LazyJ: Seamless Lazy Evaluation in Java

Alessandro Warth

Computer Science Department
University of California, Los Angeles
awarth@cs.ucla.edu

Abstract

LazyJ is a backward-compatible extension of the Java programming language that allows programmers to seamlessly tap into the power and expressiveness of lazy evaluation. It does so by extending Java's type system with lazy types. A variable of type `lazy T` (where `T` is any Java type) can hold a thunk which when evaluated will yield a value of type `T`. The existence of coercions between non-lazy and lazy types eliminates the need for explicit `delay` and `force` operations.

In this paper, we introduce LazyJ with several motivating examples and formalize the semantics of the language with Featherweight LazyJ, an extension of Featherweight Java. We also describe our implementation, which is built on top of the Polyglot extensible compiler framework.

1. Introduction

While certain kinds of programs can be written with astounding elegance and simplicity in languages with lazy (non-strict) semantics—for example, Turner's implementation of the sieve of Eratosthenes using infinite lists in [11]—state-of-the-art implementations of lazy languages usually perform considerably worse than those of eager (strict) languages. Therefore, it might be useful to design programming languages that can conveniently support both lazy and eager evaluation; that way, programmers may leverage the strengths of both approaches.

Some languages with eager semantics like ML [8] and Scheme [9] support a form of lazy evaluation through explicit `delay` and `force` constructs. While these constructs afford programmers the same amount of expressiveness as true lazy evaluation, using them is nowhere as convenient as writing lazy programs in a language with lazy semantics. Similarly, current support for eager evaluation in lazy languages such as Haskell [3] is practically non-existent.

This paper explores a novel approach to adding support for lazy evaluation to eager languages that is based on lazy types and the implicit generation of `delay` and `force` operations. We explore this idea in the context of LazyJ, an extension of the Java programming language that supports this approach to lazy evaluation. LazyJ extends Java with a `lazy` type qualifier which can be used in conjunction with any Java type. A value of type `lazy T` is a *thunk* which, when evaluated, will yield a value of type `T`.

The information provided by the types of LazyJ expressions is enough to relieve programmers of the burden of explicitly `force`ing and `delay`ing expressions. When the compiler finds an expression of a lazy type where an expression of a non-lazy type is expected, it automatically forces that expression. Conversely, when the compiler finds an expression of a non-lazy type where an expression of a lazy type is expected, it automatically delays that expression.

The rest of this paper is structured as follows. Section 2 introduces LazyJ with a number of motivating examples. Section 3 describes our compilation strategy, which translates LazyJ programs into semantically equivalent Java programs. Section 4 presents Featherweight LazyJ (FLJ), a formal model of LazyJ based on Featherweight Java [5]. Section 5 compares LazyJ with related work. Section 6 identifies future work, and section 7 concludes.

2. LazyJ by example

Note: All of the examples shown here can be compiled and run using our LazyJ implementation, available at http://www.cs.ucla.edu/~awarth/lazyj/lazyj.tar.gz.

```
class List {
  int head;
  lazy List tail;
  List(int h, lazy List t) {
    head=h;
    tail=t;
  }
}
```

Figure 1. Lazy lists in LazyJ.

Among the most important benefits of lazy evaluation is the ability to construct and process infinite data structures. Figure 1 shows how we might represent infinite lists of integers in LazyJ. Note that `List` is implemented in the same way we might implement ordinary linked lists, with one exception: the `tail` field is declared to have type `lazy List`. This means that the value of a `List`'s `tail` field is not actually the rest of the list, but instead a "recipe" [4] for its computation.

```
lazy List intsFrom(int n) {
  return new List(n, intsFrom(n+1));
}
```

Figure 2. A method that (lazily) builds an infinite list.

Figure 2 shows `intsFrom`, a method that uses our `List` class to generate a list of *all* integers starting from `n`. The method's implementation is as simple as it could possibly be: to construct a list of all integers starting from `n`, we build a *cons cell* whose head is `n`, and whose tail is the list of all integers starting from `n+1`. Note that the type of the expression being returned by this method, `new List(...)` is `List`, which differs from the method's declared return type, `lazy List`. This causes the LazyJ compiler to implicitly generate a `delay` operation that is applied to the expression, which results in the method's expected lazy behavior.

Now consider what happens when the expression `intsFrom(0).head` is evaluated. The type of the receiver in this expression—`intsFrom(0)`—is `lazy List`. LazyJ implicitly

forces receivers, and thus the body of intsFrom is evaluated, yielding the value new List(0, intsFrom(1)). Finally, the result of accessing this object's head field is 0. The implicit force operation inserted by the compiler is a result of LazyJ's typing relation, which asserts that lazy types do not contain any methods. Consequently, in order for this expression to typecheck the receiver must be coerced to the List type. (Refer to section 4 for more details on LazyJ's typing rules.)

Because recursive and mutually-recursive variable declarations are quite common in lazy programs, LazyJ permits the use of forward- and self-references inside expressions that are used to initialize lazy fields. As an example, consider the following field declaration, which initializes field ones with an infinitely long list of 1s:

```
lazy List ones=new List(1, ones);
```

(Forward- and self-references to *local variables* are not allowed for implementation-specific reasons; see section 3 for details.)

Lazy lists can be used to implement certain kinds of programs with great elegance and ease, as we shall see in the next two examples, taken from [11]. Other examples can be found in [11], [4], and [1]. Before we go into these examples, however, it is helpful for us to define a library of frequently used lazy list operations.

```
class Lib {
  static class List { ... }
  static lazy List intsFrom(int n) { ... }
  static lazy List first(int n, lazy List l) {
    if (n==0)
      return null;
    else
      return new List(l.head, first(n-1, l.tail));
  }
  static void print(lazy List l) {
    while (l!=null) {
      System.out.println(l.head);
      l=l.tail;
    }
  }
}
```

Figure 3. A LazyJ list library.

