Join Diesel
Concurrency Primitives for Diesel

Peter-Michael Santos Osera
psosera@cs.washington.edu

University of Washington
Computer Science and Engineering
WASP Research Group

January 26, 2006
Why Design New Concurrency Primitives?

- Traditional locks, monitors, etc., just aren’t cutting it.
  - “Fine” for regulating shared memory access.
  - Awkward for distributed and asynchronous programming.
- Concurrency primitives integrated into the language . . .
  - Enables static analysis for correctness and optimizations.
  - Allows for the run-time system to manage low-level details.
  - Leads to elegant syntax.
Example Declarations in Diesel

```diesel
-- declare a class hierarchy of shapes with multiple inheritance
abstract class Shape;
class Rectangle isa Shape; class Rhombus isa Shape;
class Square isa Rectangle, Rhombus;

-- declare a hierarchy of displays to draw to
abstract class Window;
class WinForm isa Window; class XWindow isa Window;

-- declare a draw function that takes a shape and a display
fun draw(:Shape, :Window):void;

-- declare some implementations for draw with various
-- argument combinations; typechecking ensures no ambiguities!
method draw(r@Rectangle, w@WinForm):void { ... }
method draw(s@Square, w@XWindow):void { ... }
```
Example Declarations in Diesel

-- declare a class hierarchy of shapes with multiple inheritance
abstract class Shape;
class Rectangle isa Shape; class Rhombus isa Shape;
class Square isa Rectangle, Rhombus;

-- declare a hierarchy of displays to draw to
abstract class Window;
class WinForm isa Window; class XWindow isa Window;

-- declare a draw function that takes a shape and a display
fun draw(:Shape, :Window):void;

-- declare some implementations for draw with various
-- argument combinations; typechecking ensures no ambiguities!
method draw(r@Rectangle, w@WinForm):void { ... }
method draw(s@Square, w@XWindow):void { ... }
Diesel in One Slide

Example Declarations in Diesel

-- declare a class hierarchy of shapes with multiple inheritance
abstract class Shape;
class Rectangle isa Shape; class Rhombus isa Shape;
class Square isa Rectangle, Rhombus;

-- declare a hierarchy of displays to draw to
abstract class Window;
class WinForm isa Window; class XWindow isa Window;

-- declare a draw function that takes a shape and a display
fun draw(:Shape, :Window):void;

-- declare some implementations for draw with various argument combinations; typechecking ensures no ambiguities!
method draw(r@Rectangle, w@WinForm):void { ... }
method draw(s@Square, w@XWindow):void { ... }
Example Declarations in Diesel

-- declare a class hierarchy of shapes with multiple inheritance
abstract class Shape;
class Rectangle isa Shape; class Rhombus isa Shape;
class Square isa Rectangle, Rhombus;

-- declare a hierarchy of displays to draw to
abstract class Window;
class WinForm isa Window; class XWindow isa Window;

-- declare a draw function that takes a shape and a display
fun draw(:Shape, :Window):void;

-- declare some implementations for draw with various
-- argument combinations; typechecking ensures no ambiguities!
method draw(r@Rectangle, w@WinForm):void { ... }  
method draw(s@Square, w@XWindow):void { ... }
Why Integrate Concurrency Primitives into Diesel?

- ... why not?
  - Diesel has an extensible object model, multimethods, a constraint-based static typechecking system ... 
  - ... and a traditional thread-lock implementation found in the standard library. Yuck.
- A testbed for exploration.
  - What other approaches to concurrency are there?
  - How expressive are they?
  - How do these primitives interplay with Diesel’s advanced features?
Related Work

- Other asynchronous/distributed computing efforts.
  - $\pi$-calculus [Milner], the Pict programming language.
  - Tuple Spaces, Linda programming language, JavaSpaces Services.
- The Join Calculus [Fournet et al].
  - Implementations: JoCaml [Fournet], Polyphonic C#/C$\omega$ [Benton], Join Java [von Itzstein et al].
Outline

Introduction
- Concurrency and Diesel
- Related Work

Language Design and Rationale
- Method Join Patterns
- Asynchronous Function Declarations
- Parameter Matching
- Join Method Overriding

