash S_2 \leq T_2}{\mathrm{cs} \vdash S_1{\rightarrow}S_2 \leq T_1{\rightarrow}T_2}$$

Figure B-6: Subtyping rules of the form *Classes* ⊢ *Type* ≤ *Type*.

B.3.7 Implementation-side Type Checking Rules

[impl-all]
$$\frac{(p,cs,ts) \vdash \textit{impl-check } MN_1, \ldots, (p,cs,ts) \vdash \textit{impl-check } MN_n}{(p,cs,ts) \vdash \textit{impl-side-type-check}}$$
where $n \geq 0$, $m \geq 0$,
$\{t_1,\ldots,t_m\} = \textit{range}(ts)$,
$\{MN_1,\ldots,MN_n\} = \cup_{i=1..m}\, \textit{dom}(\textit{meth-env}(t_i))$

[impl-method]
$$\frac{\begin{array}{c} cs \vdash (P_1,T_1)\ \textit{is-monotonic-in}\ [(P_1,T_1),\ldots,(P_n,T_n)], \\ \ldots, \\ cs \vdash (P_n,T_n)\ \textit{is-monotonic-in}\ [(P_1,T_1),\ldots,(P_n,T_n)], \\ (p,cs,ts) \vdash (P_1,T_1)\ \textit{unambiguous-with-each}\ [(P_2,T_2),\ldots,(P_n,T_n)]\ \textit{for MN}, \\ \ldots, \\ (p,cs,ts) \vdash (P_{n-1},T_{n-1})\ \textit{unambiguous-with-each}\ [(P_n,T_n)]\ \textit{for MN} \end{array}}{(p,cs,ts) \vdash \textit{impl-check MN}}$$
where $n \geq 0$,
$[(P_1,T_1),\ldots,(P_n,T_n)] = [(P,T) \mid (P,t\ \textbf{tup}) \in ts$ and $(MN{:}T) \in \textit{meth-env}(t)]$

[is-mono]
$$\frac{cs \vdash T_1 \rightarrow T_2 \leq T_{1,1} \rightarrow T_{1,2}, \ldots, cs \vdash T_1 \rightarrow T_2 \leq T_{n,1} \rightarrow T_{n,2}}{cs \vdash (P,T_1 \rightarrow T_2)\ \textit{is-monotonic-in}\ s}$$
where $n \geq 0$,
$\{T_{1,1} \rightarrow T_{1,2},\ldots,T_{n,1} \rightarrow T_{n,2}\} = \{T_{i,1} \rightarrow T_{i,2} \mid cs \vdash P \leq P_i$ and $(P_i,T_{i,1} \rightarrow T_{i,2}) \in s$ and $\textit{length}(T_{i,1}) = \textit{length}(T_1)\}$

[unambiguous-each]
$$\frac{\begin{array}{c} (p,cs,ts) \vdash (P,T)\ \textit{unambiguous-with}\ (P_1,T_1)\ \textit{for MN}, \\ \ldots, \\ (p,cs,ts) \vdash (P,T)\ \textit{unambiguous-with}\ (P_n,T_n)\ \textit{for MN} \end{array}}{(p,cs,ts) \vdash (P,T)\ \textit{unambiguous-with-each}\ [(P_1,T_1),\ldots,(P_n,T_n)]\ \textit{for MN}}$$
where $n \geq 0$

[unambiguous-]
$$(p,cs,ts) \vdash (P,S \rightarrow T)\ \textit{unambiguous-with}\ (P',S' \rightarrow T')\ \textit{for MN}$$
where $n \geq 0$, $m \geq 0$,
$P \neq P'$,
$P = (T_1,\ldots,T_n)$, $P' = (T_1',\ldots,T_m')$,
$(n = m$ and $\textit{length}(S) = \textit{length}(S')$ and $(\forall\, j.\ 1 \leq j \leq n \Rightarrow cs \vdash (T_j,T_j')\ \textit{are-related}))$
$\Rightarrow (((\textit{min}(P_1,P_1'),\text{cs}),\ldots,\textit{min}(P_n,P_n'),\text{cs})),$ $t\ \textbf{tup}) \in ts$ and $(\exists\, S''.\exists T''.$ $(\textit{length}(S'') = \textit{length}(S)$ and $(MN{:}S'' \rightarrow T'') \in \textit{meth-env}(t)))$

[are-related-1]
$$\frac{cs \vdash S \leq T}{cs \vdash (S,T)\ \textit{are-related}}$$

[are-related-2]
$$\frac{cs \vdash T \leq S}{cs \vdash (S,T)\ \textit{are-related}}$$

Figure B-7: Rules for checking the implementation-side type safety of a program. We use list notation [...] and list comprehensions (as in Haskell) in some of these rules.