Fine-Grained Interoperability through Mirrors and Contracts

Kathryn E. Gray
University of Utah
kathyg@cs.utah.edu

Robert Bruce Findler
University of Chicago
robby@cs.uchicago.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

ABSTRACT

As a value flows across the boundary between interoperating languages, it must be checked and converted to fit the types and representations of the target language. For simple forms of data, the checks and coercions can be immediate; for higher order data, such as functions and objects, some must be delayed until the value is used in a particular way. Typically, these coercions and checks are implemented by an ad-hoc mixture of wrappers, reflection, and dynamic predicates. We observe that 1) the wrapper and reflection operations fit the profile of mirrors, 2) the checks correspond to contracts, and 3) the timing and shape of mirror operations coincide with the timing and shape of contract operations. Based on these insights, we present a new model of interoperability that builds on the ideas of mirrors and contracts, and we describe an interoperable implementation of Java and Scheme that is guided by the model.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

Interoperability, mirrors, contracts, Java, Scheme

1. INTRODUCTION

No single programming language is best for all tasks. Even within a single application, different parts of a large program might be best implemented with different kinds of languages. Some parts might be best implemented with objects in a statically typed language, for example, while other parts might be best implemented with functions in a dynamically typed language.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

Operating systems and virtual machines easily accommodate multiple programming languages across multiple applications. The limiting factor for mixing languages within a single application, however, is the granularity at which the languages can interoperate. A byte stream or a COM-style component model, for example, supports only coarse-grained interoperability, because all data exchanged between languages must be explicitly marshaled to a universal data format. We seek more fine-grained support, where the values of each language can be used directly in the other language.

In this paper, we report on a design and an implementation for fine-grained interoperability between a statically typed, object-oriented language and a dynamically typed, functional language. Our experiment is based specifically on Java [21] and PLT Scheme [16], but we describe an approach to interoperability that is essentially independent of the two languages, and that can be understood as a combination of mirrors [9] and contracts [13, 15].

In general, our approach requires extending both languages. We make the languages' dynamic semantics match by adding datatypes and constructs that enable each language to express operations central to the other language. In particular, we add a notion of objects and exceptions to Scheme, and we extend Java with a notion of functions as values and with continuations. To make the static semantics of the two languages match, we add a notion of dynamically typed expressions to Java (forcing a static type system on Scheme would defeat the point of interoperability). This change makes Java more compatible with Scheme, and also compatible with languages like Smalltalk and Python. Furthermore, our approach to dynamic typing within Java should work for other statically typed languages.

Given more than a few programming languages, we cannot expect to modify all the languages to include constructs from all the others. Our goal is *not* to define a universal interface that must cater to a least-common denominator. Instead, we explore the potential for direct interoperability between specific languages.

Dynamic Typing in Java

Java supports limited dynamic typing through a catch-all `Object` type and run-time casts. These constructs cannot support more general dynamic operations, including dynamic method calls as found in languages like Smalltalk [20] and Python [36]. In Java, a method invocation is always tied to the static type of the object; this type must be known to contain the method, with the proper signature. In Smalltalk, an object is an instance of a particular class, but

methods are not statically associated with classes. Therefore, if the class of an object contains the method, it can always be called. Similarly, primitive operations in Java, such as `+`, are statically resolved to operations on specific types, such as `int` or `double`, but in Python and Scheme, such primitive operations must dispatch based on the run-time arguments.

To support Smalltalk and Python-like method dispatch in Java, we must add a new form of method invocation. Instead of adding special syntax for such calls, however, we choose to designate certain expressions as dynamically typed, and then treat method calls with dynamic targets as the new kind of method call. This naturally extends to treating a `+` expression with dynamic sub-expressions as a dynamically typed addition.

Concretely, we add a `dynamic` keyword for bindings. We use `dynamic` in the type position for a binding, but as a type, `dynamic` serves as little more than an alias for `Object`. The more important role of `dynamic` is its effect on method calls and other primitive operations. Consider the following example.

```
Food getFavoriteFood(dynamic fish, Food[] kinds) {
  if (fish.hasFavorite())
    return new Food(fish.getFavorite());
  else
    return fish.chooseFavorite(kinds);
}
```

The `dynamic` keyword indicates that uses of `fish` are dynamically typed. In particular, the use `fish.hasFavorite()` does not rely on declaring any class or interface that contains the `hasFavorite` method. Instead, the object is inspected at run-time to determine whether it implements a `hasFavorite` method of no arguments. If so, the method is called. Meanwhile, the expressions `fish.getFavorite()` and `fish.chooseFavorite(kinds)` imply separate checks for `getFavorite` and `chooseFavorite` methods, but only if those expressions are evaluated. In fact, there may exist no class that includes all of the methods `hasFavorite`, `getFavorite`, and `chooseFavorite`.

One possible implementation of `dynamic` is to use the Java reflection API. Indeed, reflection is a popular way to implement interoperability between Java and many other languages. Building on reflection has a number of drawbacks, however. It interferes with analysis, constrains the compiler, and requires extra run-time support that may go unused.

In contrast, an explicit `dynamic` form gives analysis tools and the compiler information about where dynamic features are used and how they are used. This information allows analysis tools to be more precise, and it gives the compiler writer more choice in implementations. For example, the compiler might implement `dynamic` bindings with a different kind of value than `Object` (inserting coercions along with dynamic checks), which might limit the impact of dynamic operations on the run-time system.

This insight is exactly the motivation behind *mirrors* as a model for reflection [9], instead of building Java-style reflective operations into a language's core. Our `dynamic` form can be understood partly in terms of mirrors.

Mirrors and Reflection

Support for reflection is usually implemented with a combination of virtual-machine support and a reflection API, where reflection support is built into every declared class. A mirror-based implementation of reflection, in contrast, supports reflection operations separatly from each declared class.

One way to implement mirror-based reflection is to generate a mirror class for each declared class. An instance of the mirror class embeds the base objects, and it contains all information that is needed to support reflection.

For example, consider the following `Food` class.

```
class Food {
  Integer amount;
}
```

In a Java-style reflective system, the `Food` class automatically includes methods to support reflection operations. With mirror-based reflection, the `Food` class contains no such methods, and reflection operations are instead supplied by a class like `FoodMirror`:

```
class FoodMirror extends ClassMirror {
  Food f;
  Object getField( String name ) {
    if (name.equals("amount"))
      return f.amount;
    ...
  }
}
```

A mirror encapsulates all reflection operations, it is separate from the base object, and its structure reliably reflects the structure of the core object [9]. All of these properties are needed for implementing `dynamic`, but we need additional facilities for checking types.

Static Typing with Dynamic Operations

Beyond disambiguating method calls and primitive operations, Java's type system ensures that primitive operations are not misapplied. In order to preserve the type system's guarantees for dynamically typed objects, we must protect operations from misuse by adding suitable checks around dynamic expressions.

For example, in the program

```
class Food {
  int amount;
  Date expiration;

  boolean isSuitable(dynamic fish) {
    return fish.canEat(amount)
           && (new Date()).before(expiration);
  }
}
```

the type system requires that the call `fish.canEat(amount)` produce a boolean. Thus, dynamic checks must not only ensure that `fish` contains a `canEat` method of one argument, but also that an integer is allowed as the argument, and that the result is a boolean.

In this case, checks inserted durring compilation around the `fish.canEat` call ensure that the type system's assumptions are validated at run time. When objects are passed into a dynamic method or returned as a dynamic result, immediate checks are insufficient for guaranteeing type safety. Roughly, objects that are passed into a dynamic context must be wrapped to ensure that, for subsequent method invocations, the arguments are consistent with the method's declared types. The methods may, in turn, involve `dynamic` arguments or result, so that a chain of wrappers may be necessary.

Adding `dynamic` to Java thus requires a complex array of checks and coercions to be inserted by the compiler, but we can make sense of these checks by building on our previous work for *contracts* [13, 15]. Contracts introduce run-time checks on the uses of values, including higher-order functions in Scheme.

Contracts in a Dynamic Language

In a language like Scheme, arguments to primitive operators are always checked dynamically, so the language is safe even without type declarations. Sometimes, however, a programmer would like to enforce type-like constraints that raise an error before the program misapplies operators, providing better error locality and a clear report of the fault.