This library, shown in figure 3, includes the List class (discussed previously), the first method, which produces a list containing the first n elements of the list passed to it as an argument, and print, which prints a lists's elements to the console. Note that print was carefully written in order to permit the part of the list that has already been printed to be garbage-collected; this allows print to be used with infinite lists.

Generating prime numbers Figure 4 shows the complete source code for a LazyJ program that displays the first 1,000 prime numbers using an algorithm known as the sieve of Eratosthenes. Note that the static field primes holds the list of *all* prime numbers. We use the first method to extract the first 1,000 elements of this infinite list for the sole purpose of ensuring termination. Without this call, the program would happily keep printing prime numbers forever (well, until it exhausts the computer's available memory). The ability to separate generation from selection [6], as illustrated by this example, is one of the many strengths of lazy evaluation.

Lazy input and output While many programs do not require user interaction, most real-world applications must interface with the outside world. Lazy I/O can be used to cleanly separate the parts of a program that implement business logic from the parts that implement its user interface.

```
class Primes extends Lib {
  public static List filter(int n, lazy List l) {
    if (l.head%n==0)
      return filter(n, l.tail);
    else
      return new List(l.head, filter(n, l.tail));
  }
  public static lazy List sieve(lazy List l) {
    return new List(l.head, sieve(filter(l.head, l.tail)));
  }
  public static lazy List primes=sieve(intsFrom(2));
  public static void main(String[] args) {
    print(first(1000, primes));
  }
}
```

Figure 4. The sieve of Eratosthenes in LazyJ.

```
class InsurancePremiumCalculator {
  boolean ask(String question) {
    boolean ans;
    // display a window with question and yes/no
    // buttons, block until one is pressed
    ...
    return ans;
  }
  int calculatePremium() {
    if (ask("Do you smoke?"))
      return 1000;
    boolean drinker=ask("Do you drink?");
    if (ask("Do you sky-dive?") && !drinker)
      ...
    else if (drinker && ...)
      ...
    ...
  }
}
```

Figure 5. Using Java to calculate insurance premiums.

Consider, for example, an application that is used by an insurance company's telephone sales personnel to calculate the premiums of potential customers. The company wants customers to know their time is valuable (and, truth be told, save money on its telemarketers' hourly wages), and thus it is important to ask customers for no more information than is absolutely necessary for the premium calculation.

We might choose to implement this application in Java as shown in figure 5. However, this implementation has the disadvantage that the code that implements the insurance calculation logic is cluttered by user interface code, which makes it hard to read and not modular.

A LazyJ implementation of the same application is shown in figure 6. Note that the ask method's return type is now lazy boolean, although its body is unchanged. This enables the calculatePremium method to lazily ask for all of the information it may need *up front*; this information will only actually be requested if and when it is required by the premium calculation logic in premiumCalculatorLogic. In addition to making the code which implements the logic operations easier to read, programmers may now subclass InsurancePremiumCalculator and override its premiumCalculatorLogic method to change the business logic without having to worry about I/O operations.

```
class InsurancePremiumCalculator {
  boolean ask(String question) {
    boolean ans;
    // display a window with question and yes/no
    // buttons, block until one is pressed
    ...
    return ans;
  }
  int calculatePremium() {
    return premiumCalculatorLogic(
      ask("Do you sky-dive?"),
      ask("Do you smoke?"),
      ...,
      ask("Do you drink?")
    );
  }
  int premiumCalculatorLogic(lazy boolean skydiver,
                             lazy boolean smoker,
                             ...,
                             lazy boolean drinker) {
    if (smoker)
      return 1000;
    else if (skydiver && !drinker)
      ...
    else if (drinker && ...)
      ...
    ...
  }
}
```

Figure 6. Using lazy I/O to calculate insurance premiums.

3. Compilation

The LazyJ compiler[1] is built on top of the Polyglot extensible compiler framework [10]. Like other Polyglot-based language implementations, our compiler translates LazyJ programs into semantically-equivalent Java 1.4 programs that can be compiled and run using a standard Java compiler and virtual machine.

3.1 Implementation of `delay` and `force`

As noted in section 2, LazyJ uses type information in order to determine where `delay` and `force` operations are required. In this section, we discuss LazyJ's implementation of these operations.

```
package polyglot.ext.lazyj.runtime;

public abstract class Thunk {
  protected static final Object unforced=new Object();
  protected Object value=unforced;
  protected abstract Object _force() throws Exception;
  public synchronized Object force() {
    if (value==unforced)
      try { value=_force(); }
      catch (Exception e) { throw new LazyJError(e); }
    return value;
  }
}
```

Figure 7. The `Thunk` class.

Delayed expressions are represented as *thunks*: closure-like objects that consist of an unevaluated expression and the environment required to evaluate it. Figure 7 shows the `Thunk` class, which is the central part of LazyJ's runtime library.

`Thunk` is never instantiated directly; instead, delayed expressions are translated into anonymous classes that inherit from `Thunk`. For instance, the declaration

```
lazy String s="hello"+"world";
```

is translated to

```
polyglot.ext.lazyj.runtime.Thunk s=
  new polyglot.ext.lazyj.runtime.Thunk() {
    protected Object _force() throws Exception {
      return "hello"+"world";
    }
  };
```

In order to get the whole picture we must also view the translation of a use of `s`, such as

```
char c=s.charAt(3);
```

which is

```
char c=((String)s.force()).charAt(3);
```

Generally, when the LazyJ compiler determines that an expression must be `force`d, it generates a call to the associated thunk's `force` method and a typecast to convert the result of that call to the appropriate type. This cast is always safe and could be eliminated by making `Thunk` a generic class that is parameterized by the type of the delayed expression.

