

MultiCruz: An Implementation of Multiple Dispatch

Jordan Johnson and Damian Eads
University of California, Santa Cruz
1156 High Street
Santa Cruz, CA 95064

December 1, 2005

Abstract

We designed and implemented a typed lambda-calculus that supports nominal types, subtyping and multiple dispatching of methods. We describe the syntax, typing rules, and semantics of our language. We specify the restrictions placed on the methods to guarantee their safe dispatch.

1 Introduction

Object-oriented programming provides an expressive framework for abstracting program data as well as the computations that act on the data. The most popular variants of Object-oriented languages involve classes. Informally, classes are templates for the data elements (called “fields”) contained in the objects. The functions that operate on objects are called “methods”. Methods and fields can be annotated to impose restrictions or capabilities on their use. For example, some languages permit visibility tags (often called `public` and `private`) to limit access of a method or field to certain areas of computation, typically within the scope of another method. Classes can be subtyped (or extended), and methods, overridden and overloaded. The distinction between overriding and overloading a method is an important one. When a method is called, arguments are passed to it, and a branch is executed. The strategy for choosing an appropriate branch is called dispatch. We consider a very powerful form of dispatch called multiple dispatch. We give examples of where multiple dispatch is useful. First, we review prerequisite concepts before giving a detailed discussion of multiple dispatch.

1.1 Overriding

Overriding enables a subclass of a class to replace the named computation (i.e. method) with its own variant. Method dispatch is employed to determine which method implementation to run when a method is invoked. Java uses single dispatch to make its determination.

For example, in Figure 1, a class `FooBar` overrides `toString`. When `printIt` is called, and line 6 is evaluated, which version of `toString` is called? The `Object toString()` will always be compatible, so we could in theory always dispatch the `Object` implementation, but then what’s the point of method override?

The Java compiler cannot make a choice of a method at compile time since we do not know what the run-time type of `o` will be. At run time, the Java interpreter looks at the type of the target object `o`. If the type of `o` is `FooBar`, `FooBar`’s `toString()` method is executed. If the type of `o` is a subclass of `Object` that overrides `toString()`, then its overridden implementation is chosen. Otherwise, `Object`’s `toString()` is chosen during dispatch. The act of just looking at the run-time type of the target object is called single dispatch.

```

1  public class FooBar {
2
3      public String toString() { return "FooBar!"; }
4
5      public void printIt(Object o) {
6          System.out.println(o.toString());
7      }
8
9      public static void main(String args[]) {
10         FooBar bar = new FooBar();
11         bar.printIt(new Object());
12         bar.printIt(new FooBar());
13     }
14 }
15

```

Figure 1: An example of a class with an overridden method, `toString()`

```

1  public class Account {
2
3      private double balance;
4
5      public void deposit(double amount) {
6          balance += amount;
7      }
8
9      public void deposit(double amountDollars, double amountCents) {
10         balance += amountDollars + amountCents;
11     }
12 }
13

```

Figure 2: The first version of the `Account` class. The `deposit` method has two branches.

1.2 Overloading

In contrast to overridden methods, an overloaded function is an alternative to the other functions defined with the same name. A simple example of a Java class with an overloaded function is shown in Figure 2.

Different implementations corresponding to the named method are called branches. The `Account` class in Figure 2 consists of a single method called `deposit` with two branches. There are only two ways `deposit` can be invoked because either one or two values can be passed. If one compatible value is passed, the first method is chosen for dispatch; if two values, the second method is chosen.

When the argument of one method can be a subtype of the argument of another method, things can get a bit more complicated. For example, consider the classes `Money`, `CreditCard`, `Account`, and `Bank` shown in Figures 3, 4, 5, and 6 respectively.

One might think it is sensible to write two implementations of a method, one generalized for all kinds of money, and the others, written to carefully consider semantics specific to a particular kind of money.

In Figure 5, we have changed our `Account` class to define two branches of `deposit`. The first is for general deposits of money, whether it be a cash, check, Gold Bullion, Sand Dollars, whatever. The second is for depositing against a credit card where a specific amount must be authorized. The notion of authorization

```

1  public class Money {
2      ...
3      public double getValue() {}
4      public void decreaseValue(double delta) {}
5  }

```

Figure 3: A generic class for facilitating the notion of money.

```

1  public class CreditCard extends Money {
2      ...
3      public bool authorizeCredit(double amount) {}
4
5  }

```

Figure 4: A class for representing a credit card.

```

1  public class Account {
2
3      private double balance;
4
5      public void deposit(Money money, double amount) {
6          // If the money is of a value greater than or equal to the
7          // deposit request, deposit it.
8          if (amount <= money.getValue()) {
9              balance += amount;
10             money.decreaseValue(amount);
11         }
12     }
13
14     public void deposit(CreditCard cc, double amount) {
15         // If the credit card has enough credit, deduct the credit
16         // from the account.
17         if (cc.authorizeCredit(amount)) {
18             balance += amount;
19         }
20     }
21 }

```

Figure 5: The second version of the `Account` class. In this version, we have two methods: one whose first argument is of type `Money`, and the other, of type `CreditCard`.

```

1  public class Bank {
2
3      public void requestDeposit(Account account, Money money, double amount) {
4          account.deposit(money, amount);
5      }
6
7      public static void main(String args[]) {
8          Bank bank = new Bank();
9          Account account = new Account();
10         bank.requestDeposit(account, new Money(5000.0), 1000.0);
11         bank.requestDeposit(account, new CreditCard(5000.0), 1000.0);
12     }
13
14 }

```

Figure 6: A Bank class for depositing money in generic form or in the form of a credit card transaction.

cannot be generally applied to all other kinds of money, but it is an important detail when credit card transactions are involved. For this reason, a second `deposit` branch is written.

