Implicit Genericity

Jarosław Kuśmierek$^1$ $^2$

$^1$Warsaw University
$^2$Google

Software Composition 2009
Zurich - 3 July 2009
What is our concern

- Reusability.
- Flexibility.
- Support for un-anticipated changes.
What is our concern

- Reusability.
- Flexibility.
- Support for un-anticipated changes.
What is our concern

- Reusability.
- Flexibility.
- Support for un-anticipated changes.
What this talk will be about?

- What kind of un-anticipated changes do we want to support?
  - How do we want to support them?
  - How this approach can be implemented?
- What kind of static reasoning about such changes we want to enable?
What this talk will be about?

- What kind of un-anticipated changes do we want to support?
- How do we want to support them?
- How this approach can be implemented?
- What kind of static reasoning about such changes we want to enable?
What this talk will be about?

- What kind of un-anticipated changes do we want to support?
- How do we want to support them?
- How this approach can be implemented?
- What kind of static reasoning about such changes we want to enable?
What this talk will be about?

- What kind of un-anticipated changes do we want to support?
- How do we want to support them?
- How this approach can be implemented?
- What kind of static reasoning about such changes we want to enable?
What kind of changes we want to support

```java
public class LoginDialog extends Dialog {
    Button btnDoLogin;
    Socket s;
    ...
    public LoginDialog(String hostName, ...)
    { s = new Socket(hostName);
        btnDoLogin = new Button("Login");
        ...
    }
}
```
public class LoginDialog extends Dialog {
    Button btnDoLogin;
    Socket s;
    
    public LoginDialog(String hostName, ...) {
        s = new Socket(hostName);
        btnDoLogin = new Button("Login");
    }
}
Possible solutions now

Design solutions

- Inheritance and overriding of the whole method / constructor.
- Factory methods.

Those solutions require a significant amount of work or a priori design

Language features

- First-class generics
- Aspect Oriented Programming
- Classboxes
- Context Oriented Programming

Non trivial, often costly
Possible solutions now

Design solutions

- Inheritance and overriding of the whole method / constructor.
- Factory methods.

Those solutions require a significant amount of work or a priori design.

Language features

- First-class generics
- Aspect Oriented Programming
- Classboxes
- Context Oriented Programming

Non trivial, often costly
Possible solutions now

Design solutions

- Inheritance and overriding of the whole method / constructor.
- Factory methods.

Those solutions require a significant amount of work or a priori design

Language features

- First-class generics
- Aspect Oriented Programming
- Classboxes
- Context Oriented Programming

Non trivial, often costly
Possible solutions now

**Design solutions**
- Inheritance and overriding of the whole method / constructor.
- Factory methods.

Those solutions require a significant amount of work or *a priori* design

**Language features**
- First-class generics
- Aspect Oriented Programming
- Classboxes
- Context Oriented Programming

Non trivial, often costly
Possible solutions now

**Design solutions**
- Inheritance and overriding of the whole method / constructor.
- Factory methods.

Those solutions require a significant amount of work or *a priori* design

**Language features**
- First-class generics
- Aspect Oriented Programming
- Classboxes
- Context Oriented Programming

Non trivial, often costly
Possible solutions now

Design solutions
- Inheritance and overriding of the whole method / constructor.
- Factory methods.

Those solutions require a significant amount of work or a priori design

Language features
- First-class generics
- Aspect Oriented Programming
- Classboxes
- Context Oriented Programming

Non trivial, often costly
ImpliJava

Java extended with Implicit Genericity
General idea

We add one construct: **redefinition** operator.

It allows the user to replace seamlessly one class with a **simulating** one, within a given scope.

Example

```java
with (Socket -> MySSLSocket)
with (Button -> PadLockButton)
{
    obj = new LoginDialog(...);
}
```
What are the conditions?

For a class $C$ to simulate a class $B$ we need that:

- $C$ is a subclass of $B$;
- for each constructor in $B$, there is a constructor in $C$ with:
  - a compatible parameters list (supertypes);
  - a compatible throws list (subtypes).
For a class $C$ to **simulate** a class $B$ we need that:

- $C$ is a subclass of $B$;
- for each constructor in $B$, there is a constructor in $C$ with:
  - a compatible parameters list (supertypes);
  - a compatible throws list (subtypes).
What are the conditions?

For a class \( C \) to simulate a class \( B \) we need that:

- \( C \) is a subclass of \( B \);
- for each constructor in \( B \), there is a constructor in \( C \) with:
  - a compatible parameters list (supertypes);
  - a compatible throws list (subtypes).
For a class $C$ to simulate a class $B$ we need that:

- $C$ is a subclass of $B$;
- for each constructor in $B$, there is a constructor in $C$ with:
  - a compatible parameters list (supertypes);
  - a compatible throws list (subtypes).