Note that because `Thunk`'s `force` method is `synchronized`, it is safe to write multi-threaded LazyJ programs. `_force()` does not need to be `synchronized` because it is `protected` and it is only called from `force`.

3.2 Caching

Lazy evaluation is a combination of normal-order evaluation and memoization: the only time a thunk's associated expression is ever evaluated is the first time the thunk is `force`d. The result of evaluating the expression is stored in the thunk's `value` field, which provides the result of subsequent `force`s. `value` is also used as an indicator that the thunk has not yet been `force`d. (Refer to figure 7 for the implementation of this caching mechanism.)

3.3 Boxing/Unboxing

When delaying an expression whose type is one of Java's primitive types (e.g., `int`, `float`, ...), the LazyJ compiler generates additional code that *boxes*/*unboxes* the expression in order to conform to `Thunk`'s interface. (This mechanism may no longer be necessary since Java 1.5 automatically boxes and unboxes values of primitive types.)

3.4 Capturing the environment of a delayed expression

Delayed expressions are seldom self-contained: more often than not, they reference local and global variables, formals, etc. Since there is no guarantee that an expression will be forced in the same scope in which it was created, it becomes necessary for a thunk to contain the environment (i.e., the list of name-value bindings) required for the evaluation of its associated expression.

Our implementation represents the name-value bindings that make up the environment required to evaluate a delayed expression as fields of its associated `Thunk` object. See figure 8 for an example. An interesting complication arises because Java's anonymous classes cannot refer to non-`final` local variables and arguments. In order to circumvent this restriction, our compiler generates `final` copies of such variables and references the copies instead of the

[1] Available at http://www.cs.ucla.edu/~awarth/lazyj/lazyj.tar.gz

```
class Test {
  String f="field";
  public void m() {
    String l="local";
    lazy String s=f+l;
  }
}
```

```
class Test {
  String f="field";
  public void m() {
    String l="local";
    final String _lazyj_final_l=l;
    polyglot.ext.lazyj.runtime.Thunk s=
      new polyglot.ext.lazyj.runtime.Thunk() {
        String l=_lazyj_final_l;
        public Object _force() throws Exception {
          return f+l;
        }
      };
  }
}
```

Figure 8. The `Test` class in LazyJ and its Java translation.

original variables inside the body of the thunk's `_force()` method. In our example, a `final` copy of `l` called `_lazyj_final_l` is generated by the LazyJ compiler.

This results in different semantics for delaying fields and local variables, since fields are delayed by reference, and local variables are delayed by value. A discussion of this issue is included in section 6.

3.5 Exception handling

```
package polyglot.ext.lazyj.runtime;

public class LazyJError extends Error {
  protected Exception source;
  public LazyJError(Exception s) { source=s; }
  public Exception getSource() { return source; }
}
```

Figure 9. The `LazyJError` class.

Since LazyJ allows expressions that may throw exceptions to be delayed, it must also provide a mechanism for handling those exceptions. The problem is that expressions are seldom `forced` in the same context in which they are delayed, and therefore it is certainly possible that the part of the program that `forces` an expression will not do so inside an appropriate `try/catch` block.

Currently, LazyJ gets around this problem by wrapping exceptions inside `LazyJError` objects and re-throwing them. This mechanism is implemented by `Thunk`'s `force` method, shown in figure 7. The `LazyJError` class is shown in figure 9. Since Java's `Errors` are not checked by the compiler, the programmer is free to handle `LazyJErrors` or not, whenever a thunk is forced.

Problems with this approach and plans for improving LazyJ's exception handling mechanism are discussed in section 6.

3.6 Method overloading

In Java 1.5, a class cannot have two methods called `m`, one of which takes a single argument of type `List<String>`, and another that takes a single argument of type `List<Integer>`. This restriction exists because the Java Virtual Machine (JVM) does not provide support for generics. Instead, generic types such as `List<String>` are compiled into ordinary (i.e., non-generic) types—`List`, in our example—and the necessary typecasts are generated by the compiler. This technique is called *type erasure*.

Java's erasure causes similar problems to any language which is implemented on top of Java. In LazyJ, the types `lazy String` and `lazy Integer` both compile to `polyglot.ext.lazyj.runtime.Thunk`, which results in problems with method overloading. Even if `Thunks` were generic (i.e., parameterized by the type of expression they yield when `forced`), the compilation of the Java translation would still fail due to Java's erasure.

4. Featherweight LazyJ

This section describes Featherweight LazyJ (FLJ), an extension of Featherweight Java (FJ) [5] that formalizes LazyJ. This formalism consists of three parts:

- FJ_{EDF}, an extension of FJ that supports explicit `delay` and `force` operations;
- a typing relation for FLJ;
- and finally, a translation that maps well-typed FLJ programs to semantically-equivalent FJ_{EDF} programs.

We now describe each of these in turn.

4.1 Featherweight Java with explicit `delay` and `force`

```
T  ::= Eager C | Delayed C
CL ::= class C extends C {T̄ f̄; K M̄}
K  ::= C(T̄ f̄) {super(f̄); this.f̄=f̄;}
M  ::= T m(T̄ x̄) {return t;}
t  ::= x | t.f | t.m(t̄) | new C(t̄) | (T)t |
       delay t | force t
v  ::= new C(v̄) | delay t
```

Figure 10. FJ_{EDF} syntax.