For example, suppose a Scheme library includes the following function.

```
;; save-fish : integer string -> void
(define (save-fish fish-id data)
  (write data (open-output-file
               (format "fishes/~a.txt" fish-id))))
```

The comment in the first line indicates that this function is intended to be used with an integer as the first argument. The integer is used to generate a filename using the `sprintf`-like `format` function, where the `"~a"` format directive converts any kind of value to a string.

If a client calls `save-fish` with a value other than an integer, the resulting filename might be invalid, or it might access files that are intended to be private. For example, someone might call `save-fish` with `"../../passwords"` as the second argument.

To protect against such problems, the Scheme programmer can insert an explicit check for `safe-fish`'s argument.

```
(define (save-fish-checked fish-id data)
  (check integer? fish-id)
  (save-fish fish-id data))
;; export save-fish-checked to clients as save-fish
```

As we saw with Java, sometimes no immediate check suffices. For example, in

```
;; load-interactive : (-> integer) -> string
(define (load-interactive get-id)
  (let ([fish-id (get-id)])
    (read (open-input-file
           (format "fishes/~a.txt" fish-id)))))
```

the implementor of `load-interactive` not only insists that `get-id` is a function, but also that the function produces an integer. Simply checking `(procedure? get-id)` does not ensure that the function `get-id` produces an integer, and a `will-return-integer?` function cannot be implemented in general (without solving the halting problem). Thus, `load-interactive` must check the result of `get-id` before proceeding. To avoid cluttering `load-interactive` with checks, the checks can be applied by wrapping a given `get-id` with a checked version.

```
(define (load-interactive-checked get-id)
  (check procedure? id)
  (define (checked-get-id)
    (let ([id (get-id)])
      (check integer? id)
      id))
  (load-interactive checked-get-id))
;; export load-interactive-checked to clients
;; as load-interactive
```

Clearly, implementing these checks is tedious, and the checks can be implemented in different ways that might expose different optimizations to the compiler (e.g., eliminating redundant checks) or might allow better error messages.

For these reasons, PLT Scheme programmers do not write checks directly, but instead use a construct similiar to an `apply-contract` form.

```
(define save-fish-contract
  (integer string . -> . void))
(define save-fish-checked
  (apply-contract save-fish-contract
                  save-fish))

(define load-interactive-contract
  ((-> integer) . -> . string))
(define load-interactive-checked
    (apply-contract load-interactive-contract
                    load-interactive))
```

Contracts more clearly express the intended checks, and the compiler ensures that the intended checks are actually applied (in an efficient way, and with clear error messages). In other words, after declaring contracts, the programmer can think of the function arguments as having the declared "types," although the type requirements are enforced dynamically.

Contracts and Static Types

The Scheme-oriented view of contracts extends naturally to object-oriented programming [15]. Requiring explicit contracts to support fine-grained interoperability places too much burden on the programmer, as many checks will be needed to ensure safety. Further, for type-safety to be maintained, no contract can be forgotten, which we cannot expect of programmers.

Therefore, we exploit Java's type system to reliably infer and insert all necessary contracts automatically. For example, as we saw before, in the method

```
boolean isSuitable(dynamic fish) {
  return fish.canEat(amount)
         && (new Date()).before(expiration);
}
```

the type system requires that the call `fish.canEat(amount)` produces a boolean, etc. We can express all of these type requirements as an interface-like contract.

```
contract CanEatContract { boolean canEat(int amt); }
```

The method name, argument type, and return type in this contract can all be inferred from the context of the call `fish.canEat(amount)`. Given this contract, a compiler could expand the original method to include an explicit contract application, as follows.

```
boolean isSuitable(dynamic fishOrig) {
  CanEatContract fish = applyContract(CanEatContract,
                                      fishOrig);
  return fish.canEat(amount)
         && (new Date()).before(expiration);
}
```

We explore the details of such compilation more precisely in Section 2.

Roadmap

Interoperability with Scheme motivates adding `dynamic` into Java, and it inspires our use of contract annotations to preserve type safety, but `dynamic` in Java is useful even without Scheme interoperability.

Meanwhile, making Java integrate smoothly with Scheme involves many issues besides dynamic expressions in Java,

and our detailed implementation experience is closely tied to interoperability with Scheme.

We therefore stage our presentation in four layers:

- Section 2 sketches an implementation of `dynamic` by converting a Java program with `dynamic` into a plain Java program with explicit mirror classes. This implementation sketch provides a flavor of the coercions and checks that are introduced by a contract system.
- Section 3 considers an application that is implemented with a combination of Java and Scheme. The example highlights how interoperability can be understood in terms of contracts, and it demonstrates how contracts provide a useful common vocabulary for dynamic expressions across languages.
- Section 4 discusses design details specific to making Scheme and Java interoperate smoothly, such as how Java classes are mapped to specific module and class constructs in PLT Scheme.
- Section 5 discusses the implementation of the design from Section 4. Our implementation [24] compiles Java to PLT Scheme, so that we can exploit all of our existing infrastructure for programming environments [14].

2. IMPLEMENTING DYNAMIC EXPRESSIONS

A compiler for Java with `dynamic` must choose a uniform representation for `dynamic` values. One possible implementation is a conversion to plain Java: use a `Mirror` interface in place of `dynamic`, and create a `Mirror` implementation for each class, interface, and primitive type in the original program (if its instances might flow to a dynamic expression).

2.1 Adding Mirrors

Returning to our example,

```
class Food {
  int amount;
  Date expiration;

  boolean isSuitable(dynamic fish) {
    return fish.canEat(amount)
           && (new Date()).before(expiration);
  }
}
```

the converted signature for the `isSuitable` method using `Mirror` is

```
boolean isSuitable(Mirror fish)
```

If an expression contains a call of `isSuitable` with a non-`dynamic` argument,

```
food.isSuitable(new Shark())
```

the compiler must insert a coercion to the argument's mirror.

```
food.isSuitable(new SharkMirror(new Shark()))
```

Given a `Shark` declaration

```
class Shark {
  ....
  boolean canEat(int amt) { .... }
}
```

the compiler generates a mirror class.

```
class SharkMirror implements Mirror {
  final Shark orig;
  SharkMirror(Shark _orig) { orig = _orig; }
  ...
}
```

The main operation that the compiler needs on the `Mirror` interface is `call`, which dynamically locates a method by name and argument count, and then applies an array of `Mirror` values to produce a `Mirror` result. Thus, the implementation of `SharkMirror` includes a `call` method as follows.

```
class SharkMirror implements Mirror {
  ...
  Mirror call(String name, Mirror[] args) {
    if (name.equals("canEat") && (args.length == 1)){
      ....
    } else
      raise new Error("method not understood");
  }
}
```

To implement the dynamic call, the `call` method of the class `SharkMirror` must unpack its arguments (raising an error if an argument does not have a suitable type), call the original method, and then pack the result as a `Mirror`.

```
Mirror call(String name, Mirror[] args) {
  if (name.equals("canEat") && (args.length == 1)){
    int amt;
    if (args[0] instanceOf IntegerMirror)
      amt = ((IntegerMirror)args).intVal();
    else
      raise new Error("bad argument");
    Boolean result = orig.canEat(amt);
    return new BooleanMirror(result);
   } else
    raise new Error("method not understood");
}
```

This `Mirror` implementation includes dynamic checks that are just like contract checks. The check that the argument is an integer is like the check in `save-fish-checked`. The coercion of `result` to a mirror is much like wrapping `get-id` with `check-get-id`; it can be viewed as applying the contract "must be used as an integer" to the result of the method.

These contracts are driven by the `Shark` class declaration. The `canEat` method is declared to accept a single `int` argument and return a `boolean` result, so the `"canEat"` case in `SharkMirror`'s `call` checks for an integer and wraps its result as a boolean.

2.2 Adding Unmirrors

In our Scheme examples, only definitions acquire contracts. In Java with `dynamic`, variable declarations and dynamic expressions both acquire contracts. For example, the call `fish.canEat(amount)` implies a contract (which we named `CanEatContract` from the Introduction) on `fish`.