In the Bank class shown in Figure 6, two deposits take place. The first one deposits generic money into an account, and we expect that the first branch, `deposit(Money)`, will be chosen. For the second deposit, things are less clear. The second branch of `requestDeposit` is only compatible if the second argument value subtypes `CreditCard`. There are two approaches one can take.

First, any value passed to the second argument of `requestDeposit` must be a subtype of `Money`, so the first branch will always be compatible. During type checking, the first method is statically chosen for dispatch since we can always guarantee the types will be compatible. It is called single dispatch because the compiler or type checker only looks at the type of the target object at run-time.

Second, we can dispatch the first branch of `deposit` for the first deposit, and the second branch for the second deposit. This requires a run-time check for type compatibility of the second argument when `requestDeposit` is invoked since it is not known before execution. Multiple dispatch involves checking the type of all arguments at execution time. Safely facilitating such behavior is complicated, and is the subject of this paper.

1.3 Multiple Dispatch

As mentioned earlier, multiple dispatch differs from single dispatch in that the choice of the method to dispatch is influenced by all of the types of arguments. A number of languages already employ multiple dispatch, and significant research has been conducted [3, 1, 4].

2 Formalisms and Design

We have designed and implemented a typed λ -calculus called MultiCruz with structure types, subtyping, and methods. Our approach employs constructs from Castagna’s $\lambda&$ -calculus [1] for multiple dispatch. The only values in the language are numbers, booleans, structures, and methods. Procedure, structure, and method values cannot be expressed literally by the programmer.

There are only a few built-in types in MultiCruz, **Bool**, **Int**, and **Datum**. The rest are either: (a) function types $\mathbf{S} \rightarrow \mathbf{T}$ where the domain \mathbf{S} and range \mathbf{T} are also valid MultiCruz types; or (b) method types, defined as a list of function types; or (c) structure types. The valid types is more precisely defined by the grammar in Figure 8.

$$\begin{aligned}
v &::= n \\
&| p \\
&| (\text{procval } (\bar{x} : \bar{\mathbf{T}}) e E) \\
&| (\text{struct } \mathbf{T}_0 \mathbf{T}_1 \bar{v}) \\
&| (\text{method } ((\bar{x} : \bar{\mathbf{T}}) e E) \dots) \\
&| \text{EmptyMM}
\end{aligned}$$

Figure 7: A grammar representing the types of values in MultiCruz.

$$\begin{aligned}
\mathbf{T} &::= \mathbf{A} \\
&| \mathbf{X} \\
&| (\bar{\mathbf{T}} \rightarrow \mathbf{T}) \\
&| (\text{method } (\bar{\mathbf{T}} \rightarrow \mathbf{T}) \dots)
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{A} &::= \mathbf{Int} \\
&| \mathbf{Bool} \\
&| \mathbf{Datum} \\
&| \mathbf{Struct}
\end{aligned}$$

Figure 8: Two grammars. The first defines the set of valid types in MultiCruz $\mathcal{T}(\mathbf{T})$. A type \mathbf{S} is a valid type iff $\mathbf{S} \in (\mathbf{T})$. The second defines the set of primitive types.

$$\begin{aligned}
e &::= v \\
&| x \\
&| (\text{if } e_0 e_1 e_2) \\
&| (\text{join } m e) \\
&| (\text{app } e_0 e_1 \dots e_n) \\
&| (\text{of } e l) \\
&| (\text{new } x e_0 e_1 \dots e_n) \\
&| (\text{primapp } \text{prim } e_0 e_1 \dots e_n) \\
&| (\text{as } \mathbf{T} e) \\
&| (m \Rightarrow e_0 e_1 \dots e_n)
\end{aligned}$$

Figure 9: A grammar defining the abstract syntax of MultiCruz.

2.1 Syntax

The abstract syntax of a MultiCruz expression is defined by the grammar shown in Figure 2.1.

Following from it, a MultiCruz expression can be any of the following.

- **value**: either a numeric value, a boolean value, an instance of a structure, a lambda closure, or a method closure.
- **variable reference**: an identifier used to retrieve a stored value.
- **if**: conditional testing and branching. If e_0 evaluated to *true*, e_1 is evaluated. Otherwise, e_2 is evaluated.
- **join**: extends a method m is extended with a new branch e (evaluating to a λ -closure) with **join**.
- **app**: applies the function expression e_0 to the evaluation of the arguments e_1, e_2, \dots and e_3 .
- **of**: retrieves the field of the structure expression e named by n .
- **new**: constructs a new structure, initializing the ordered fields of the struct to their respective values.
- **primapp**: evaluates a primitive operator to the expressions. The primitive operators built-in in MultiCruz are **and**, **or**, **add**, **mul**, **eq1**, and **le**.
- **as**: casts the value of an expression e to a type \mathbf{T} .

Many of the expression constructs above operate on structure, method, and procedure values. The structure and method values are typed with definitions. To elaborate, a MultiCruz program consists of a list of definitions followed by an expression. The grammar defining the list of acceptable definitions is shown in Figure 10.

$$\text{program} ::= \text{defs} * e$$

Evaluation of a MultiCruz program is a four-step process. First, we check the validity of the syntax. Second, we evaluate the definitions to provide a structure environment used for type-checking and to form

$$\begin{aligned}
\text{defs} \quad ::= & \quad (\text{def} - \text{struct} \ \mathbf{T} \ (l_0 : \mathbf{T}_0) \ \dots) \\
& \quad | \quad (\text{def} - \text{struct} \ \mathbf{T} \ \text{ext} \ \mathbf{S} \ (l : \mathbf{T} \dots)) \\
& \quad | \quad (\text{def} \ l \ e) \\
& \quad | \quad (\text{defm} \ f \ (a_1 : t_1 \ a_2 : t_2 \ \dots \ a_n : t_n) : t \ e)
\end{aligned}$$

Figure 10: A grammar defining the syntax for defining structures, named assignments, and method definitions.

method values to be used for joining and dispatch. Third, we type-check the program. Fourth, the program is run if it was successfully typechecked.

