TRANSPARENT PROXIES FOR JAVA FUTURES

Polyvios Pratikakis
Jaime Spacco
Michael Hicks

University of Maryland, College Park
Concurrent Programming

- Threads of execution: Thread objects running in parallel
- Asynchronous method invocations: Methods can be called asynchronously
- e.g:
  
  Main thread:
  
  ```
  o = new O();
  o.m(); //async
  ```
Concurrent Programming

- Threads of execution: Thread objects running in parallel
- Asynchronous method invocations: Methods can be called asynchronously
- e.g:

  Main thread:  
  ```
  o = new O();
  o.m(); //async
  ```

  Child thread:  
  ```
  o.m();
  ```
Concurrent Programming

- Threads of execution: Thread objects running in parallel
- Asynchronous method invocations: Methods can be called asynchronously

```
// e.g:
Main thread:               Child thread:
o = new O();
o.m(); //async ⇒ o.m();
...```

Transparent Proxies for Java Futures – p. 2/51
Futures

- What happens with returned value?
- “Future” or “promise”: a placeholder for the result
- “Claim” a future:
  If the result is not available, wait for it
- Futures are proxies. Other examples:
  - Suspensions (lazy invocation)
  - Remote objects
  - Other wrappers
Java Transparent Proxy Framework

- Static analysis and program transformation
  - Based on *qualifier inference* system
  - Formalization and proof of soundness
- Implementation of Futures via async methods
  - Also lazy invocations, other applications
- Benefits
  - Simple programming model
  - Can improve application performance
(future e) means e executes in parallel

Lisp is functional and dynamically typed

No need for the programmer to insert claims: The runtime system checks every access. If it is a future, claim before access

Programmer only inserts future notations

Futures are transparent
Java Futures not Transparent

- Java is statically typed
- Futures in JSR166 must be claimed explicitly:

```java
public interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    ...
}
```
Programming Overhead

To convert a method invocation to asynchronous:

- Change the call site to be asynchronous
- Change the type of the result to **Future**
- Change the type of variables to which the result flows
- Insert claims

It is usual to claim early to avoid rewriting a lot of code

**Question**: can we do this automatically?
Type Qualifiers

- Qualifiers refine the meaning of types
- `final Integer is to Integer like Future<String> is to String`
- “Proxyness” is a type qualifier: `proxy` or `nonproxy`. If `x` has type
  - `proxy String` then `x could be a proxy`
  - `nonproxy String` then `x is not a proxy`
- `nonproxy ≤ proxy`
Automatic Transformation

- Use qualifier inference to determine where proxies flow
- Transform program based on results
  - Rewrite proxy types
  - Insert claims whenever a proxy is used concretely
Example: Initial Program

```java
procRequest(Socket sock) {
    Buffer in = readBuf(sock);
    Request req = translate(in);
    Buffer out = process(req);
    writeBuf(sock, out);
}
Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Async Calls

```java
procRequest(Socket sock) {
    Buffer in = readBuf(sock);

    Request req = translate(in);

    Buffer out = process(req);

    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    Buffer in = proxy @readBuf(sock);

    Request req = proxy @translate(in);

    Buffer out = proxy @process(req);

    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ... 
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    Request req = proxy @translate(in);
    Buffer out = proxy @process(req);
    writeBuf(sock, out);
}
Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);

    Request req = proxy @translate(in);

    Buffer out = proxy @process(req);

    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);

    Request req = proxy @translate(proxy in);

    Buffer out = proxy @process(req);

    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);

    Request req = proxy @translate(proxy in);

    Buffer out = proxy @process(req);

    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ... 
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    proxy Request req = proxy @translate(proxy in);
    Buffer out = proxy @process(req);
    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    proxy Request req = proxy @translate(proxy in);
    Buffer out = proxy @process(req);
    writeBuf(sock, out);
}
Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    proxy Request req = proxy @translate(proxy in);
    Buffer out = proxy @process(proxy req);
    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    Buffer in = readBuf(sock);
    Request req = translate(in);
    Buffer out = process(req);
    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    proxy Request req = proxy @translate(proxy in);
    proxy Buffer out = proxy @process(proxy req);
    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);

    proxy Request req = proxy @translate(proxy in);

    proxy Buffer out = proxy @process(proxy req);

    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    proxy Request req = proxy @translate(proxy in);
    proxy Buffer out = proxy @process(proxy req);
    writeBuf(sock, proxy out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ... 
    return result;
}
```
Example: Qualified Types

```java
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    proxy Request req = proxy @translate(proxy in);
    proxy Buffer out = proxy @process(proxy req);
    writeBuf(sock, proxy out);
}
Request translate(Buffer in) {
    Request result;
    ... in.foo() ... 
    return result;
}
```
procRequest(Socket sock) {
    proxy Buffer in = proxy @readBuf(sock);
    proxy Request req = proxy @translate(proxy in);
    proxy Buffer out = proxy @process(proxy req);
    writeBuf(sock, proxy out);
}