Featherweight Java with explicit `delay` and `force` (FJ_{EDF}) extends Featherweight Java (FJ) with explicit `delay` and `force` operations, as well as `Delayed` and `Eager` types. The grammar of the language is shown in figure 10. Note that while FJ types are class names, FJ_{EDF} types have the form `Eager C` or `Delayed C`, where `C` is a class name.[2] Note also that `delayed` expressions are values. This is necessary in order for lazy evaluation to work properly: if delayed expressions were not values, they could not be used as arguments to method or constructor calls given FJ's call-by-value semantics.

$$\frac{}{\texttt{T} <: \texttt{T}}$$

$$\frac{\texttt{T}_1 <: \texttt{T}_2 \qquad \texttt{T}_2 <: \texttt{T}_3}{\texttt{T}_1 <: \texttt{T}_3}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\ldots\}}{\texttt{Eager C} <: \texttt{Eager D}}$$

$$\frac{CT(\texttt{C}) = \texttt{class C extends D } \{\ldots\}}{\texttt{Delayed C} <: \texttt{Delayed D}}$$

Figure 11. FJ_{EDF} subtyping.

[2] FJ_{EDF}'s `Eager` types are equivalent to FJ types.

Figure 11 shows FJ_{EDF}'s subtyping relation, which essentially separates types into eager and delayed "families". An eager type cannot be a subtype of a delayed type, and vice-versa.

$$\frac{\texttt{x} : \texttt{T} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{Eager C}_0 \qquad \mathit{fields}(\texttt{C}_0) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\Gamma \vdash \texttt{t}_0.\texttt{f}_i : \texttt{T}_i} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{t}_0 : \texttt{Eager C}_0 \\ \mathit{mtype}(\texttt{m}, \texttt{C}_0) = \overline{\texttt{S}} \to \texttt{T} \\ \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{T}} \qquad \overline{\texttt{T}} <: \overline{\texttt{S}}\end{array}}{\Gamma \vdash \texttt{t}_0.\texttt{m}(\overline{\texttt{t}}) : \texttt{T}} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\texttt{C}) = \overline{\texttt{S}}\ \overline{\texttt{f}} \\ \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{T}} \qquad \overline{\texttt{T}} <: \overline{\texttt{S}}\end{array}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{t}}) : \texttt{Eager C}} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{Eager C}}{\Gamma \vdash (\texttt{T})\texttt{t}_0 : \texttt{T}} \quad \text{(T-CAST)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{Eager C}}{\Gamma \vdash \texttt{delay}\ \texttt{t}_0 : \texttt{Delayed C}} \quad \text{(T-DELAY)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{Delayed C}}{\Gamma \vdash \texttt{force}\ \texttt{t}_0 : \texttt{Eager C}} \quad \text{(T-FORCE)}$$

$$\frac{\begin{array}{c}\overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{Eager C} \vdash \texttt{t}_0 : \texttt{S}_0 \qquad \texttt{S}_0 <: \texttt{T}_0 \\ \mathit{CT}(\texttt{C}) = \texttt{class C extends D}\ \{\ldots\} \\ \mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{T}} \to \texttt{T}_0)\end{array}}{\texttt{T}_0\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\texttt{return}\ \texttt{t}_0;\}\ \texttt{OK in C}} \quad \text{(T-METHOD)}$$

$$\frac{\begin{array}{c}\texttt{K} = \texttt{C}(\overline{\texttt{T}}\ \overline{\texttt{g}}, \overline{\texttt{S}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{g}});\ \texttt{this}.\overline{\texttt{f}}{=}\overline{\texttt{f}};\} \\ \mathit{fields}(\texttt{D}) = \overline{\texttt{T}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \texttt{OK in C}\end{array}}{\texttt{class C extends D}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK}} \quad \text{(T-CLASS)}$$

Figure 12. FJ_{EDF} typing.

FJ_{EDF}'s typing relation is shown in figure 12. The rule T-Field asserts that only fields of eager values may be accessed. Similarly, T-Invk asserts that only methods of eager values may be called. T-New asserts that constructor calls always yield eager values. T-Cast ensures that only eager expressions may be typecast; delayed expressions are excluded because casts require type information to be available during the program's execution, and the type of a delayed expression is not known until the expression is `forced`. Note that there is no restriction on the destination type, which may be eager or lazy. T-Delay and T-Force ensure that arguments to `delay` and `force` are eager and delayed, respectively. Redundant `delay`s are not permitted since they are not needed by our translation. Consequently, `Delayed Delayed C`, for example, is not a valid type. T-Method performs the usual substitution on the body of the method and makes `this` an eager value (this is sensible since programmers are not permitted to call methods or access fields of delayed objects).

FJ_{EDF}'s auxiliary definitions are shown in figure 13. These are identical to those of FJ, and are shown here only for the sake of completeness. Note that the *override* relation requires the type of a method to be exactly the same as that of the method it overrides. This restriction makes it possible for the type system to track whether values will be eager or lazy, which enables the compiler to implicitly add `delay` and `force` operations where needed.[3]

[3] This restriction could be relaxed to covariant return types and contravariant argument types, as long as the laziness/eagerness of the types is conserved.

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C extends D}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{S}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{S}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C extends D}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{S m}(\overline{\texttt{S}}\ \overline{\texttt{x}})\ \{\texttt{return t;}\} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \overline{\texttt{S}} \to \texttt{S}}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C extends D}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m is not defined in}\ \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{D})}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C extends D}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{S m}(\overline{\texttt{S}}\ \overline{\texttt{x}})\ \{\texttt{return t;}\} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C extends D}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m is not defined in}\ \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{D})}$$

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{D}) = \overline{\texttt{S}} \to \texttt{S}_0 \text{ implies } \overline{\texttt{T}} = \overline{\texttt{S}} \text{ and } \texttt{T}_0 = \texttt{S}_0}{\mathit{override}(\texttt{m}, \texttt{D}, \overline{\texttt{T}} \to \texttt{T}_0)}$$

Figure 13. FJ_{EDF} auxiliary definitions.