The `def - struct` construct defines a structure type with a unique type name, a super-type name, and a list of field names with types. The names correspond to other structures that were defined previously. If the first form of `def - struct` is used, we assume the supertype is `Struct`, the eventual supertype of all structures.

The `def` construct is used to extend the store with an expression before compilation.

The `defm` construct defines a method with a name f . There are two possibilities here. First, a method under the name f might not have been defined yet. If this is the case, the method is an `EmptyMM`. Otherwise, we simply extend the method to include the new branch e .

2.2 Typing

Typing is not only the basis for ensuring the validity of MultiCruz programs, it is essential for dispatch. For this reason, we pay significant attention to the type-checking rules. We relied heavily on Pierce’s seminal book on typing to formulate many of our typing rules [5]. We also borrowed many concepts needed for multiple dispatch from Castanga[1] and Chambers[2].

2.2.1 Subtyping

We start with a few fundamental rules for subtyping. The subtyping relation on the set of types is denoted by $<:$. If a type \mathbf{S} subtypes another type \mathbf{T} , we write $\mathbf{S} <: \mathbf{T}$. Note again that the types are expressed with the language defined in the grammar in Figure 8. Throughout the discussion of typing, heavy use is made of argument list and field specifier definitions. Therefore, we have adopted the notation $(\bar{l} : \bar{\mathbf{T}})$ to denote a list of zero or more field specifiers of the form $(l_i : \mathbf{T}_i)$, and $(\bar{l} : \bar{\mathbf{T}})$ to denote zero or more argument specifiers of the form $l_i : \mathbf{T}_i$. Also, \bar{e} denotes zero or more expressions of the form e_i .

$$\begin{array}{c}
\text{TBase} \frac{}{\bar{\mathbf{T}} <: \mathbf{Datum}} \quad \text{TReflexive} \frac{}{\mathbf{T}_1 <: \mathbf{T}_2} \quad \text{TTrans} \frac{\mathbf{T}_1 <: \mathbf{T}_3 \quad \mathbf{T}_3 <: \mathbf{T}_2}{\mathbf{T}_1 <: \mathbf{T}_2}
\end{array} \tag{1}$$

All types subtype `Datum`. The subtyping relation is reflexive, transitive, and anti-symmetric. Next, given a structure value, the relationship between the type of the struct and its supertype is clearly defined by

$$\text{TSE} \frac{SE(\mathbf{T}_1) = (\mathbf{T}_1 \ \text{extends} \ \mathbf{T}_2 \ (\bar{l} : \bar{\mathbf{T}}))}{\mathbf{T}_1 <: \mathbf{T}_2}. \tag{2}$$

The most important subtyping rule for dispatch is

$$\text{TArgCompat} \frac{|\bar{\mathbf{S}}| = |\bar{\mathbf{T}}| = n \quad \forall i : 1 \leq i \leq n. \mathbf{S}_i <: \mathbf{T}_i}{\bar{\mathbf{S}} <: \bar{\mathbf{T}}}. \tag{3}$$

It ensures that one argument list is compatible with another. For example, given the Java method `foo` shown below,

```
void foo(String o);
void foo(Object o);
```

we know that any value sequence that is valid for the first method, is valid for the second method. Applying the argument compatibility rule, the first method subtypes the second. This subtyping rule allows us to make “is also valid for” determinations between method branches.

More formally,

$$\text{TFuncCompat} \frac{\overline{\mathbf{S}}_2 <: \overline{\mathbf{S}}_1 \quad \mathbf{T}_1 <: \mathbf{T}_2}{\overline{\mathbf{S}}_1 \rightarrow \mathbf{T}_1 <: \overline{\mathbf{S}}_2 \rightarrow \mathbf{T}_2}. \quad (4)$$

A more thorough treatment of function subtyping is discussed in Pierce but note that the rule is carefully formulated to ensure proper one-way compatibility between one function and another.

The basic rules for type checking are shown below.

$$\text{TVar} \frac{x : \mathbf{T} \in \mathbf{\Gamma}}{\mathbf{\Gamma} \vdash x : \mathbf{S}} \quad \text{TBool} \frac{}{\mathbf{\Gamma} \vdash p : \mathbf{Bool}} \quad \text{TNum} \frac{}{\mathbf{\Gamma} \vdash n : \mathbf{Int}} \quad (5)$$

Simply stated, we require that boolean values be of type **Bool**; numbers, of type **Int**; and variables, the type of the variable in the context $\mathbf{\Gamma}$.