Taking representation issues into account, the call to `canEat` must also coerce arguments to `Mirror`s and coerce the result from `Mirror`. Just as for declaration-side contracts and coercions, user-side contracts and coercions can be naturally packaged together in a compiler-generated `Unmirror` class, as follows.

```
....
  boolean isSuitable(Mirror fishOrig) {
    return (new CanEatUnmirror(fishOrig)).canEat(amount)
           && (new Date()).before(expiration);
  }
....
final class CanEatUnmirror {
  Mirror m;
  CanEatUnmirror (Mirror _m) { m = _m; }

  boolean canEat(int amt) {
    Mirror[] args = new Mirror[1];
    args[0] = new IntegerMirror(amt);

    /* Might raise a dynamic exception */
```

```
    Mirror result = m.call("canEat", args);

    if (result instanceOf BooleanMirror)
      return((BooleanMirror)args).booleanVal();
    else
      raise new Error("bad result");
  }
}
```

As in `SharkMirror`, the checks and coercions in the class `CanEatUnmirror` are based on the static types surrounding the `fish.canEat(amount)` call. Specifically, the argument is packaged using `IntegerMirror` because the type of the argument `amount` is `int`. The result is checked to be a `boolean` value because the result of `fish.canEat(amount)` is used as a `boolean`.

In general, inference for checks and coercions involves a straightforward traversal of a program's expression. Binding positions, return positions, and conditional-test positions impose a specific type on dynamic expressions, as do argument positions of normal Java method calls for non-overloaded methods. If the context of a dynamic expression does not impose a specific type, then it is treated as `dynamic`. Finally, for dynamically typed method calls, arguments with specific types impose requirements on the method.

The generated `CanEatUnmirror` class above has a `final` declaration to emphasize that it is completely under the control of the compiler, which might choose to eliminate the instance and inline its method call. An inlining optimization produces checks immediately around the dynamic `canEat` call, which is what a programmer would intuitively expect. In contrast, the `IntegerMirror` object cannot be eliminated in general; it encapsulates obligations for a dynamic implementation of `canEat`, to ensure that `canEat` uses the integer safely. This wrapper is consistent with the intuition that statically typed objects passed to dynamic code must be wrapped with checks to guard the object.

2.3 General Dynamic Expressions

Looking back at the expansion of `isSuitable`, we can see that it does not include any reference to `Shark`, because the original method includes no reference to `Shark`. Indeed, the method may be called with any kind of object, and it will succeed as long as the object's class provides a suitable `canEat` method. The object might, for example, be an instance of the following `Tuna` class.

```
class Tuna {
  dynamic size;
  boolean finicky = false;

  dynamic canEat(dynamic food) {
    if (finicky)
      return food.isTasty();
    else
      return food < size;
  }
}
```

The `canEat` method for the `Tuna` class potentially uses its argument as a kind of food, instead of an amount of food. This method should nevertheless work as an argument to `Food`'s `isSuitable` method, as long as the `Tuna` instance is not finicky, because this `canEat`'s argument is declared `dynamic`. If the tuna is finicky, however, the guarded code detects the error and throws an exception.

The delayed checking of `canEat`'s argument is reflected in its `Mirror` by a lack of checking or coercion of the arguments in the `call` method.

```
class TunaMirror implements Mirror {
  Tuna orig;
  ...
  Mirror call(String name, Mirror[] args) {
    if (name.equals("canEat") && (args.length == 1)) {
      return orig.canEat(args[0]);
    } else ....
    } else
      raise new Error("method not understood");
  }
}
```

The implementation also has no checking or coercion for the result of `canEat`, because the original return type is declared `dynamic`.

This strategy for dynamic method calls extends naturally to other kinds of dynamic expressions, such as binding a dynamic value to a statically typed variable, or adding a `double` to a dynamic expression. For example, given the interface

```
interface OceanFish { double weight(); }
```

the method

```
double totalWeight(dynamic fish, dynamic waterWt) {
  OceanFish cf = fish;
  return cf.weight() + waterWt;
}
```

expands as

```
double totalWeight(Mirror fish, Mirror waterWt) {
  OceanFish cf =
       (new OceanFishUnmirror(fish)).val();
  return cf.weight()
         + new DoubleUnmirror(waterWt).val();
}
....
class OceanFishUnmirror {
  Mirror m;
  OceanFishUnmirror(Mirror _m) { m = _m; }
  OceanFish val() {
    if (m instanceOf OceanFishMirror)
      return ((OceanFishMirror)m).oceanFishVal();
    else
      raise new Error("not an OceanFish");
  }
}
class DoubleUnmirror {
  Mirror m;
  DoubleUnmirror(Mirror _m) { m = _m; }
  Double val() {
    if (m instanceOf DoubleMirror)
      return ((DoubleMirror)m).doubleVal();
    else
      raise new Error("not a double");
  }
}
```

Supporting overloaded operators like `+` with multiple dynamic arguments is only slightly more complex.

2.4 Checks and Coercions

Our implementation sketch for `dynamic` demonstrates the two main ingredients in our model of interoperability. Mirrors expose information about a class in a way that is more explicit than Java reflection and more under the control of the compiler. Contracts correspond to wrappers that enforce proper use of an object's methods. Composing these two operations provides a compact and understandable model of interoperability.

3. INTEROPERABILITY

The previous section shows how mirrors and contracts can express the dynamic checks and coercions that are necessary to preserve Java's type safety in the presence of dy-

```
;;Implement server functionality
(define (add servlet) ....)
(define (send obj) ....)
(define stylesheets ....)

;;Define contracts for the arguments to server functions
(define servlet
  (object-contract
    (ok (url . -> . boolean))
    (page (url . -> . html))))

(define dialog
  (object-contract
   (page (-> html))
   (parse (form-response . -> . any))))

;;Export server functionality with contracts
(provide/contract [add (servlet . -> . void)])
(provide/contract [send (dialog . -> . any)])
(provide/contract [stylesheets bool])
```

Figure 1: Server API from Scheme

namic operations. We now show how dynamically enforced contracts preserve Java's type safety in Scheme, and how Scheme's contracts are preserved in Java programs. In addition, blame assignment [13] pinpoints the source of the error when Scheme code abuses a Java value or when Java code abuses a Scheme value.

3.1 Server and Servlets

We begin by illustrating the need for contracts with an example. To implement a web-based service, a programmer might benefit by using both Scheme and Java. Scheme provides continuations, which are convenient for implementing web sessions that span requests to the server [22, 33]. Java, meanwhile, supports many existing libraries for manipulating HTML, and specific request handlers can be conveniently packaged as Java-style objects.

Figure 1 presents the interface for a Scheme-implemented server module. The `add` and `send` functions are implemented in terms of objects:

- The `add` function registers a servlet to handle requests to a particular URL. The servlet is represented by an object that matches the contract `servlet`. Such an object has two methods: `ok`, that checks whether this servlet should handle a request at the given URL, and `page`, that generates a page to satisfy the request. The contracts `url` and `html` are defined elsewhere and match standard data structures for urls and html.

- The `send` function uses continuations to implement an interactive dialogue with a web client. It takes an object that matches the `dialog` contract to represent the interaction. This object's `page` method generates a "question" web page to send to the client, and when the client responds, the `parse` method is called to parse the HTTP-delivered response string into a value representing the client's response. (Like `url` and `html`, `form-response` is defined elsewhere. It matches the bindings provided to a CGI script.) The result of `parse` becomes the result of the `send` function.

The variable `stylesheets` indicates if the server is configured to support style sheets.

The contract for `add` specifies that it accepts one argument, matching the `servlet` contract, and it returns nothing. The contract for `send` indicates that it accepts one argument, this time matching the `dialog` contract, and it can return any value. Finally, `stylesheets` must be a boolean. The `servlet` contract matches objects with two methods. The first method, `ok`, accepts a `url` and returns a boolean, and the second method, `page` also accepts a `url`, but returns `html`. Similarly, `dialog` has two methods: `page` that accepts no arguments and returns `html` and `parse`, from `form-response` to `any`, allowing any value as its result.

Java programmers can use the Scheme server to implement their web pages. With the `dynamic` extension, this can be done without additional API calls or marshaling. For example, a simple servlet is implemented in Java plus `dynamic` as follows.

```
class Hello {
  boolean ok(dynamic u) {
    return "hi".equalsIgnoreCase(u.target());
  }
  HTML page(dynamic u) {
    return new HTML("Hello World!");
  }
}
```

This servlet presents a simple page, which has no interactive content, containing the classic greeting. The `ok` method of `Hello` assumes that the given value for `u` is an object containing a target method, which will return the end destination of a URL (i.e. returning `"hi"` for the URL `"www.server.com/hi"`). The `page` method does not provide any links or requests for further information, so the dynamic argument is not used.