If those conditions are met, class $B$ can be replaced with class $C$ in any scope, using with $(B \rightarrow C)$ ...
Implementation

How to implement it on top of existing VM?
ImpliJava has a simple and efficient implementation schema as a translation to Java:

- We store per-thread mapping from classes to their active replacements.
- Each redefinition operator adds to that list.
- Each `new` expression is replaced with the lookup to this per-thread mapping and a call to an obtained `construction-method`.
- ... Plus a special treatment of the `Thread` class ...
ImpliJava has a simple and efficient implementation schema as a translation to Java:

- We store per-thread mapping from classes to their active replacements.
- Each redefinition operator adds to that list.
- Each `new C` expression is replaced with the lookup to this per-thread mapping and a call to an obtained `construction-method`.
- ... Plus a special treatment of the `Thread` class ...
ImpliJava has a simple and efficient implementation schema as a translation to Java:

- We store per-thread mapping from classes to their active replacements.
- Each redefinition operator adds to that list.
- Each `new C` expression is replaced with the lookup to this per-thread mapping and a call to an obtained `construction-method`.
- ... Plus a special treatment of the `Thread` class ...
ImpliJava has a simple and efficient implementation schema as a translation to Java:

- We store per-thread mapping from classes to their active replacements.
- Each redefinition operator adds to that list.
- Each `new` expression is replaced with the lookup to this per-thread mapping and a call to an obtained `construction-method`.
- ... Plus a special treatment of the `Thread` class ...
ImpliJava has a simple and efficient implementation schema as a translation to Java:

- We store per-thread mapping from classes to their active replacements.
- Each redefinition operator adds to that list.
- Each `new C` expression is replaced with the lookup to this per-thread mapping and a call to an obtained `construction-method`.
- ... Plus a special treatment of the `Thread` class ...
Thanks to the fact that ImpliJava only modifies method creation, its impact is negligible in practical cases.

<table>
<thead>
<tr>
<th>test case</th>
<th>Java time</th>
<th>ImpliJava time</th>
<th>Interlacing time</th>
<th>cost</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 fields</td>
<td>1022</td>
<td>1134</td>
<td>2512</td>
<td>11%</td>
<td>145%</td>
</tr>
<tr>
<td>4 fields</td>
<td>4790</td>
<td>4940</td>
<td>8497</td>
<td>3.1%</td>
<td>77%</td>
</tr>
<tr>
<td>Socket</td>
<td>1806</td>
<td>1872</td>
<td>2309</td>
<td>3.6%</td>
<td>27%</td>
</tr>
<tr>
<td>JButton</td>
<td>5515</td>
<td>5508</td>
<td>6688</td>
<td>0%</td>
<td>21%</td>
</tr>
</tbody>
</table>
Static verification

Static verification of redefinitions

Can we verify the influence of a given redefinition?
public class ServiceLocator
{
    Socket fSocket;
    public Socket getSocket { return fSocket; }
    public Socket connect() {... return new Socket(...);}
}

...

// object created outside
ServiceLocator loc = new ServiceLocator(); // of the redefinition
...

with (Socket -> MySSLSocket)
{ MySSLSocket s1 = loc.getSocket(); // type error
  MySSLSocket s2 = loc.connect(); // type OK
}

Does a redefinition influence the type of an expression?
How to verify that?

Concept

The effect of a redefinition depends on the moment of object creation, w.r.t. the scope of the given redefinition.
How to verify that?

Concept

The effect of a redefinition depends on the moment of object creation, w.r.t. the scope of the given redefinition.

Solution

We add annotations expressing relations between the creation of an object and two actions:

- the creation of a parent object (method target, field owner etc.) - denoted using “+”;
- the execution of a method - denoted using “*”.
How to verify that?

Concept
The effect of a redefinition depends on the moment of object creation, w.r.t. the scope of the given redefinition.

Solution
We add annotations expressing relations between the creation of an object and two actions:
- the creation of a parent object (method target, field owner etc.) - denoted using “+”;
- the execution of a method - denoted using “*”.

We can add such annotations for each:
- method result;
- object field;
- variable.
Why such relations?

- They are “transitive”,
  
  \[ \ldots = \text{expr} \ . \text{method}() \]

- They allow us to reason about the influence of redefinitions.
Why such relations?

- They are “transitive”,
  
  \[
  \ldots = \text{expr .method()}
  \]

- They allow us to reason about the influence of redefinitions.
Why such relations?