The rule for subsumption is given below,

$$\text{TSubsumption} \frac{\mathbf{\Gamma} \vdash e : \mathbf{S} \quad \mathbf{S} <: \mathbf{T}}{\mathbf{\Gamma} \vdash e : \mathbf{T}} \quad (6)$$

It states that an expression that is of type **S** can also be of a type **T** if $\mathbf{S} <: \mathbf{T}$. Subsumption is essential for implicit casts. In Java, subsumption would be used to allow the following,

```
void takeObject(Object o) {}
void passString() { takeObject("hi"); }
```

The typing rule for `if` is standard,

$$\text{TIf} \frac{\mathbf{\Gamma} \vdash e_0 : \mathbf{Bool} \quad \mathbf{\Gamma} \vdash e_1 : \mathbf{T} \quad \mathbf{\Gamma} \vdash e_2 : \mathbf{T}}{\mathbf{\Gamma} \vdash (\text{if } e_0 \ e_1 \ e_2) : \mathbf{T}}. \quad (7)$$

Given a valid `lambda` expression, it is of a function type,

$$\text{TLambda} \frac{\mathbf{\Gamma}' = \mathbf{\Gamma}, \bar{x} : \overline{\mathbf{T}} \quad \mathbf{\Gamma}' \vdash e : \overline{\mathbf{T}}_1}{\mathbf{\Gamma} \vdash (\text{lambda } (\bar{x} : \overline{\mathbf{T}}_0) \ e) : \overline{\mathbf{T}}_0 \rightarrow \overline{\mathbf{T}}_1} \quad (8)$$

Application commands `app` require the input types to be compatible with the function input types,

$$\text{TApp} \frac{\mathbf{\Gamma} \vdash e_0 : \overline{\mathbf{T}}_1 \rightarrow \mathbf{T}_2 \quad \overline{\mathbf{T}}'_1 <: \overline{\mathbf{T}}_1 \quad \mathbf{\Gamma} \vdash \bar{e}_1 : \overline{\mathbf{T}}'_1}{\mathbf{\Gamma} \vdash (e \ \bar{e}) : \overline{\mathbf{T}}_2}. \quad (9)$$

A primitive application is only valid if the input types subtype the expected input types defined by the primitive in question. The expression is of a type that depends on the primitive used (e.g. `eq1` takes in **Int** and evaluates to **Bool**),

$$\text{TPrim} \frac{\mathbf{\Gamma} \vdash \text{prim} : \overline{\mathbf{T}} \rightarrow \mathbf{T}' \quad \mathbf{\Gamma} \vdash \bar{e} : \overline{\mathbf{T}}}{\mathbf{\Gamma} \vdash (\text{prim } \bar{e}) : \mathbf{T}'}. \quad (10)$$

When constructing an object of type \mathcal{ST} , all fields must be specified. The fields must be in the order they are defined in \bar{e} . Additionally, all the fields for the supertype structs must be defined. We use the function $\mathcal{F}(\mathcal{ST})$ to denote the fields of a type \mathcal{ST} . If \mathcal{ST} 's direct supertype is a **Struct**, then $\mathcal{F}(\mathcal{ST})$ is an order list of \mathcal{ST} 's fields. Otherwise, $\mathcal{F}(\mathcal{ST})$ is the concatenation of \mathcal{ST} 's fields with the $\mathcal{F}(\mathcal{ST}')$ where \mathcal{ST}' is the direct supertype of \mathcal{ST} .

More formally, we have

$$\text{Fields} \frac{}{\mathcal{F}(\text{Struct}, \sigma) \downarrow ()} \quad (11)$$

$$\text{Fields} \frac{E_s(Y) = (Y \ X \ (l_y : \mathbf{T}_y) \dots) \quad \mathcal{F}(X, \sigma) = ((l_x : \mathbf{T}_x) \dots)}{\mathcal{F}(Y, \sigma) = ((l_x : \mathbf{T}_x) \dots (l_y : \mathbf{T}_y) \dots)}, \quad (12)$$

and

$$\text{TNew} \frac{\mathcal{F}(\mathcal{ST}) = ((\bar{l} : \bar{\mathbf{T}})) \quad \Gamma \vdash \bar{e} : \bar{\mathbf{T}}}{\Gamma \vdash (\text{new } \mathcal{ST} \ \bar{e}) : \mathcal{ST}}. \quad (13)$$

The typing rule for the structure element accessor function `of` ensures that l_i is a valid structure element name. If so, the expression is of the type of that structure element. This is given by

$$\text{TOf} \frac{\mathcal{F}(\mathcal{ST}) = ((\bar{l} : \bar{\mathbf{T}})) \quad (l_i : \mathbf{T}_i) \in \mathcal{F}(\mathcal{ST}) \quad \Gamma \vdash e : \mathcal{ST}}{\Gamma \vdash (\text{of } e \ l_i) : \mathcal{ST}}. \quad (14)$$

Multicruz allows downcasts and upcasts. With downcasts, we assume *a priori* knowledge of the type of the expression at the time of type-checking.

$$\text{TUpCast} \frac{\Gamma \vdash e : \mathbf{S} \quad \mathbf{S} <: \mathbf{T} \quad \mathbf{S} \neq \mathbf{T}}{\Gamma \vdash (\text{as } \mathbf{T} \ e) : \mathbf{T}} \quad \text{TDownCast} \frac{\Gamma \vdash e : \mathbf{T} \quad \mathbf{S} <: \mathbf{T} \quad \mathbf{S} \neq \mathbf{T}}{\Gamma \vdash (\text{as } \mathbf{S} \ e) : \mathbf{S}} \quad (15)$$

If a downcast is performed or a cast between two unrelated types, the type-checker cannot guarantee anything so we use the `StupidCast` rule as described in [5].

$$\text{TStupidCast} \frac{\Gamma \vdash e : \mathbf{S} \quad \mathbf{S} \not<: \mathbf{T} \quad \mathbf{T} \not<: \mathbf{S}}{\Gamma \vdash (\text{as } \mathbf{T} \ e) : \mathbf{T}} \rightarrow \text{Generates Warning} \quad (16)$$

The most critical typing rules for multiple dispatch are the rules for `join` and `dispatch`. There must be exactly one choice for a branch during method dispatch, i.e. there must be no ambiguity. We first formally define ambiguity and unrelatedness. We then follow with a proof that two types that are unrelated cannot be ambiguous.