This servlet is registered with the server using the `add` function from the Scheme server library:

```
add(new Hello());
```

Our second example, in Figure 2, uses the `send` function to create an interactive web session. The following `Welcome` servlet asks the client for a name, and then responds with a customized "Hello" web page using the response. If the server supports stylesheets, a fancier web page is generated. The servlet programmer knows that the String return type of `send` will be satisfied, because the `parse` method of `Who` returns a `String`.

A revised `Welcome` servlet can use `send` to get a number from a second interaction.

```
class Welcome {
  ....
  HTML page(dynamic u) {
    String name = send(new Who());
    int num = send(new FavNum());
    ....
  }
}

class FavNum {
  HTML page(dynamic u) {
    return new HTML(.... "What's your favorite number?"
                    ....);
  }
  int parse(dynamic a) {
    return a.extractValue("num");
  }
}
```

The expected return value of the second use of `send` is different from the first use, where the value is stored to an

```
class Welcome {
  ....
  HTML page(dynamic u) {
   String name = send(new Who());
   if (stylesheets)
      return new HTML(....)
   else
      return new HTML(...., "Hi " + name));
  }
}
class Who {
  HTML page(dynamic u) {
    return new HTML(.... "What's your name?" ....
                        new Form("name", ...) ....
                        new Button("Submit", ... u ...));
  }

  String parse(dynamic a) {
    return a.extractValue("name");
  }
}
```

Figure 2: Interactive Servlet

`int` instead of a `String`. Again, the programmer knows that the expectation will be satisfied, because `FavNum`'s `parse` produces an `int`.

A sophisticated whole-program analysis might be able to discover, as the programmer knows, that the `String` and `int` results from the two different `send` calls are correlated with the `String` and `int` results from the `Who` and `FavNum` classes. Dynamically checking the results, however, allows this program to run safely even without such analyses.

3.2 Origin of Contracts

The context of each use of a `dynamic` variable dictates its contract. For example, the implementation of `ok` in `Hello` expects `u` to be an object that has a `String`-producing `target` method. The compiler determines this expectation by examining the context surrounding the use of `u`. The attempt to dispatch a method indicates that this use of `u` must be an object. The method name used and the number of arguments supplied provide the information regarding what method the object must have. The implementation of `equalsIgnoreCase` specifies that the argument must be a `String`, which provides the final piece of information regarding the expected types of `u`.

In all circumstances, the use of a dynamic variable provides the information necessary to derive a contract for it. For example, the conditional position in an `if` must be a boolean, the right-hand side of an assignment must match the type of the variable in the left-hand side, the argument to a cast must match the cast type, a constructor's actual arguments must match the constructor's formal arguments, and primitive operations must all have sensible inputs. If a dynamic variable occurs in the argument to a method invocation, the class or interface of the receiver dictates the type. If, however, a dynamic variable occurs as the receiver of a method, the contract contains a single method whose arguments match the argument types of the method invocation. If both the receiver and the arguments to a method invocation have type dynamic, no contract is placed on the arguments, and the receiver's contract has a single method whose arguments have the `any` contract.

Contracts additionally arise from declarations found within Scheme libraries, such as the contracts added in the servlet library. These contracts are attached to references to the Scheme variables.

3.3 Contract and Evaluation Notation

To discuss the coercions and checks performed by contracts without committing to an implementation strategy, we show contracts in superscripts above contracted expressions. For the `ok` example, from Section 3.1, we write an abstract *targetMethod* contract for the use of `u` and apply it as follows.

$$targetMethod = \texttt{object}\{\texttt{target} : (\rightarrow string)\}$$

```
boolean ok(dynamic u) {
  return "hi".equalsIgnoreCase(u^targetMethod.target());
}
```

In this notation, *targetMethod* provides a name for the contract, `object` constructs a contract for an object with the given methods. The methods are specified by name `target`, with the contract following the colon for the method arguments (if the method has arguments) appear to the left of the $\rightarrow$, and the result appears to the right.

We show evaluation via algebraic simplification of expressions, much like earlier models of Java [18, 27]).

3.4 Checking Contracts

Contract enforcement occurs during program execution. When a contracted value is encountered, the actual values are checked against the expectations represented by the contract. To explain how the presence of contracts affect evaluation, we first examine the evaluation of contracts in a simple setting. In particular, we first consider dynamic variables that are bound to values from Scheme, where the contract inferred from the type context in Java matches the contract written by the Scheme programmer.

First, imagine that the dynamic variable `stylesheet` is used in a context expecting a boolean value. Assuming that the actual value of `stylesheet` is `false`, the evaluation first looks up value, then checks the contract against the value. Since the contract matches the value, the contract is discarded

```
if (stylesheet^bool) ....
⇒ if (false^bool) ....
⇒ if (false) ....
```

If, instead, the value associated with `stylesheet` is `3`, the contract checker detects and reports an error instead of discarding the contract.

```
if (stylesheet^bool) ....
⇒ if (3^bool) ....
⇒ error
```

In this case, the blame for the contract violation lies with Scheme, since Scheme promised that `stylesheet` would be a boolean.

Wheras simple contracts like *bool* can be checked immediately, not all contracts can be checked right away. To demonstrate, we return to our servlet example. Consider the call to the `add` function:

```
add(new Hello());
```

where the function `add` is a `dynamic` variable. The contract for this use is

$hello$ = `object{ ok` : ($any \to bool$)
`page` : ($any \to html$) }

`add`$^{(hello \to void)}$`(new Hello());`

In this case, we cannot determine if `add`'s arguments match the contract by considering only `add` itself. We can, however, check that `add` is a function and that it has the right arity. Afterward, we distribute the argument and result contracts to the arguments and to the result of the application, as follows.

`add`$^{(hello \to void)}$`(new Hello())`
$\Rightarrow$ `add`$^{(hello \to void)}$`(objref)`
$\Rightarrow$ `(add(objref`^{hello}`))`^{void}
$\Rightarrow$ ....

Assigning blame for such contracts is more complex than for simple contracts like *bool*. Clearly, if `add` is not a function, Scheme must be to blame. If the argument to `add` is not an html object, however, Java must be to blame, since the html objects flow from Java to Scheme. Similar reasoning shows that if the results of `add` do not match the contract, Scheme must be to blame, since Scheme is supplying the objects to Java. In general, the language that is in control of the evaluation at the point where values pass between languages must be responsible for those values. Since function arguments flow in the opposite direction from function results, the guilty party for a function argument is the opposite of the guilty party for a function result.

To show blame more clearly during evaluation, we annotate contracts that blame Java with `JS:` and contracts that blame Scheme with `SJ:`. We put both letters into each exponent to capture all of the potentially guilty parties for each contract, even though the evaluation will arrange to have the guilty party first in each exponent at the point the violation is discovered. We also show the annotations being pushed into the arrow contract as a separate step that corresponds to the check that a value is a procedure of the right arity.

The following shows the earlier example with the additional annotations. In the transition from the second step to the third, the annotations are pushed into the arguments and results of `add`'s contract. This corresponds to the discovery that `add` is indeed a function.

`add`$^{SJ:(hello \to void)}$`(new Hello())`
$\Rightarrow$ `add`$^{SJ:(hello \to void)}$`(objref)`
$\Rightarrow$ `add`$^{(JS:hello \to SJ:void)}$`(objref)`
$\Rightarrow$ `(add(objref`$^{JS:hello}$`))`$^{SJ:void}$
$\Rightarrow$ ....

Like procedure contracts, object contracts must also be distributed to method arguments and results during method invocation. For example, when `objref` flows into Scheme, it maintains the *hello* contract until Scheme invokes one of its methods. At that point, the *any* and *html* contracts distribute to the method invocations arguments and result.

`(send objref`$^{JS:hello}$ `page url)`
$\Rightarrow$ `(send objref page url`$^{SJ:any}$`)`$^{JS:html}$
$\Rightarrow$ ...

Consequently, `url` values that flow from Scheme into Java must respect Java's type discipline, as enforced by the contract checker.