Request translate(proxy Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
Example: Future Transformation

```java
procRequest(Socket sock) {
    Buffer in = readBuf(sock);

    Request req = proxy translate(in);

    Buffer out = proxy process(req);

    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    ... in.foo() ...
    return result;
}
```
Example: Future Transformation

```
procRequest(Socket sock) {

    Object in = new Proxy{
        private Object result;
        public void run() {
            result = readBuf(sock); }
        public synchronized Object get(){
            ... return result; }
    }();
    Executor.run((Runnable)in);

    Request req = @translate(in);
    Buffer out = @process(req);
    writeBuf(sock,out);
}
```
procRequest(Socket sock) {
    Object in = new Proxy{
        private Object result;
        public void run() {
            result = readBuf(sock); }
        public synchronized Object get(){
            ... return result; }
    }();
    Executor.run((Runnable)in);
    Object req = new Proxy{... translate(in) ...}
    Object out = new Proxy{... process(req) ...}
    writeBuf(sock,out);
}
Example: Qualified types

```java
Request translate(Buffer in) {
    Request result;
    ...
    in.foo();
    ...
    return result;
}
```
Example: Qualified types

```java
Request translate(Buffer in) {
    Request result;
    ...
    in.foo();
    ...
    return result;
}
```

concrete use
Example: Qualified types

```java
Request translate(Buffer in) {
    Request result;
    ...
    (nonproxy in).foo();
    ...
    return result;
}
```
Example: Qualified types

```java
Request translate(Buffer in) {
    Request result;
    ...
    (nonproxy in).foo();
    ...
    return result;
}
```
Example: Qualified types

```java
Request translate(Buffer in) {
    Request result;
    ...
    (nonproxy in).foo();
    ...
    return result;
}
```
Example: Future Transformation

```java
Request translate(Object inF) {
    Request result;

    ...

    Buffer in = (Buffer)
        (inF instanceof Proxy ?
          inF.get() :
          inF);

    in.foo();

    ...

    return result;
}
```
Other Uses of Proxies

- User can define:
  - What is a concrete usage (places where an unwrapping would be necessary)
  - Where proxies are created
  - What is a claim

- E.g. suspensions (lazy proxies):
  - Concrete usage: same as futures
  - Proxy creation: lazy invocations
  - Claim: evaluate the invocation and return its result
Other Uses of Qualifier Inference

- tainted / untainted qualifiers (for security)
- null / non-null qualifiers: can be used to enforce stronger interfaces
- stack / heap allocation for objects: Figure out which objects can be stack-allocated, and if/when they need to be moved to the heap
Formalism: Checking System

- $FJ_Q$: Featherweight Java + Qualifiers: checking system
  - If the programmer annotated every type with a qualifier by hand, would it be correct?

- Proof of soundness for $FJ_Q$:
  - If the annotated program typechecks, then everything is claimed before used
Formalism: Inference System

- $FJ^i_Q$: Inference system that produces $FJ_Q$ programs
  - Find the values of the proxy qualifier automatically

- Correctness of inference:
  - If $FJ^i_Q$ finds a solution, the resulting annotated program typechecks in $FJ_Q$
Formalism: Features

- Use set types to improve precision:

```java
class A { ... }
class B extends A { ... }
class C extends A { ... }
...
A var = new B();
```

Variable `var` has type `{B}^A`

- Selective flow-sensitive coercions from proxy to nonproxy
Implementation

- Uses Soot
  - Jimple: Three-address code SSA-like representation of bytecode
  - Spark: Points-to analysis engine
- Extends Spark’s Andersen-style points-to analysis with selective flow-sensitivity
- Bytecode to bytecode transformation
- Handles the full Java language (exceptions, JNI, reflection, etc)
Performance

- RMI peer-to-peer application
- Each peer searches in the network to find service providers
Service findService(LocalPeer self, String sName) {
    Service s = self.getService(sName);
    if (s != null) return s;
    else {
        self.forward(
            new FindServiceMessage(sName));
        return getRemoteService(self, sName);
    }
}

Service findService(LocalPeer self, String sName) {
    Service s = self.getService(sName);
    if (s != null) return s;
    else {
        self.forward(
            new FindServiceMessage(sName));
        return getRemoteService(self, sName);
    }
}
Example

```java
Service findService(LocalPeer self, String sName) {
    Service s = self.getService(sName);
    if (s != null) return s;
    else {
        self.forward(
            new FindServiceMessage(sName));
        return getRemoteService(self, sName);
    }
}
```
Service findService(LocalPeer self, String sName) {
    Service s = self.getService(sName);
    if (s != null) return s;
    else {
        self.forward(
            new FindServiceMessage(sName));
        return getRemoteService(self, sName);
    }
}
Example

```java
Service findService(LocalPeer self,
                     String sName) {
    Service s = self.getService(sName);
    if (s != null) return s;
    else {
        @self.forward(
            new FindServiceMessage(sName));
        return $getRemoteService(self, sName);
    }
}
```

$ denotes a lazy invocation
## Analysis Performance

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Time (s)</th>
<th>classes analyzed</th>
<th>classes w/ fut.</th>
<th>changed</th>
<th>claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI</td>
<td>73</td>
<td>1324</td>
<td>27</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>FS</td>
<td>92</td>
<td>1324</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SPARK</td>
<td>66</td>
<td>1320</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Related Work

- Futures and async methods
  - MultiLISP (Halstead et al.)
  - Promises (Liskov and Shrira)
  - Touch elimination (Flanagan and Felleisen)
  - Polyphonic C# (Benton et al.)
  - Lazy Task Creation (Halstead, Frigo et al.)
  - SCOOP/Eiffel (Compton)
  - Async RMI (Raje et al, Sysala et al.)
Related Work

- Static Analysis
  - Points-to analysis (many)
  - Qualifier inference (Foster et al.)
  - Value flow analysis (Heintze and Tardieu)
Future Work

- Incorporate into a general framework for qualifiers in Java
  - Arbitrary qualifier partial order
  - Context-sensitive analysis
  - Faster algorithm

- Incremental analysis
  - Only re-run analysis for files affected by local change
  - Don’t reanalyze the library every time
Summary

- Framework for programming with proxies
  - Simplifies programming process
  - Avoids violations of transparency
- Novel static analysis
  - Extends qualifier inference with flow-sensitive coercions
  - Has additional applications (Stack-allocated objects, not-null types, etc)
- Formalization and proof of soundness
- Prototype implementation