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{(\texttt{new C}(\overline{\texttt{v}}))\texttt{.f}_i \to \texttt{v}_i} \quad \text{(E-PROJNEW)}$$

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t}_0)}{(\texttt{new C}(\overline{\texttt{v}}))\texttt{.m}(\overline{\texttt{u}}) \to [\overline{\texttt{x}} \mapsto \overline{\texttt{u}}, \texttt{this} \mapsto \texttt{new C}(\overline{\texttt{v}})]\texttt{t}_0} \quad \text{(E-INVKNEW)}$$

$$\frac{\texttt{Eager C} <: \texttt{T}}{(\texttt{T})(\texttt{new C}(\overline{\texttt{v}})) \to \texttt{new C}(\overline{\texttt{v}})} \quad \text{(E-CASTNEW)}$$

$$\frac{\texttt{t}_0 \to \texttt{t}_0'}{\texttt{t}_0.\texttt{f} \to \texttt{t}_0'.\texttt{f}} \quad \text{(E-FIELD)}$$

$$\frac{\texttt{t}_0 \to \texttt{t}_0'}{\texttt{t}_0.\texttt{m}(\overline{\texttt{t}}) \to \texttt{t}_0'.\texttt{m}(\overline{\texttt{t}})} \quad \text{(E-INVK-RECV)}$$

$$\frac{\texttt{t}_i \to \texttt{t}_i'}{\texttt{v}_0.\texttt{m}(\overline{\texttt{v}}, \texttt{t}_i, \overline{\texttt{t}}) \to \texttt{v}_0.\texttt{m}(\overline{\texttt{v}}, \texttt{t}_i', \overline{\texttt{t}})} \quad \text{(E-INVK-ARG)}$$

$$\frac{\texttt{t}_i \to \texttt{t}_i'}{\texttt{new C}(\overline{\texttt{v}}, \texttt{t}_i, \overline{\texttt{t}}) \to \texttt{new C}(\overline{\texttt{v}}, \texttt{t}_i', \overline{\texttt{t}})} \quad \text{(E-NEW-ARG)}$$

$$\frac{\texttt{t}_0 \to \texttt{t}_0'}{(\texttt{C})\texttt{t}_0 \to (\texttt{C})\texttt{t}_0'} \quad \text{(E-CAST)}$$

$$\frac{\texttt{t}_0 \to \texttt{t}_0'}{\texttt{force}\ \texttt{t}_0 \to \texttt{force}\ \texttt{t}_0'} \quad \text{(E-FORCE-ARG)}$$

$$\texttt{force}\ (\texttt{delay}\ \texttt{t}_0) \to \texttt{t}_0 \quad \text{(E-FORCEDELAY)}$$

Figure 14. FJ_{EDF} evaluation.

Figure 14 shows FJ_{EDF}'s small-step evaluation relation. The E-CastNew rule shows that only casts to eager types are allowed (there is no rule for casts to delayed types). The rules for `force` expressions (E-Force-Arg and E-ForceDelay) ensure that `force`'s argument is always evaluated before the operation is applied to it. There is no evaluation rule for `delay` expressions (after all, they are already values). However, in order to properly capture Java's static scoping, substitutions on `delay` expressions must also be performed on their associated expressions. This is analogous to the substitution of free variables in a lambda expression in the lambda calculus [5]. The other evaluation rules are identical to those of FJ.

We have encoded FJ_{EDF} in the Coq theorem prover [2] and used this encoding[4] to prove a type soundness theorem for FJ_{EDF} using the standard progress and preservation style.

THEOREM 4.1. *(Progress)* If $\vdash$ `t` : `T`, then either `t` is a value, `t` contains a subexpression of the form `(U)v` where $\vdash$ `v` : `S` and `S` $\not<:$`U`, or there exists some term `s` such that `t` $\rightarrow$ `s`.

THEOREM 4.2. *(Preservation)* If $\Gamma \vdash$ `t` : `T` and `t` $\rightarrow$ `s`, then there exists some type `S` such that $\Gamma \vdash$ `s` : `S` and `S` <: `T`.

Together these theorems imply that the evaluation of well-typed FJ_{EDF} programs cannot result in a type error.

4.2 FLJ typing

For simplicity, FLJ uses the same syntax as FJ_{EDF} (see figure 10). It is FLJ's typing relation—a relaxation of its FJ_{EDF} counterpart—that makes FLJ a more convenient medium for expressing lazy programs. This section describes FLJ's typing rules.

$$\frac{\texttt{T}_1 <: \texttt{T}_2}{\mathit{coercible}(\texttt{T}_1, \texttt{T}_2)} \quad \text{(C-SUB)}$$

$$\frac{\texttt{Eager C} <: \texttt{Eager D}}{\mathit{coercible}(\texttt{Eager C}, \texttt{Delayed D})} \quad \text{(C-EAGER)}$$

$$\frac{\texttt{Delayed C} <: \texttt{Delayed D}}{\mathit{coercible}(\texttt{Delayed C}, \texttt{Eager D})} \quad \text{(C-DELAYED)}$$

$$\mathit{getClass}(\texttt{Eager C}) = \texttt{C}$$

$$\mathit{getClass}(\texttt{Delayed C}) = \texttt{C}$$

Figure 15. FLJ's *coercible* relation and *getClass* function.