The contract inferred from Java's type context may not always match the contract written by the Scheme programmer. Still, we can use the same basic technique for checking such mismatched contracts. Instead of using a single contract to enforce both Java's and Scheme's obligations however, we use two contracts: one synthesized from Java's type context, and one extracted from Scheme's `provide/contract` form. Since the two contracts have different sources, there are now three potential sources of an error: Java does not meet Scheme's contract, Scheme does not meet Java's contract, or the two contracts do not match. To capture the third kind of blame, we imagine a third party, named C, that mediates the composition and is blamed for any mismatch between the Java-side contracts and the Scheme-side contracts.

As an example, imagine that the Java program contains

```
1 + add(new Hello())
```

The context of `add` dictates that its contract must be $hello \to int$ but Scheme has promised $hello \to void$.

To enforce these two contracts on the same value, we put them both into the exponent on `add`, using C as the opposite party for both the Scheme contract and the Java contract:

`1+add`$^{SC:(hello \to void),CJ:(hello \to int)}$`(new Hello())`

Since the value initially flows from Scheme to Java, we put SC on the Scheme contract, indicating that Scheme is initially responsible for the value itself, and we put CJ on the Java contract, indicating that Java is responsible for arguments flowing into the value. Following the same simplification steps as before, but with the new contracts, we get

$\Rightarrow$ `1+add`$^{SC:(hello \to void),CJ:(hello \to int)}$`(objref)`
$\Rightarrow$ `1+add`$^{(CS:hello \to SC:void),(JC:hello \to CJ:void)}$`(objref)`
$\Rightarrow$ `1+(add(objref`$^{JC:hello,CS:hello}$`))`$^{SC:void,CJ:int}$

At this point, the two contracts on `objref` are identical, so we know that the composer can never be blamed. The contract on the results, however, are different. Assuming that `add` matches its Scheme contract, we eventually get something like the following (where `void` is a special value in Scheme that has no operations, unlike `void` in Java):

$\Rightarrow$ `1+void`$^{SC:void,CJ:int}$

Now, since the value is indeed void, the Scheme contract can be discarded and we are left with

$\Rightarrow$ `1+void`$^{CJ:int}$

which aborts the program, since void is not an integer. In this case, C is blamed, indicating that a mis-match between the two contracts was detected.

In a real implementation, the annotations S, J, and C point to the program points where the code first began interoperating, so that the contract failure pinpoints a specific expression, and not merely a specific side of the interoperating code. We believe that precise blame is crucial to providing the kind of feedback that a Java programmer expects from a type failure, even when that failure is dynamic rather than static.

4. JAVA AND SCHEME SPECIFICALLY

So far, we have described a model of interoperability that is mostly independent of Java and Scheme. Our introduction of coercions depends on Java-side `dynamic` annotations, but the same strategy works with other languages as long as interoperating operations are syntactically distinguished from language-local operations. Our strategy for introducing contracts for type safety could, in principle, be adapted to other typed languages. Finally, constructors such as `->`

and `object-contract` are specific to Java and Scheme, but the underlying model of contracts and coercions works the same with different constructors for different languages.

Meanwhile, to complete the interoperability picture for Java and Scheme specifically, we must consider many additional details, including the rules of coercion for various forms of data, how modularity constructs in each language map to the other to enable cross-language references, and how the control-flow constructs of each language relate.

4.1 Data Mapping

Java provides two kinds of data: primitive values and instances of classes. Primitive values include `int`s, `double`s, and `char`s. Arrays are instances of array classes, but they are provided special syntax and language support. To a lesser extent, instances of the `String` class are also treated differently than other classes.

Scheme provides a richer set of primitive data, including real numbers, complex numbers, characters, strings, symbols, pairs, and vectors. Of course, Scheme also provides functions as values.

To bridge the gap between objects and functions, we add functions to Java and objects to Scheme. Among primitive data, a Java integer can be used directly as a Scheme exact number, a Java `double` can be used directly as a Scheme inexact number, and Scheme numbers can be used directly in Java as long as they can be represented as an `int` or `double` value. Characters are nearly equivalent in both languages. Finally, strings and arrays are treated specifically, as we describe below. For all other forms of data, operations must be performed through facilities provided in the language specifying the data.

Functions and Objects

A function call in our extended Java has the same syntax as a static method call, except that the function position is an expression of type `dynamic`. A static method can be used in a dynamic expression position, in which case it is coerced to a Scheme function. Java methods cannot be used as functions. We have so far not added a non-static procedure form to Java (to enable closures to be created in Java), but such a form would be straightforward to define.

An object in our extended Scheme is an instance of a class. Scheme classes can be implemented as subtypes of Java classes, and vice-versa. A `class` form in Scheme creates a class, a `new` form instantiates a class, a `send` form performs a method call on a given object, and field accessor and mutator functions support field access.

Strings

A Java `String` is different from a Scheme string. In Java, a `String` is also an `Object`, whereas a PLT Scheme string is not naturally a PLT Scheme object. Instead of picking a uniform representation, we implement strings differently for each language, and `dynamic` coercions shift between representations.

Arrays

A Scheme vector is like an array, but a Java array cannot be a Scheme vector, because a Java array can be cast to and from `Object`. Also, assignments to an array index must be checked to ensure that only objects of a suitable type are placed into the array. For example, an array created to contain `Fish` objects might be cast to `Object[]`. Assignments into the array must be checked to ensure that only `Fish` objects appear in the array.

To allow casts and implement Java's restrictions, a Java array is an instance of a class that descends from `Object`, and it has no relation to a Scheme vector. This mismatch between Java and Scheme is partly due to a lack of support for mutable vectors in our current contract implementation. The mismatch limits interoperability, and we hope to resolve this mismatch with future improvements to our contract system.

4.2 Program Organization

Java provides packages and classes for program organization. Packages organize classes in the large, and classes organize static methods and nested classes (in addition generating objects). A unit of compilation in Java is orthogonal to either of these in Java; a compilation is defined by a set of `.java` files that include mutual type references.

PLT Scheme provides a `module` form for organizing programs in the large [17]. A `module` contains definitions and expressions, much like the conventional Scheme top level, but with explicit exports of definitions that other modules can use, and with explicit imports for all bindings that are not defined within the module. A module is also a compilation unit in PLT Scheme. Modules cannot be nested, but they can be placed into a *collection* hierarchy that is analogous to a package hierarchy.

From the Scheme perspective, Java packages and the static parts of classes can be treated like modules, with the class itself as an implicit export of the "module". From the Java perspective, Scheme modules can be treated like classes with only static components.

4.3 Control Flow

Java and Scheme provide essentially the same call-by-value, eager evaluation model. Scheme supports first-class continuations and tail evaluation, both of which are consistent with Java [11, 35], so our extended Java supports them.

PLT Scheme provides an exception system that behaves much like Java's. A value can be raised as an exception using `raise`, which is like Java's `throw`, and an exception can be caught using `with-handlers`, which is like Java's `try`. The `with-handlers` form includes a predicate for the exception and a handler, which is analogous to Java's implicit instance test with `catch` and the body of the `catch` form. We implement Java's `finally` clause using `dynamic-wind`.

Unlike Java's `throw`, the PLT's `raise` accepts any value, not just instances of a throwable. Nevertheless, PLT tools work best when the raised value is an instance of the `exn` record. This record contains fields specifying the message, source location of the error, and tracing information. We therefore connect the `Throwable` Java class to PLT Scheme's exception records.

When the `Throwable` is given to `throw`, a contract is implied that coerces the Throwable into the Scheme form. A `catch` form implies a contract that expects a Scheme exception record and coerces the value into a `Throwable`. In circumstances where the exception value does not reflect a `Throwable` instance, the coercion creates a new `Throwable` instance of the appropriate type with the information provided by the exception record.

Besides generally fostering interoperability, this re-use of PLT Scheme's exception system ensures that Java programs running within DrScheme get source highlighting and stack traces for errors. All of Java's other built-in exception classes derive from Throwable, and therefore inherit this behavior.

4.4 Using Scheme from Java

Our extended Java compiler treats the package name `scheme` as an entry point into the Scheme module namespace. Consequently, the import statement

```
import scheme.web.server;
```

directs the compiler to search for a Scheme module `server` in the "web" collection.

The Java program can refer to a binding that is exported by the `server` module in the same way as accessing a package member or a static class member:

```
server.serve();
```

Statically, the compiler determines whether `serve` is exported from the `server` module. If not, a compile-time error is reported. Otherwise, the reference is treated as an expression of type `dynamic`.