Notation: unrelated types We write $\bar{\mathbf{S}} \Delta \bar{\mathbf{T}}$ to say that $\bar{\mathbf{S}}$ and $\bar{\mathbf{T}}$ are unrelated.

Notation: number of arguments We write $|\bar{\mathbf{S}}|$ to denote the number of arguments for the argument type $\bar{\mathbf{S}}$.

Definition: ambiguous Two function argument type lists $\bar{\mathbf{S}}$ and $\bar{\mathbf{T}}$ are *ambiguous* if and only if neither subtypes the other and there exists a third argument type list $\bar{\mathbf{X}}$ such that $\bar{\mathbf{X}} <: \bar{\mathbf{S}}$ and $\bar{\mathbf{X}} <: \bar{\mathbf{T}}$.

Definition: unrelated argument list types Two function argument type lists $\bar{\mathbf{S}}$ and $\bar{\mathbf{T}}$ are *unrelated* if and only if $|\bar{\mathbf{X}}| \neq |\bar{\mathbf{Y}}|$ or $\exists i. (\bar{\mathbf{X}}_i \Delta \bar{\mathbf{Y}}_i)$.

Definition: unrelated non-argument list types Two types $\bar{\mathbf{S}}$ and $\bar{\mathbf{T}}$, both of which are not argument list types, are said to be unrelated if and only if neither $\mathbf{S} <: \mathbf{T}$ nor $\mathbf{T} <: \mathbf{S}$.

Given these definitions, a theorem follows.

Theorem: nonambiguity of unrelated argument list types Given two argument lists $\bar{\mathbf{S}}$ and $\bar{\mathbf{T}}$, if $\bar{\mathbf{S}}\Delta\bar{\mathbf{T}}$, then $\bar{\mathbf{S}}$ and $\bar{\mathbf{T}}$ are not ambiguous.

Proof We are given $\bar{\mathbf{S}}\Delta\bar{\mathbf{T}}$. There are two possible reasons for this:

- Case I. $|\bar{\mathbf{S}}| \neq |\bar{\mathbf{T}}|$. By the subtype rules, neither $\bar{\mathbf{S}} <: \bar{\mathbf{T}}$ nor $\bar{\mathbf{T}} <: \bar{\mathbf{S}}$. Further, no other type list $\bar{\mathbf{X}}$ where $\bar{\mathbf{X}} \neq \bar{\mathbf{S}}$ and $\bar{\mathbf{X}} \neq \bar{\mathbf{T}}$ can have $|\bar{\mathbf{X}}| = |\bar{\mathbf{S}}|$ and $|\bar{\mathbf{X}}| = |\bar{\mathbf{T}}|$.
- Case II. $\exists i$ such that $\mathbf{S}_i\Delta\mathbf{T}_i$. Assume $\bar{\mathbf{S}}$ and $\bar{\mathbf{T}}$ are ambiguous. For some other $\bar{\mathbf{X}}$ where $\bar{\mathbf{X}} \neq \bar{\mathbf{S}}$ and $\bar{\mathbf{X}} \neq \bar{\mathbf{T}}$, let $\bar{\mathbf{X}} <: \bar{\mathbf{S}}$ and $\bar{\mathbf{X}} <: \bar{\mathbf{T}}$. By the definition of subtyping on argument lists, we have $\mathbf{X}_i <: \mathbf{S}_i$ and $\mathbf{X}_i <: \mathbf{T}_i$. This is only possible if $\mathbf{S}_i <: \mathbf{T}_i$ or $\mathbf{T}_i <: \mathbf{S}_i$, so $\bar{\mathbf{X}}$ must not exist.

This enables a very powerful type-checking feature: the type-checker can generate an error for ambiguous dispatches. If the type-checker does not generate an error, it can guarantee a single, unique choice will exist for dispatch without knowing *a priori* the exact types of the arguments that will be passed at run-time. To summarize ambiguity, we give an example of an ambiguous method in Java,

```
class Bar {
  void foo(String s, Object o) {}
  void foo(Object o, String s) {}
  static void main(String args[]) {
    Bar bar = new Bar();
    bar.foo(new String(), new String());
  }
}
```

The MultiCruz type-checker will not allow such ambiguity.

We conclude our discussion of typing rules with the rules for join and dispatch.

The first expression to `join` must be a method, and the second expression, a procval. If the join is successful, the procval is incorporated into the method. Several conditions are imposed on the candidate procval to insert:

- its argument list type must be unique in the context of the other branches in the method, i.e. redefinitions are not allowed.
- its types must not break the rule that if the argument list of one branch subtypes any of the others, then the return types will subtype each other.
- for every two-fold combination of branches, either one branch subtypes the other, or they are unrelated.

More formally, we have

$$\text{TJoin} \frac{\Gamma \vdash e_0 : \{\bar{\mathbf{T}}_i \rightarrow \mathbf{U}_i \mid 0 \leq i < n\} \quad \Gamma \vdash e_1 : \bar{\mathbf{T}}_n \rightarrow \mathbf{U}_n \quad \Gamma \vdash \forall i : 0 \leq i < n. \bar{\mathbf{T}}_i \neq \bar{\mathbf{T}}_n \quad \Gamma \vdash \forall i, j : 0 \leq i, j \leq n. \bar{\mathbf{T}}_i <: \bar{\mathbf{T}}_j \text{ implies } \mathbf{U}_i <: \mathbf{U}_j \quad \Gamma \vdash \forall i, j : 0 \leq i, j \leq n. (\bar{\mathbf{T}}_i <: \bar{\mathbf{T}}_j \text{ or } \bar{\mathbf{T}}_j <: \bar{\mathbf{T}}_i \text{ or } \bar{\mathbf{T}}_i\Delta\bar{\mathbf{T}}_j)}{\Gamma \vdash (\text{join } e_0 e_1) : \{\bar{\mathbf{T}}_i \rightarrow \mathbf{U}_i \mid 0 \leq i \leq n\}}. \quad (17)$$