As was discussed in previous sections, LazyJ implicitly `force`s and `delay`s expressions as necessary. These implicit operations are formalized as coercions. Figure 15 shows the *coercible* relation, which specifies what coercions are supported by the language. If `C` is a subclass of `D`, then both `C` types (i.e., `Eager C` and `Delayed C`) are coercible to either of the `D` types (`Eager D` or `Delayed D`). FLJ's subtyping relation, which is used in the definition of *coercible*, is identical to that of FJ_{EDF} (see figure 11 for details).

Figure 16 shows the FLJ typing rules that are different from those in FJ_{EDF}. T-Field and T-Invk ignore the "eagerness" of receivers. This is accomplished by using the function *getClass*, which extracts the class of its FLJ type argument (this function is shown in figure 15). T-Invk and T-New use the *coercible* relation in order to relax FJ_{EDF}'s argument type checking rules. Finally, T-Cast also relaxes the restrictions of its FJ_{EDF} counterpart, allowing any well-typed term to be typecast to any type.

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{T} \qquad \mathit{fields}(\mathit{getClass}(\texttt{T})) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\Gamma \vdash \texttt{t}_0.\texttt{f}_i : \texttt{T}_i} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{t}_0 : \texttt{T}_0 \\ \mathit{mtype}(\texttt{m}, \mathit{getClass}(\texttt{T}_0)) = \overline{\texttt{S}} \rightarrow \texttt{T} \\ \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{T}} \qquad \mathit{coercible}(\overline{\texttt{T}}, \overline{\texttt{S}})\end{array}}{\Gamma \vdash \texttt{t}_0.\texttt{m}(\overline{\texttt{t}}) : \texttt{T}} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\texttt{C}) = \overline{\texttt{S}}\ \overline{\texttt{f}} \\ \Gamma \vdash \overline{\texttt{t}} : \overline{\texttt{T}} \qquad \mathit{coercible}(\overline{\texttt{T}}, \overline{\texttt{S}})\end{array}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{v}}) : \texttt{Eager C}} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{T}_0}{\Gamma(\texttt{T})\texttt{t}_0 : \texttt{T}} \quad \text{(T-CAST)}$$

Figure 16. FLJ typing (rules that are the same as in FJ_{EDF} omitted for brevity).

$$[\![\texttt{x}]\!] = \texttt{x} \quad \text{(TR-VAR)}$$

$$\frac{\Gamma \vdash \texttt{t}_0 : \texttt{T} \qquad \mathit{coerce}(\texttt{t}_0, \texttt{Eager}\ \mathit{getClass}(\texttt{T})) = \texttt{t}_0'}{[\![\texttt{t}_0.\texttt{f}]\!] = \texttt{t}_0'.\texttt{f}} \quad \text{(TR-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{t}_0 : \texttt{T} \qquad \mathit{coerce}(\texttt{t}_0, \texttt{Eager}\ \mathit{getClass}(\texttt{T})) = \texttt{t}_0' \\ \mathit{mtype}(\texttt{m}, \mathit{getClass}(\texttt{T})) = \overline{\texttt{S}} \rightarrow \texttt{S} \qquad \mathit{coerce}(\overline{\texttt{t}}, \overline{\texttt{S}}) = \overline{\texttt{t}'}\end{array}}{[\![\texttt{t}_0.\texttt{m}(\overline{\texttt{t}})]\!] = \texttt{t}_0'.\texttt{m}(\overline{\texttt{t}'})} \quad \text{(TR-INVK)}$$

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \qquad \mathit{coerce}(\overline{\texttt{t}}, \overline{\texttt{T}}) = \overline{\texttt{t}'}}{[\![\texttt{new C}(\overline{\texttt{t}})]\!] = \texttt{new C}(\overline{\texttt{t}'})} \quad \text{(TR-NEW)}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T} \qquad \mathit{coerce}(\texttt{t}, \texttt{Eager}\ \mathit{getClass}(\texttt{T})) = \texttt{t}'}{[\![(\texttt{Eager C})\texttt{t}]\!] = (\texttt{Eager C})\texttt{t}'} \quad \text{(TR-CASTEAGER)}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T} \qquad \mathit{coerce}(\texttt{t}, \texttt{Eager}\ \mathit{getClass}(\texttt{T})) = \texttt{t}'}{[\![(\texttt{Delayed C})\texttt{t}]\!] = \texttt{delay}\ ((\texttt{Eager C})\texttt{t}')} \quad \text{(TR-CASTDELAYED)}$$

$$[\![\texttt{delay t}]\!] = \texttt{delay}\ [\![\texttt{t}]\!] \quad \text{(TR-DELAY)}$$

$$[\![\texttt{force t}]\!] = \texttt{force}\ [\![\texttt{t}]\!] \quad \text{(TR-FORCE)}$$

Figure 17. Translating from FLJ to FJ_{EDF}.

$$\frac{\Gamma \vdash [\![\texttt{t}]\!] : \texttt{Eager C}}{\mathit{coerce}(\texttt{t}, \texttt{Eager D}) = [\![\texttt{t}]\!]} \quad \text{(C-EAGEREAGER)}$$

$$\frac{\Gamma \vdash [\![\texttt{t}]\!] : \texttt{Delayed C}}{\mathit{coerce}(\texttt{t}, \texttt{Eager D}) = \texttt{force}\ [\![\texttt{t}]\!]} \quad \text{(C-DELAYEDEAGER)}$$

$$\frac{\Gamma \vdash [\![\texttt{t}]\!] : \texttt{Eager C}}{\mathit{coerce}(\texttt{t}, \texttt{Delayed D}) = \texttt{delay}\ [\![\texttt{t}]\!]} \quad \text{(C-EAGERDELAYED)}$$

$$\frac{\Gamma \vdash [\![\texttt{t}]\!] : \texttt{Delayed C}}{\mathit{coerce}(\texttt{t}, \texttt{Delayed D}) = [\![\texttt{t}]\!]} \quad \text{(C-DELAYEDDELAYED)}$$

Figure 18. Translating and coercing expressions.