Naming conventions in Scheme are different than naming conventions in Java, and Scheme names can contain characters that are not permitted within Java names. Several such characters are used frequently by convention, including "-", "?", and ">". To ease cross-language references, the Java compiler converts Scheme names that follow certain conventions to names using Java conventions. Dashes in Scheme names are dropped, and the following letter is capitalized; arrows `->` in scheme names are replaced by `To`, and the following letter is capitalized; and trailing question marks are replaced by `P`. Thus, a Scheme name `add-server` becomes `addServer`, and `url-ok?` becomes `urlOkP`. In the rare case where a convention does not make a Scheme name Java-friendly, the programmer can implement a small Scheme module to re-export a name that is more friendly.

4.5 Using Java from Scheme

A Scheme module imports bindings with an `import`-like `require` form, and this form can be used with a suitable path to import a Java class into a Scheme module. The module gains access to the class, its (non-private) static members, a set of field accessors, and the nested classes of the Java code.

Although every Java identifier is a legal Scheme identifier, Java's overloading and scoping rules mean that the full name of a Java method or field includes type information. Scheme programmers can access a specific method or field in a class by using a mangled name. For example, the mangled name of the `canEat` method of `Shark` is `canEat-int`. This mangling causes two problems for interoperability. The first problem is that asking the programmer to manually mangle names make calling Java from Scheme difficult. The second problem is that it inhibits the use of Scheme-oriented code on Java objects.

A related problem is that instantiating PLT Scheme classes differs from instantiating Java classes. Java's `new` form is similar to a method call, in that the class can support multiple constructors, and the arguments in the `new` form disambiguate the overloaded call. Scheme classes have a single initialization entry point (though classes can support optional keyword arguments for this initialization). Instantiating a Java class from Scheme is therefore a two-step process of creating the object and then calling an initialization method that corresponds to a specific constructor.

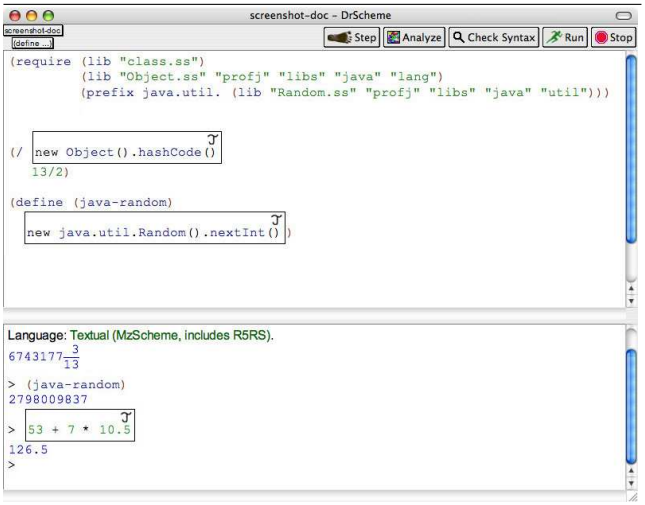

Figure 3: Java Box

We are exploring two solutions to these problems. The first solution is to introduce Java-specific notation into Scheme programs for Java-specific calls. Figure 3 shows boxed Java expressions within a Scheme program in the DrScheme programming environment. This approach makes class instantiation and method calls simpler, at least when no overloading must be resolved.

Our second solution is to compile Java classes with extra mappings for non-overloaded method names. These extra mappings convert a Java-style name to a Scheme-style name, such as converting `canEat` to `can-eat`, without mangling the name with types. In the common case where overloading is not used, this makes invocation of Java methods look syntactically like invocations of Scheme methods, which is especially valuable for reusing Scheme-specific code on Java objects.

Another possibility is to adjust dynamic method calls in Scheme. Method dispatch in Scheme could use contract information to search for a method in the object whose mangling is consistent with the actual arguments.

A Scheme module can attach contracts to its exports, as in Figure 1, to protect itself from abuse and to help pinpoint misuse of the module from Java code. When writing code that is specifically intended for use from Java, a Scheme programmer might more conveniently create a Java wrapper for the Scheme module, where the Java types effectively provide contract declarations for the Scheme code.

5. IMPLEMENTATION

Our implementation compiles Java to PLT Scheme. We compile Java to Scheme (in contrast to the more popular approach of compiling Scheme to Java) so that we can run Java programs to execute within our DrScheme environment [14] and so we can control the compilation and error messages of Java programs for pedagogic purposes [23].

Our compiler begins by parsing Java code using a LEX-

/YACC-style parser generator. It then processes the Java source for type-checking and contract and mirror insertions, and then produces PLT Scheme modules for evaluation. Compilation preserves the original source location information, so that run-time errors, including contract violations, can report their exact location.

5.1 Compilation Model

As mentioned in Section 4.2, Java classes are treated as PLT Scheme modules, so our compiler produces one module per Java class. However, reference cycles can occur among classes in Java, as long as they do not lead to inheritance cycles. For example in the classes `Swordfish` and `Bait`

```
class Swordfish extends Fish {
  Bait favBait = new Bait ...
}
class Bait {
  Fish f;
  ... (Swordfish) f ...
}
```

`Swordfish` creates instance of `Bait`, and `Bait` casts to `Swordfish`. Compiling both the instantiation and the cast requires information about the structure of the other class. These two classes contain cyclic references.

PLT Scheme modules, the compilation unit for Scheme programs, do not allow cyclic references. Therefore, to accommodate reference cycles between classes, our compiler produces a single Scheme `module` for each collection of strongly-connected Java classes. In the case of `Swordfish` and `Bait`, the classes are compiled into one module, which reflects that these classes form a single compilation unit. Meanwhile, each class used by the dependent group is imported into the `module`.

To retain the Java notion of importing individual classes for program organization, our compiler actually produces $N+1$ modules for N mutually dependent Java sources: one that combines the Java code into a compilation unit, and then one for each class to re-export the parts of the compilation unit that are specific to that class.[1] So, in addition to one module containing `Swordfish` and `Bait`, the compilation of these classes also creates a `Swordfish` module, which re-exports the `Swordfish` class and any static members, and a `Bait` module, which re-exports the `Bait` class and any static members. Thus, Scheme and Java programmers alike can import each class individually.

5.2 Statements and Expressions

Java and PLT Scheme both enforce the same evaluation order on their programs. Therefore, those Java constructs that are subsumed by Scheme constructs have a straightforward translation. For example,

```
bool a = varA < varB, b = varA > 0;
if (a && b)
 res = varA;
else
 res = varB;
```

translates into

```
(let ((a (< varA varB))
      (b (> varA 0)))
  (if (and a b)
      (set! res varA)
      (set! res varB)))
```

with annotations specifying the source location. Indeed, the majority of Java's statements and expressions translate as expected.

[1] If a class is not a member of any dependency cycle, then the compiler produces only one module.

5.3 Classes

A Java class can contain fields, methods, nested classes (and interfaces), and code blocks, each of which can be static. A PLT Scheme class is similar, except that it does not support static members. Nevertheless, a static member closely corresponds to a Scheme function, value, or expression within a restricted namespace, i.e., a `module`, so static Java members are compiled to these Scheme forms.

An instance of a class is created with the `new` form. Evaluation of this form triggers the evaluation of the expressions in the top level of the class body. These expressions serve the same purpose as a single Java constructor. However, a Java class can contain multiple constructors, preventing a direct translation from a Java constructor to a sequence of top-level expressions. Instead, we translate Java constructors as normal methods in the Scheme class, and we translate a Java `new` expression into a Scheme `new` followed by a call to a constructor method.

In supporting `dynamic`, mirrors and unmirrors are necessary for each class. The mirror provides both checks and conversions for objects when they enter dynamic contexts. These checks ensure that methods are provided with the correct number and kind of arguments. After the checks, the argument values are unmirrored as the method is called. The result of a method is also mirrored before it is returned. For a class

```
class Fish {
  String getColor(String format) { ... }
}
```

the mirror generated by the compiler is

```
(define fish-mirror
  (class object% ()
    (define (fish ...))
    ; ". args" puts all arguments into a list
    (define (getColor . args)
      (if (= (length args) 1)
          (if (is-string? (first args))
              (apply-string-mirror
               (send fish getColor
                     (apply-string-unmirror
                      (first args))))
              (raise TYPE-MISMATCH-ERROR))
          (raise ARG-NUMBER-WRONG-ERROR)))
    (super-new)))
```

The `apply-string-unmirror` wraps the value with another class, in which each method return is checked before return, and all arguments are converted before the function is called. The `is-A?` check, performs a check that ensures the provided value is the correct kind of data, and where an object is required, contains the correct fields and methods.