We finally define the typing rule for dispatch. Given a method e_0 and the argument expressions \bar{e}_1 , the dispatch is valid if the argument expressions evaluates to a compatible method. The greatest lower bound

$$\begin{aligned}
r ::= & \text{(if } p \ e_0 \ e_1) \\
& | \text{(lambda } (\bar{x} : \bar{\mathbf{T}}) \ e) \\
& | \text{(join (method } ((\bar{x} : \bar{\mathbf{X}})e \ \sigma) \dots) \text{ (procval } ((\bar{x} : \bar{\mathbf{X}})e \ \sigma) \dots))} \\
& | \text{(app (procval } ((\bar{x} : \bar{\mathbf{X}})e \ \sigma) \dots)\bar{v}) \\
& | \text{(primapp prim } v_0 \ \dots \ v_n) \\
& | \text{(new } \mathbf{X} \ v_0 \ \dots \ v_n) \\
& | \text{(of (struct } \mathbf{X}_0 \ \mathbf{Y}_0 \ v_0 \ \dots \ v_n) \ l) \\
& | \text{(as } \mathbf{T} \ v) \\
& | \text{(} m \Rightarrow e_0 \ e_1 \dots) \\
& | \text{(defn } \dots e) \\
& | \text{(def } x \ e) \\
& | \text{(def } x : \mathbf{T}e)
\end{aligned}$$

Figure 11: A grammar defining the redexes for the MultiCruz language. The third to last redex expresses a list of definitions followed by an expression. The second to last redex defines a store extension. The last redex is used for method definitions.

of the relation over the set of argument list types in the method e_0 determines the type of the dispatch expression,

$$\text{TDispatch} \frac{\Gamma \vdash \{\bar{\mathbf{T}}_i \rightarrow \mathbf{U}_i\}_{0 \leq i \leq n} \quad \Gamma \vdash \bar{e}_1 : \bar{\mathbf{T}} \quad \Gamma \vdash \bar{\mathbf{T}}_j = \min \{\bar{\mathbf{T}}_i | \bar{\mathbf{T}} <: \bar{\mathbf{T}}_i\}}{\Gamma \vdash (e_0 \Rightarrow \bar{e}_1) : \mathbf{U}_i}. \quad (18)$$

This completes the formalization of the typing rules for MultiCruz.

3 Small-step Semantics

We formalize the evaluation of programs in MultiCruz by defining small-step semantics rules. Our redexes are given by the grammar shown in Figure 11.

Upon checking MultiCruz input programs for validity, MultiCruz will reduce the `defm` and `def-struct` expressions to a generic `def` form; method definitions are tagged with user-specific types to permit recursion. Without such tags, a recursive program will be rejected by the type checker.

Next, we have the evaluation contexts as shown in Figure 12 and the global evaluation rules defined in Figure 13.

The evaluation rules for `if` are given by

$$\text{SIfTrue} \frac{}{\langle \langle \text{if true } e_0 \ e_1 \rangle, E, \sigma, \mu \rangle \rightarrow e_0} \quad \text{SIfFalse} \frac{}{\langle \langle \text{if false } e_0 \ e_1 \rangle, E, \sigma, \mu \rangle \rightarrow e_1}. \quad (19)$$

As expected lambda expressions evaluate to a lambda closure with the current state of the store. We have

$$\text{SLambda} \frac{}{\langle \langle \text{lambda } (\bar{x} : \bar{\mathbf{T}}_0) \ e \rangle, E, \sigma, \mu \rangle \rightarrow \langle \text{procval } (\bar{x} : \bar{\mathbf{T}}_0) \ e \ E \rangle}. \quad (20)$$

$$\begin{aligned}
H ::= & \text{(if } H e_0 e_1) \\
& | \text{(join } H e) \\
& | \text{(join } v H) \\
& | \text{(app } v \dots H e \dots) \\
& | \text{(} H \Rightarrow e_0 e_1 \dots) \\
& | \text{(} v \Rightarrow v \dots H e \dots) \\
& | \text{(primapp prim } H e) \\
& | \text{(primapp prim } v H) \\
& | \text{(of } H l) \\
& | \text{(new } X \dots H e \dots) \\
& | \text{(as } TH) \\
& | (H) \\
& | (H\text{defn}\dots e)
\end{aligned}$$

Figure 12: A grammar defining the evaluation contexts for the small-step semantics for MultiCruz.

$$\begin{aligned}
H[e] \rightarrow H[e'] & \quad \text{iff } e \rightarrow e' \\
H[e] \rightarrow \mathbf{error} & \quad \text{iff } e \rightarrow \mathbf{error}
\end{aligned}$$

Figure 13: The global evaluation rules for MultiCruz.