4.3 Translating FLJ to FJ_{EDF}

Figures 17 and 18 show the mutually-recursive relations $[\![\bullet]\!]$ and *coerce*, respectively, which are used in our translation from FLJ to equivalent FJ_{EDF}.

The translation relation $[\![\bullet]\!]$ maps well-typed FLJ expressions to FJ_{EDF} expressions.This relation ensures that the translated version of the expression will adhere to FJ_{EDF}'s typing rules. For example, the rules TR-Field and TR-Invk coerce receivers to eager values (which results in the insertion of `force`s where appropriate). Similarly, TR-Invk and TR-New coerce the arguments of methods and constructors to their appropriate types. Casts are translated by the rules TR-CastEager (which is used when the cast's associated type is eager), and TR-CastDelayed (which is used when the cast's associated type is delayed). TR-CastEager simply `force`s the cast's associated expression. TR-CastDelayed first `force`s the expression, then casts it to the eager version of the cast's associated type, and finally `delay`s the result. `force`ing is necessary here because FJ_{EDF}'s typecasts are only defined for eager types (this restriction exists because the only way to determine the type of an expression at runtime—this information is needed during the execution of a cast operation—is to `force` to get the expression into the form `new C(...)`). Finally, TR-Force and TR-Delayed simply translate `force` and `delay`'s arguments, respectively.

Our translation uses the *coerce* relation (shown in figure 18) in order to force or delay the translations of subexpressions so that the resulting translated expression is well-typed in FJ_{EDF}. For example, the translation of the FLJ-like expression `new List(0, null)` is `new List(0, delay null)`. Here, the rule TR-New is applied to the whole expression, which results in the coercion of `null` to `lazy List`. This coercion in turn uses the rule C-EagerDelayed, since `null` has type `Eager Object`.

Another application of *coerce* is in the translation of method bodies. Consider, for example, the method `Delayed C m() { return new D(); }`, where `D` is a subclass of `C`. The correct translation of `m`'s body is obtained with *coerce*(`new D()`, `Delayed C`). This example illustrates why, in each of the rules that implement *coerce* (see figure 18), the class name of $[\![\mathtt{t}]\!]$'s type and the class name of the desired type are not the same.

4.4 FLJ soundness?

We designed FJ_{EDF}, proved its soundness using the Coq theorem prover, and designed the translation from FLJ to FL_{EDF} so that we could prove the soundness of FLJ—a task which we have yet to complete. The particular theorem we would like to prove is that every well-typed FLJ program can be translated to a well-typed FJ_{EDF} program; in other words, that our translation is sound (i.e., both complete and well-typed).

5. Related Work

Both Scheme [9] and ML [8] include support for lazy evaluation via explicit `delay` and `force` operations. Although the expressiveness of these constructs is identical to that of LazyJ's lazy types, they are less convenient for the programmer.

In Scheme, the form `(delay <expression>)` is defined to have the same semantics as the expression `(make-promise (lambda () <expression>))`, where `make-promise` performs the same work of caching the result of the evaluation that is performed in LazyJ by `Thunk`'s `force` method. `force` is implemented as `(lambda (x) (x))` (i.e., it simply calls the function returned by `make-promise`). Because of Scheme's dynamic typing, programmers must be careful to avoid redundant `delay`s and `force`s, which at best can result in performance degradation, and at worst could cause type errors at run-time (e.g., not applying enough `force`s to an expression which has been `delay`ed multiple times before using it).

ML provides a `delay` function of type `(unit -> 'a) -> 'a susp` to create suspensions (i.e., thunks), which provides similar functionality to that of Scheme, although it is a bit less convenient to use, since `delay`'s argument is a function that takes the dummy value `()` and returns the value to be `delay`ed instead of the value itself. This complication is required because of the language's call-by-value semantics and the absence of a mechanism for defining new syntax such as Scheme's `define-syntax`. The object returned by `delay` uses side effects in order to cache the result of the last `force` (ML's `force` has type `'a susp -> 'a`). LazyJ's approach can be thought of as the opposite of ML's: instead of using `delay` and `force` and having the compiler infer the types of expressions, LazyJ programmers annotate the types of variables and methods as `lazy` or not, and the compiler generates the necessary `delay` and `force` operations. We believe that LazyJ's approach makes for more readable code. For example, using explicit `delay` and `force` operations, the `intsFrom` method from section 2 would be implemented as follows:

```
static lazy List intsFrom(int n) {
  return delay new List(n, intsFrom(n+1);
}
```

Clients of `intsFrom` would have to force its result before using it. Consider the following version of `Lib`'s `print` method using explicit `delay`s and `force`s:

```
static void print(lazy List l) {
  while ((force l)!=null) {
    System.out.println((force l).head);
    l=l.tail;
  }
}
```

We believe that the presence of `delay` and `force` operations unnecessarily lowers a program's level of abstraction. Moreover, our approach is a more natural fit for languages in which the types of variables and functions/methods are always explicitly declared, such as Java.

```
List intsFrom(int n) {
  return new List(n, intsFrom(n+1));
}
```

Figure 19. A version of `intsFrom` written in Wadler's *odd* style of lazy evaluation.