5.4 Fields and Methods

Non-`static` Java fields translate into Scheme `field` declarations. A `static` Java field, meanwhile, translates into a Scheme top-level definition. Thus, the fields

```
class Square extends Shape {
  static int avgLength = 1;
  int sideLength = 1;
}
```

become, roughly

```
(define avgLength 1)
```

and

```
(define Square
  (class Shape
    (field (sideLength 1)) ···))
```

However, the above translation does not connect the variable `avgLength` to the containing class `Square`. If multiple classes within a compilation unit contain a static field `avgLength`, the definitions would conflict. For non-static fields, Scheme classes do not allow subclasses to shadow field names again potentially allowing conflicts. Additionally, to avoid conflicts between Java's distinct namespaces for fields, methods, and classes, we append a `~f` to the name. Therefore, we combine `avgLength` with the class name and `~f`, forming the result as `Square-avgLength~f`, and `sideLength` becomes `Square-sideLength~f`. Note that Scheme programmers using this name effectively indicate the field's class.

Compilation generates a mutator function for both of these fields, plus an accessor function for the instance (non-`static`) field. Since the `module` form prohibits mutating an imported identifier, the mutator function `Square-avgLength-set!` provides the only means of modifying the static field's value. If the static field is `final`, this mutator is not exported. Also, instance field mutators are not generated when they are `final`. Thus, even without Scheme-side compile-time checking, Scheme programmers cannot violate Java's `final` semantics.

Similarly, instance methods translate into Scheme methods and static methods into function definitions with the class name prepended, but the name must be further mangled to support overloading. For example, if the `Who` class in Section 3 contained an additional `page` that took two arguments, name mangling would be necessary to distinguish them within Scheme. The translated methods would become `page-dynamic` and `page-dynamic-dynamic`.

Constructors are compiled as methods, which we identify with special names. The constructor for `Welcome` in Section 3 translates into `Welcome-constructor`. Technically the `-constructor` suffix is not necessary to avoid conflicts, but it clarifies that the method corresponds to a constructor.

A `private` Java member *does not* translate to a `private` Scheme member, because `static` Java members are not part of the Scheme class, but Java allows them to access all of the class's members. We protect `private` members from outside access by making the member name local to a module with PLT Scheme's `define-local-member-name` macro; the Java-to-Scheme compiler ensures that all accesses within a compilation unit are legal. Our compiler does not currently preserve protection for `protected` and package members.

5.5 Nested Classes

In Java, a nested class may either be `static` or an instance class, also known as an inner class. An inner class can appear as a member of a class, within statement blocks, or after `new` (i.e. an anonymous inner class).

Static nested classes are equivalent to top-level classes that have the same scope as their containing class, with the restriction that they may not contain inner classes. These can be accessed without directly accessing the containing class. When compiled to Java bytecodes, nested classes are lifted out and result in separate `.class` files. We also lift a nested class, and provide a separate module for external access. We treat a nested class and its container as members of a cycle, placing both in the same module.

Inner classes are also compiled to separate classes. Unlike static nested classes, they may not be accessed except through an instance of their containing class or within the containing method. A separate module is therefore not provided, and construction may only occur through a method within the containing class.

For member inner classes, the name of a nested class is the concatenation of the containing class's name with the class's own name. Class `B` in

```
class A {
  class B {
  }
}
```

is accessed as `A.B`. For named classes within a block statement, we amend the name with uniquely specifying information, so that it cannot conflict with other class names, and lift the class as we would a member inner. Similarly, anonymous inner classes are given a unique name and lifted, as is done by bytecode compilers.

5.6 Dynamically checked values

After type checking determines the expected types of dynamically typed variables, the compiled uses are augmented with contracts and mirrors. Consider the following program fragment:

```
URL knownURL = ...;
boolean sameAs( dynamic givenURL ) {
  return knownURL.equals(givenURL);
}
```

From the context, this usage of dynamic variable `givenURL` assumes that `givenURL` is an `Object`. Within the call to `equals`, `givenURL` is replaced with

```
(if (satisfies-interface? Object givenURL)
    (apply-unmirror-Object givenURL)
    (ERROR))
```

When executed, the `satisfies-interface?` function ensures that `givenURL` is a Scheme object containing the methods (with the correct number of arguments) of a Java `Object` instance. The `apply-unmirror-Object` function wraps thee usage of `givenURL` with the unmirror for Object discussed in Section 5.3. This unmirror ensures that values returned from method calls on `givenURL` are properly checked and unmirrored. Additionally, `apply-unmirror-Object` guarantees that any Java values passed as method arguments are properly mirrored.

For interoperability, Java Strings entering Scheme need to be converted into Scheme strings. Therefore, a string unmirror converts a Scheme string into an instance of the Java class, by embedding the value inside of the Java object. Similarly, a String mirror extracts the Scheme string from the `String` object.

When dynamically typed variables are used as objects, with unknown classes, the wrapping uses a contract to ensure that the value is an object with the named method or field, and then mirrors the arguments in place, while unmirroring the expected value. In the example

```
URL serverBase = ...;
...
... givenURL.rootedAt(serverBase) ...
```

The `givenURL` use is wrapped with an object-contract, ensuring that givenURL is an object (not necessarily a Java `Object`) with a rootedAt method taking one argument. The `serverBase` value is mirrored and the result of the call is checked and unmirrored in place.

The translation of dynamically typed variables used as functions is similar to their use as objects.

```
send(who) + " welcome"
```

the `dynamic send` variable use is wrapped with a contract: `(any . -> . string)`. This contract ensures that `send` is a function that returns a string. The `who` value is wrapped with a mirror. Further, a string unmirror is wrapped around the return.

5.7 Native Methods

Methods annotated in Java with `native` are typically implemented in C according to JNI [28]. Although PLT Scheme also provides a native interface to C code, considerable work remains to bridge the PLT and JNI interfaces.

For now, our system assumes Scheme as the implementation language for `native` methods. When the compiler encounters a class using `native` methods, such as

```
class Time {
 static native long getSeconds(long since);
 native long getLifetime();
}
```

the resulting `module` for `Time requires` a Scheme module `Time-native-methods` which must provide a function for each native method. The name of the native method must be the Scheme version of the name, with `-native` appended at the end. Thus a native function for `getSeconds` should be named `Time-getSeconds-long-native` and `getLifetime` should be `getLifetime-native`.

Within the compiled code, a stub method is generated for each native method in the class, which calls the Scheme native function. When `getSeconds` is called, its argument is passed to `Time-getSeconds-long-native` by the stub, along with the class value, relevant accessors and mutators, and generics for private methods. An instance method, such as `getLifetime`, additionally receives `this` as its first argument.

6. RELATED WORK

Related work fits into three broad categories: dynamic expressions for statically typed languages, language interoperability, and other interoperable implementations of Java with functional languages.

6.1 Mixing Dynamic and Static Types

Strongtalk [8] adds an optional static type system to Smalltalk. On the boundary between typed and untyped expressions the compiler either assumes a type or relies on an annotation from the programmer. Unlike our system, these types are not validated at runtime, so that if a type annotation or assumption is incorrect, typed operations can be misapplied.

Starting from the other side, others have proposed a `Dynamic` type constructor for ML [1, 12], and our `dynamic` declaration is similar to this constructor. The `Dynamic` constructor encapsulates data with unknown types, with the intent that untyped data arises from I/O calls, as opposed to other programming languages. In order to extract data from a `Dynamic` value, programmers first explicitly test the type of the value. Only data with named types can be extracted. Like our system, these types are checked at run time, but a programmer must explicitly cast the data and provide datatype definitions for all values.

6.2 Language Interoperability

Component models, such as COM [34], CORBA [31], SOM [25], and DOM [37], support language interoperability through a shared interface definition language (IDL). Each IDL is effectively a universal interface for data interchange, so that more than two languages can communicate with an application, but this "least common denominator" approach limits the granularity of interoperability. Furthermore, the component model is generally specified outside the programming language, so that it is relatively opaque to compilers and analysis tools.