- They are “transitive”,
  
  \[ \ldots = \text{expr .method()} \]
  
  \[ * \iff * \]

- They allow us to reason about the influence of redefinitions.
**Why such relations?**

- They are “transitive”,

\[ ... = \text{expr} \ .\text{method}() \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>⇐</td>
<td>*</td>
</tr>
<tr>
<td>*</td>
<td>⇐</td>
<td>*</td>
</tr>
<tr>
<td>+</td>
<td>⇐</td>
<td>+</td>
</tr>
</tbody>
</table>

- They allow us to reason about the influence of redefinitions.

---

**Jarosław Kuśmierk**  
Implicit Genericity
Why such relations?

- They are “transitive”,

\[ \ldots \equiv \text{expr} . \text{method()} \]

\[
\begin{array}{ccc}
* & \equiv & * \\
* & \equiv & + \\
+ & \equiv & + \\
\end{array}
\]

- They allow us to reason about the influence of redefinitions.
When the code is extended with our annotations, the compiler can enrich the type information for an expression in the scope of the redefinition.

```java
public class ServiceLocator
{
    Socket +fSocket;
    public Socket +getSocket() { return fSocket; }
    public Socket *connect() { return new Socket(...); }
    public ServiceLocator() { fSocket = connect(); }
}
...
ServiceLocator sl1 = new ServiceLocator();
with (Socket -> MySSLSocket)
{ ServiceLocator sl2 = new ServiceLocator();
    MySSLSocket a = sl1.connect(); //"*" method on any object
    MySSLSocket b = sl2.getSocket(); //"+" method on a local one
}
```
Summary

So what do we get in the end?
Summary of the Implicit Genericity proposal:

- Allows one to replace within any component some of its subcomponents with compatible ones.
- Is enabled by simple annotations at the point of the calls.
- Has very simple semantics. Avoids influencing:
  - method dispatch,
  - inheritance,
  - the type of an object during runtime.
- It only introduces the lookup of a class during creation.
- But is still very expressive.
- Thanks to that it is efficient.
- Using simple, verifiable, optional annotations, one can reason statically about the influence of the redefinitions.
Summary of the Implicit Genericity proposal:

- Allows one to replace within any component some of its subcomponents with compatible ones.
- Is enabled by simple annotations at the point of the calls.
- Has very simple semantics. Avoids influencing:
  - method dispatch,
  - inheritance,
  - the type of an object during runtime.

It only introduces the lookup of a class during creation.

**But is still very expressive.**

- Thanks to that it is efficient.
  - Using simple, verifiable, optional annotations, one can reason statically about the influence of the redefinitions.
Summary of the Implicit Genericity proposal:

- Allows one to replace within any component some of its subcomponents with compatible ones.
- Is enabled by simple annotations at the point of the calls.
- Has very simple semantics. Avoids influencing:
  - method dispatch,
  - inheritance,
  - the type of an object during runtime.

It only introduces the lookup of a class during creation. **But is still very expressive.**

- Thanks to that it is efficient.
- Using simple, verifiable, optional annotations, one can reason statically about the influence of the redefinitions.
The next step is to include Implicit Genericity in a language with:

- mixins,
  ... in such a way a replacement can refer not only to one specific class but to any class containing a given chain of mixins. This effectively means replacing a family of classes with another family of classes;

- hygienic identifiers approach (Kuśmirek, Bono, TOOLS 2007),
  ... in such a way the replacement of a subsequence of mixins will not cause clashes;

- modular constructors (aka modular initialization protocol) (Bono, Kuśmirek, TOOLS 2007),
  ... to fulfill more easily the requirements of the simulating class relation.
The next step is to include Implicit Genericity in a language with:

- **mixins**, in such a way a replacement can refer not only to one specific class but to any class containing a given chain of mixins. This effectively means replacing a family of classes with another family of classes;

- **hygienic identifiers** approach (Kuśmierek, Bono, TOOLS 2007), in such a way the replacement of a subsequence of mixins will not cause clashes;

- modular constructors (aka modular initialization protocol) (Bono, Kuśmierek, TOOLS 2007), to fulfill more easily the requirements of the simulating class relation.
The next step is to include Implicit Genericity in a language with:

- **mixins**, 
  ... in such a way a replacement can refer not only to one specific class but to any class containing a given chain of mixins. This effectively means replacing a family of classes with another family of classes;

- **hygienic identifiers** approach (Kuśmierek, Bono, TOOLS 2007),
  ... in such a way the replacement of a subsequence of mixins will not cause clashes;

- **modular constructors** (aka **modular initialization protocol**) (Bono, Kuśmierek, TOOLS 2007),
  ... to fulfill more easily the requirements of the simulating class relation.
Thank You!

Any questions?