In [13], Wadler et al. describe two different styles of lazy evaluation: *odd*, which is easy to encode in languages that provide explicit `delay` and `force` operations but results in too much evaluation (shown in figure 19), and *even*, which properly delays evaluation but is more difficult to encode in these languages (shown in figure 2). The paper also describes an extension of ML that includes a *lazy* modifier that can be used in (i) datatype declarations, which makes constructors return suspensions, and (ii) function declarations, which results in delaying the evaluation of the function's body. While Wadler's lazy functions can be expressed easily in LazyJ by declaring a method's return type as `lazy`, his language provides no mechanism for assigning lazy semantics to selected arguments of a method, which is easily expressed in LazyJ. The authors claim that their language allows programmers to use the even style with ease, although it is difficult to write in the odd style. LazyJ allows for both styles to be easily expressed, as shown

[4] Our encoding is based on Stephanie Weirich's encoding of Featherweight Java in [14], and is available at http://www.cs.ucla.edu/ awarth/lazyj/fj-edf.v

in figures 19 and 2. By simply changing the method's return type from `lazy List` to `List`, we obtain a version of the method which always creates a `List` (node) object when called, regardless of whether or not that object is actually required (Wadler's odd style).

Lambda4J [7], a Java library for functional programming, provides lazy lists and associated operations such as `map`, `fold`, etc. While these could certainly be used to implement `intsFrom` and most of the examples shown in section 2, Lambda4J does not support other forms of lazy evaluation (e.g., anything that does not involve lists) and therefore cannot be used to implement many idioms supported by LazyJ (one such example is shown in figure 6).

6. Discussion and Future Work

The LazyJ compiler creates `final` copies of local variables that are used in a delayed expression. Unfortunately, this may affect the program's semantics. It is certainly possible that the value of a local variable will change after it is used in an expression that is delayed. Because the code in the thunk does not reference the variable itself, but instead its copy, this change will have no effect on the value obtained from forcing the thunk! A different approach in which non-`final` variables are not permitted to be delayed (whereby an attempt to do so would result in a compilation error) may be more desirable. This would result in no loss of expressiveness, as the programmer may still elect to manually create `final` copies of such variables, and the semantics of the program would be more straightforward.

The practice of wrapping exceptions in `LazyJErrors` is not satisfactory for several reasons. First, since force operations are implicit, it is not obvious to the programmer where `try/catch` blocks should be placed to handle the exceptions which may result from forcing a thunk. Even when the programmer knows where he may want to handle such exceptions, it is not obvious what those exceptions might be—and the compiler provides no help whatsoever! A much better way of dealing with exceptions might be to augment `lazy` types with optional `throws` clauses, similar to those of methods. This would enable the typechecker to ensure that the exceptions which may be thrown when a delayed expression is forced are handled appropriately. The inclusion of this feature is planned for a future release of LazyJ.

One of design choices we faced while formalizing LazyJ was whether to use coercions or to modify the subtyping relation so that `Eager C` is a subtype of `Delayed C` and vice-versa. We chose to use coercions because of the potential complications that could be introduced by a cyclic subtyping relation [12]. Since it is rather easy to forget to place a coercion in a particular typing rule (and such omissions are difficult to catch!), it would be useful to prove that the typing relation shown in this paper is equivalent to one with a cyclic subtyping relation and no coercions.

Finally, while mixing laziness and side effects can be useful (as demonstrated by our lazy I/O example in section 2), it can also make programs difficult to understand. It might be helpful to allow methods to be declared to be `pure`, meaning that they are side effect free (this could easily be checked by the compiler). This would enable the compiler to produce warnings or error messages for calls to non-`pure` methods found in lazy methods (i.e. those whose return types are lazy).

7. Conclusion

We have described LazyJ, an extension of the Java programming language in which both eager and lazy evaluation can be expressed conveniently with minimal additional syntax (a single keyword!). LazyJ enables Java programmers to make use of a variety of techniques involving the use of lazy evaluation, such as partially generating potentially infinite data structures on demand. These techniques can improve modularity and provide for better abstractions, which in turn can enhance programmer productivity. We have also described our implementation, which translates LazyJ programs to Java. Finally, we have formalized LazyJ and have begun working on a proof of soundness for the language.

8. Acknowledgements

The author wishes to thank Jeff Fischer for his collaboration on the Coq soundness proof of $\mathrm{FJ}_{\mathrm{EDF}}$, and Todd Millstein, Stephen Murrell, Paul Eggert, and the anonymous reviewers for comments on numerous drafts of this work.

References

[1] Richard Bird. *Introduction to Functional Programming*. Pearson Education, 1998.

[2] Coq home page. `http://coq.inria.fr/`.

[3] Peyton Jones et al. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[4] Peter Henderson. *Functional Programming – Application and Implementation*. Series in Computer Science. Prentice/Hall International, 1980.

[5] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[6] Simon Peyton Jones. Wearing the hair shirt: a retrospective on Haskell. Invited talk at POPL 2003.

[7] Lambda4J home page. `http://www.nongnu.org/lambda4j/`.

[8] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[9] IV N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, Jr. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[10] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of CC 2003: 12th International Conference on Compiler Construction*. Springer-Verlag, April 2003.

[11] D. A. Turner. Recursion equations as a programming language. In J. Darlington, D. Turner, and P. Henderson, editors, *Functional Programming and Its Applications: An Advanced Course*, New York, NY, USA, 1982. Cambridge University Press.

[12] Philip Wadler. Personal communication, January 2005.

[13] Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language without even being odd. In *Workshop on Standard ML*, Baltimore, 1998.

[14] Stephanie Weirich. Proof of the soundness of featherweight java using Coq. `http://www.cis.upenn.edu/proj/plclub/wiki-static/fj-coq.tar.gz`.