Following the evaluation of a lambda expression, at some point, it may be applied. The store embedded in the lambda closure is extended with the values of the arguments. The body is then evaluated with the extended store. This gives the following for application:

$$\text{SApp} \frac{E'_B = E_B[x_0 \mapsto v_0][x_1 \mapsto v_1] \dots [x_n \mapsto v_n]}{\langle (\text{app} (\text{procval } (\bar{x} : \bar{\mathbf{T}} e E_B) v_0 v_1 \dots v_n)), E, \sigma, \mu \rangle \rightarrow \langle e, E'_B, \sigma, \mu \rangle}. \quad (21)$$

The `new` operator constructs a new struct value,

$$\text{SNew} \frac{\sigma(\mathbf{X}) = (\mathbf{X} \mathbf{Y}(l_0 : \mathbf{T}_0 \dots l_n : \mathbf{T}_n))}{\langle (\text{new } \mathbf{X} v_0 \dots v_n), E, \sigma, \mu \rangle \rightarrow \langle \text{struct } \mathbf{X} \mathbf{Y} v_0 \dots v_n \rangle}. \quad (22)$$

The `of` operator retrieves a name field from a struct value,

$$\text{SOf} \frac{\mathcal{F}(\mathbf{X}) = ((l_0 : \mathbf{T}_0) \dots (l_n : \mathbf{T}_n)) \quad l_i = l}{\langle (\text{of} (\text{struct } \mathbf{X} \mathbf{Y} v_0 \dots v_n) l), E, \sigma, \mu \rangle \rightarrow v_i}. \quad (23)$$

The type-checker has a more complicated responsibility when processing `join`. This simplifies the operation semantics, the new branch is incorporated into the existing method,

$$\text{SJoin} \frac{}{\langle (\text{join} (\text{method } ((\bar{x} : \bar{\mathbf{T}}_0) e_0 E_0) \dots) (\text{procval } (\bar{x} : \bar{\mathbf{T}}_0) e_n E_n)), E, \sigma, \mu \rangle \rightarrow \langle \text{method } ((\bar{x} : \bar{\mathbf{T}}_0) e_0 E_0) \dots ((\bar{x} : \bar{\mathbf{T}}_0) e_n E_n) \rangle}. \quad (24)$$

The small-step semantic rule for dispatch is trickier. The greatest lower bound branch is applied to the input arguments based on their types,

$$\text{SDispatch} \frac{\bar{v} : \bar{\mathbf{V}} \quad \bar{\mathbf{T}}_j = \min\{\bar{\mathbf{T}}_i | \bar{\mathbf{V}} <: \bar{\mathbf{T}}_i\} \quad E'_j = E_j[x_0 \mapsto v_0][x_1 \mapsto v_1] \dots [x_n \mapsto v_n]}{\langle (\text{method } ((\bar{x}_0 : \bar{\mathbf{T}}_0) e_0 E_0) \dots ((\bar{x}_n : \bar{\mathbf{T}}_n) e_n E_n)), E, \sigma, \mu \rangle \rightarrow \langle e_j, E'_j, \sigma, \mu \rangle}. \quad (25)$$

The `def` operator is used to define variables or methods,

$$\text{SVarDef} \frac{}{\langle (\text{def } x \text{ EmptyMM}), E, \sigma, \mu \rangle \rightarrow \langle E, \sigma, \mu \rangle} \quad \text{SMethodDef} \frac{v = (\text{method } f_0 f_1 \dots) \quad \mu' = \mu[x \mapsto v]}{\langle (\text{def } x : \mathbf{T} v), E, \sigma, \mu \rangle \rightarrow \langle E, \sigma, \mu' \rangle}. \quad (26)$$

If the input value to define is not a method, we have

$$\text{SVarDefNonMethod} \frac{v \neq \text{EmptyMM}}{\langle (\text{def } x v), E, \sigma, \mu \rangle \rightarrow \langle E[x \mapsto v], \sigma, \mu \rangle}. \quad (27)$$

The rules for variable lookups must account for the fact that methods and non-methods are stored. Thus, we give

$$\text{SVarLookup} \frac{E(x) \neq \text{false} \quad v = E(x)}{\langle (\text{var } x), E, \sigma, \mu \rangle \rightarrow v} \quad (28)$$

and

$$\text{SVarLookup} \frac{E(x) = \text{false} \quad \mu(x) \neq \text{false} \quad v = \mu(x)}{\langle (\text{var } x), E, \sigma, \mu \rangle \rightarrow v}. \quad (29)$$

If a variable cannot be found in the store, an error is generated,

$$\text{SVarLookupErr} \frac{E(x) = \text{false} \quad \mu(x) = \text{false}}{\langle (\text{var } x), E, \sigma, \mu \rangle \rightarrow \text{Generates Error}}. \quad (30)$$

4 Implementation Details

MultiCruz runs on the Petite Chez Scheme interpreter available from <http://www.scheme.com/>. The Indiana `match.ss` script was used heavily for pattern matching. Our test harness made use of Scheme macros. This gave us significant expressive power when implementing MultiCruz. Before expansion, MultiCruz is 3280 lines of code, and after expansion, 23488 lines.

5 Conclusion

We have formalized a variant of the typed λ -calculus that supports nominal typing, structure types, methods, and multiple dispatch. We have implemented the MultiCruz type-checker and interpreter in Scheme. The MultiCruz type-checker can verify safe method dispatch of a unique branch.

6 Source Code

The source code for MultiCruz is available at <http://www.cs.ucsc.edu/~eads/MultiCruz>. To run MultiCruz, type the following at the command line:

```
petite -q run.ss
```

We will consider writing a tutorial for MultiCruz in the near future.