The Microsoft .NET framework [32, 26, 10] is similar to component models, but interoperating languages are all compiled to a Common Intermediate Language (CIL). This approach make interoperability more apparent to the compiler, but also limits implementation strategies for languages to those that fit well into CIL.

SWIG [4] bridges C-level libraries and higher-level (often dynamic) languages by automatically generating glue code from C and C++ sources. Programmers can implement new modules for SWIG to support bridges from C to new programming languages, but only to the degree supported by a foreign-function interface in the non-C language. In short, SWIG supports interoperability much like component models or .NET, but with C as the universal API.

The nlffi-gen [6] foreign-function interface for ML produces ML bindings for C header files. The system also provides an encoding of C datatypes into ML so that ML programmers may directly access C datatypes. Marshaling of data occurs within the ML program, written by the programmer, where necessary. The system does not provide support for ML data representation into the C code. Interoperating between ML and C is easier for the programmer, although type-safety and data representation are still a concern for the programmer.

Furr and Foster have developed a system that analyzes C foreign-function libraries for OCaml [19]. Their tool uses OCaml type information to statically locate misuses of the OCaml data within C. In a sense, this tool effectively bridges the gap between a strongly typed language and a weakly typed language by strengthening the latter's type system.

6.3 Java and Functional Languages

Several Scheme implementations implement interoperability with Java while compiling Scheme to Java.

JScheme [2, 3] compiles to Java a dialect of Scheme resembling the R^4RS standard. Within Scheme, the programmer can use static methods and fields, create instances of classes and access their methods and fields, and implement existing interfaces. Scheme names containing certain characters are interpreted automatically as manglings of Java names. Java's reflection functionality is employed to select (based on the runtime type of the arguments) which method to call. This technique is slower than selecting the method statically, but requires less mangling. From Java, users can call the JScheme compiler or interpreter and receive Java values from Scheme programs. These values are the Java representations of Scheme values (i.e instances of closure objects, pair classes and other values which require the user to not just know Scheme but how the compiler compiles Scheme), and must be cast from Object at runtime.

The code fragment below illustrates a use of Scheme's complex numbers through the JScheme API:

```
import jscheme.JS;
...
int x, y;
...
JS scheme = new JS();
scheme.load(new java.io.FileReader("math.init"));
```

```
Object z = scheme.call("make-rectangular",
                       scheme.toObject(x),
                       scheme.toObject(y));
double m =
   scheme.doubleValue(scheme.call("magnitude", z));
...
```

JScheme's interoperability through an API is typical of Java–Scheme implementations, and other implementations that target Java.

SISC [29] interprets the R^5RS standard dialect of Scheme, using a Java class to represent each kind of Scheme value. Closures are represented as Java instances containing an explicit environment. Various SISC methods provide interaction with Java [30]. As with JScheme the user may instantiate Java objects, access methods and fields, and implement an interface. When passing Scheme values into Java programs, they must be converted from Scheme objects into the values expected by Java, and vice-versa. To access Scheme from Java, the interpreter is invoked with appropriate pointers to the Scheme code.

Kawa [7] compiles R^5RS code to Java bytecode. Functions are represented as classes, and Scheme values are represented by Java implementations. Java static methods may be accessed through a special primitive function class. Values must be converted from Kawa specific representations into values expected by Java. In general, reflection is used to select the method called, but in some cases, the compiler can determine which overloaded method should be called and specifies it statically. Accessing Scheme values from Java is similar to access in JScheme.

MLj [5] compiles ML to Java, and it supports considerable interoperability between ML and Java code. MLj relies on static checking and shared representations ensure that interoperability is safe; not all ML values can flow into Java. Of course, MLj adds no new degree of dynamic typing to Java.

7. CONCLUSION

Fine-grained language interoperability is an elusive goal. Practically all attempts fall into one of two categories: relatively coarse-grained interoperability for all languages through a universal data API, or relatively fine-grained and carefully crafted interoperability for a pair languages.

Our attempt falls squarely in the second category, but we believe that it is a step towards a general model of fine-grained interoperability. We believe that such a model can be used to formally support soundness claims, to guide compiler writers in implementing interoperability, and to help programmers understand the definition of a particular instance of interoperability. Certainly, when implementing our current system, understanding coercions through a composition with contracts helped us to insert coercions in the right places.

Many steps remain toward a complete model. In particular, we suggest a formal account of interoperability in terms of mirrors and contracts, but a complete formal account remains ongoing work. Also, the Java- and Scheme- specific details of our implementation are as ad hoc as any previous attempt; we have more work to do in finding general principles. Nevertheless, by showing a hint of a formal model and by completing an implementation for Java and Scheme, we hope to have demonstrated the role of contracts and mirrors in taming interoperability.

Our implementation is available with the current release of the DrScheme software system: www.plt-scheme.org/download/ The language level must be set to Experimental Languages: ProfessorJ : Java + dynamic

8. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Computing Systems*, 1991.

[2] K. Anderson, T. Hickey, and P. Norvig. *JScheme User manual*, Apr. 2002. `jscheme.sourceforge.net/jscheme/doc/userman.html`.

[3] K. R. Anderson, T. J. Hickey, and P. Norvig. SILK - a playful blend of Scheme and Java. In *Proc. Workshop on Scheme and Functional Programming*, Sept. 2000.

[4] D. M. Beazley. SWIG : An easy to use tool for integrating scripting languages with C and C++. In *Tcl/Tk Workshop*, 1996.

[5] N. Benton and A. Kennedy. Interlanguage working without tears: Blending SML with Java. In *Proc. ACM International Conference on Functional Programming*, pages 126–137, 1999.

[6] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL: Workshop on multi-language infrastructure and interoperability*, Sept. 2001.

[7] P. Bothner. Kawa: Compiling Scheme to Java. In *Lisp Users Conference*, Nov. 1998.

[8] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1993.

[9] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programing languages. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[10] M. C. Carlisle, R. E. Sward, and J. W. Humphries. Weaving ada 95 into the .net environment. In *Proceedings of the ACM SIGAda international conference on Ada*, 2002.

[11] J. Clements and M. Felleisen. A tail-recursive semantics for stack inspections. In *Proc. European Symposium on Programming*, 2003.

[12] D. Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Computing Systems*, 1999.

[13] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. ACM International Conference on Functional Programming*, Oct. 2002.

[14] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, Sept. 1997.

[15] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *Proc. European Conference on Object-Oriented Programming*, 2004.

[16] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[17] M. Flatt. Composable and compilable macros: You want it when? In *Proc. ACM International Conference on Functional Programming*, pages 72–83, Oct. 2002.

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.

[19] M. Furr and J. S. Foster. Checking Type Safety of Foreign Function Calls. Technical Report CS-TR-4627, University of Maryland, Computer Science Department, Nov. 2004.

[20] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[21] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition edition, 2000.

[22] P. Graunke. *Web Interactions*. PhD thesis, Northeastern University, 2003.

[23] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *OOPSLA Companion: Educators' Symposium*, pages 170–177, Oct. 2003.

[24] K. E. Gray and M. Flatt. Compiling Java to PLT Scheme. In *Proc. Workshop on Scheme and Functional Programming*, Sept. 2004.

[25] J. Hamilton. Interlanguage object sharing with som. In *COOTS*, 1996.

[26] J. Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, 2003.

[27] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition edition, 1999.

[29] S. G. Miller. SISC: A complete Scheme interpreter in Java. `sisc.sourceforge.net/sisc.pdf`, Feb. 2003.

[30] S. G. Miller and M. Radestock. *SISC for Seasoned Schemers*, 2003. `sisc.sourceforge.net/manual`.

[31] OMG(1997). The common object request broker: Architecture and specification. Revision 2.0 July 1995, Update July 1996, Object Management Group, formal document 97-02-25 www.omg.org.

[32] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, second edition edition, 2002.

[33] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proc. ACM International Conference on Functional Programming*, 2000.

[34] D. Rogerson. *Inside COM*. Microsoft Press.

[35] W. Tao. *A Portable Machanism for Thread Persistence and Migration*. PhD thesis, University of Utah, 2000.

[36] G. van Rossum. *Python Reference Manual*, Nov. 2004. `http://docs.python.org/ref/ref.html`.

[37] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *COOTS*, 1996.