References

- [1] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 182–192, New York, NY, USA, 1992. ACM Press.
- [2] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst.*, 17(6):805–843, 1995.
- [3] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Modular open classes and symmetric multiple dispatch for java. *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2000.
- [4] Neal Glew. Type dispatch for named hierarchical types. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 172–182, New York, NY, USA, 1999. ACM Press.
- [5] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

A Another Introductory Java Example

```
1 public class Mood {
2
3     private static java.util.Random rand = new java.util.Random();
4
5     public static void main(String args[]) {
6         Driver d = new Driver();
7         Vehicle v = makeRandomVehicle();
8         System.err.println(v.toString());
9         d.drive(v);
10    }
11
12    public static Vehicle makeRandomVehicle() {
13        return (rand.nextBoolean()) ? (Vehicle)new Airplane(30000, 500) :
14            (Vehicle)new Car(3000, 100);
15    }
16
17 }
```

```
1 public class Driver {
2
3     void drive(Vehicle v) {
4         v.turnWheel();
5     }
6
7     void drive(Airplane a) {
8         a.adjustFlaps();
9         a.adjustAilerons();
10        a.adjustThrottle();
11        a.turnWheel();
12    }
13 }
```

```
1 public class Vehicle {
2
3     private int weight;
4
5     public Vehicle(int weight, int speed) {
6         this.weight = weight;
7     }
8
9     int getWeight() {
10        return weight;
11    }
12
13    void turnWheel() {
14        System.out.println("Turning the wheel.");
15    }
16 }
```

```

1 public class Airplane extends Vehicle {
2
3     public Airplane(int weight, int speed) {super(weight, speed);}
4
5     void adjustFlaps() {
6         System.err.println("Adjusting_flaps.");
7     }
8
9     void adjustAilerons() {
10        System.err.println("Adjusting_ailerons.");
11    }
12
13    void adjustThrottle() {
14        System.err.println("Adjusting_throttle.");
15    }
16 }

```

```

1 class Car extends Vehicle {
2
3     Car(int weight, int speed) {super(weight, speed);}
4
5 }

```

B Multiple Dispatch: A Matter of Life and Death

B.1 Java

```

1 class DogShow {
2
3     public static void main(String args[]) {
4         Person fred = new Person();
5         Dog spot = new Dog();
6         MeanDog butch = new MeanDog();
7         introduce(fred, spot);
8         introduce(fred, butch);
9     }
10
11    static void introduce(Person a, Dog b) {
12        a.pet(b);
13    }
14 }

```

```

1 class Person {
2
3     void pet(Dog d) {
4         System.out.println("Nice_doggie!");
5         d.react();
6     }
7 }

```

```

8     void pet(MeanDog d) {
9         System.out.println("No_way!_He'll_bite!");
10    }
11 }

```

```

1 class Dog {
2
3     void react() {
4         System.out.println("***_Wags_tail..._pant!_***");
5     }
6
7 }

```

```

1 class MeanDog extends Dog {
2
3     void react() {
4         System.out.println("Grrr._*chomp*");
5     }
6
7 }

```

B.2 Multiple Dispatch Doesn't Bite!

```

1 (defstruct Person ())
2 (defstruct Dog ())
3 (defstruct MeanDog extends Dog (bigTeeth? : Bool))
4
5 (defm (reaction of d : Dog) : Int
6     10)                                     ;; Positive!
7 (defm (reaction of d : MeanDog) : Int
8     (if (of d bigTeeth?)
9         100                                 ;; Very negative!
10        10))                               ;; Little teeth...bad but not as bad.
11
12 (defm (pet p : Person d : Dog) : Int
13     (reaction of => d))                   ;; OK, we'll pet it...
14
15 (defm (pet p : Person d : MeanDog) : Int
16     0)                                     ;; Not goin' near that thing!
17
18 (defm (introduce p : Person d : Dog) : Int
19     (pet => p d))
20
21 (def fred (new Person))
22
23 (def spot (new Dog))
24 (def killer (new MeanDog true))
25
26 (def ans1 (introduce => fred spot))
27 (def ans2 (introduce => fred killer))

```

```
28 |
29 | (and (eql ans1 10) (eql ans2 0))
```

C Factorial Program in MultiCruz

```
1 ;; Sample program: factorial.
2
3 (defm (fact n : Int) : Int
4   (if (eql n 0) 1 (mul n (fact => (add n (mul 1 1))))))
5
6 (def a1 (fact => 1))
7 (def a2 (fact => 2))
8 (def a3 (fact => 3))
9 (def a4 (fact => 4))
10 (def a5 (fact => 5))
11
12 (and (eql a1 1)
13      (and (eql a2 2)
14            (and (eql a3 6)
15                  (and (eql a4 24) (eql a5 120)))))
16 ; s.b. (bool true)
```

D List Processing in MultiCruz

```
1 ;; Sample program: list length
2
3 ;; Data defn:
4 (defstruct List ())
5 (defstruct Empty extends List ())
6 (defstruct Cons extends List (first : Datum
7                                rest : List))
8
9 ;; Methods:
10 (defm (len ls : List) : Int 1) ;; shouldn't be called
11 (defm (len ls : Empty) : Int 0)
12 (defm (len ls : Cons) : Int
13   (add 1 (len => (of ls rest))))
14
15 ;; Try some casts, too:
16 (defm (sum ls : List) : Int ;; shouldn't be called
17   (as Int false))
18 (defm (sum ls : Empty) : Int 0)
19 (defm (sum ls : Cons) : Int
20   (add (as Int (of ls first))
21         (sum => (of ls rest))))
22
23 (def E (new Empty))
24 (def ls1 (new Cons 5 (new Cons 6 (new Cons 7 (new Cons 8 E)))))
25 (def ls2 (new Cons 2 (new Cons 2 (new Cons 2 (new Cons 2 E)))))
26
```

```
27 (def a1 (len => ls1))
28 (def a2 (len => (new Cons true (new Cons false E))))
29 (def a3 (len => E))
30
31 (def a4 (sum => E)) ;0
32 (def a5 (sum => ls1)) ; 26
33 (def a6 (sum => ls2)) ; 8
34
35 (and (eq1 a1 4)
36      (and (eq1 a2 2)
37            (and (eq1 a3 0)
38                  (and (eq1 a4 0)
39                        (and (eq1 a5 26) (eq1 a6 8)))))))
```