Dynamic Semantics and Typechecking
- Motivation
- Approaches

Future Work
Method Join Patterns

Example: A Thread-Safe Buffer

```plaintext
fun get():string;
fun put(:string):void;
method get():string and put (s@string):void { s }
```

- `and` joins together multiple different-named headers.
  - The method implements both `get` and `put`.
- Methods execute when all headers have been called.
- Calls are **queued up** until some method is enabled.
  - The earliest calls to `put` and `get` are consumed first.
Example Call Sequence

```
fun get():string;
fun put(:string):void;
method get():string and
    put (s@string):void { s }
```

Queue of Calls

Thread 1
```
put("one");
print_line(get());
```

Thread 2
```
put("two");
```

Thread 3
```
print_line(get());
```

Output
Example Call Sequence

fun get():string;
fun put(:string):void;
method get():string and
    put (s@string):void { s }

Queue of Calls

Thread 1
put("one");
print_line(get());

Thread 2
put("two");

Thread 3
print_line(get());

Output
**Example Call Sequence**

```plaintext
fun get():string;
fun put(:string):void;
method get():string and
  put (s@string):void { s }

Thread 1
put("one"); --(B)
print_line(get());

Thread 2
put("two");

Thread 3
print_line(get());
```

Queue of Calls

```plaintext
put("one")
```

Output

```
One
Two
```
Example Call Sequence

```
fun get():string;
fun put(:string):void;
method get():string and
    put (s@string):void { s }
```

Queue of Calls

```
put("one")
```

Output

```
"One"  "Two"
```

Thread 1

```
put("one"); // (B)
print_line(get());
```

Thread 2

```
put("two");
```

Thread 3

```
print_line(get());
```

Peter-Michael Santos Osera

Join Diesel
Example Call Sequence

```
fun get():string;
fun put(:string):void;
method get():string and
  put (s@string):void { s }
```

Queue of Calls

```
put("one") put("two")
```

Output

Thread 1
```
put("one"); --(B)
p
```

Thread 2
```
put("two"); --(B)
p
```

Thread 3
```
```
Example Call Sequence

fun get():string;
fun put(:string):void;
method get():string and
  put (s@string):void { s }

Queue of Calls
put("one") put("two")

Output

Thread 1
put("one"); --(B)
print_line(get());

Thread 2
put("two"); --(B)

Thread 3
print_line(get());
**Example Call Sequence**

```plaintext
fun get():string;
fun put(:string):void;
method get():string and
    put (s@string):void { s }

Queue of Calls
put("two")

Output
"One"
```

**Thread 1**
put("one"); --(B)
print_line(get());

**Thread 2**
put("two"); --(B)

**Thread 3**
print_line(get());

---

Peter-Michael Santos Osera  
Join Diesel
Example Call Sequence

```plaintext
fun get():string;
fun put(:string):void;
method get():string and
    put (s@string):void { s }
```

Queue of Calls

```
put("two")
```

Output

"One"
Example Call Sequence

```plaintext
fun get():string;
fun put(:string):void;
method get():string and
    put (s@string):void { s }

Queue of Calls
put("one"); --(B)
print_line(get());
put("two"); --(B)
print_line(get());

Output
"One" "Two"
```
Buffer as State Machine

- Essentially a state machine.
  - Calls to put(...) → full state.
  - Calls to get(...) → empty state.
- The design of the state machine makes us thread-safe.
  - Calls to get block until there is an element available.
  - One get call consumes exactly one element of the buffer.
- Common idiom: state of buffer is captured in function calls.
Values are Returned to All Call Sites

Example: A Team-Assignment System

```pseudocode
let var id:int := 0;
fun captain():int;
fun tactician():int;
fun grunt():int;
method captain():int and tactician():int and grunt():int {
    id := id + 1; id }
```

- Return value is returned to all non-void call-sites.
- Return type must be a subtype of each of the non-void return types of the headers.
Asynchronous Function Declarations

Example: An Asynchronous Buffer

```plaintext
fun get():string;
async fun put(:string):void;
method get():string and
    async put (s@string):void { s }
```

- Simple rule: **calls to async functions do not block.**
  - Conceptually, `put(s)` spawns a new thread that waits for a `get()` call.
Async Functions Return Futures

Example: An Asynchronous Team-Assignment System

```plaintext
let var id:int := 0;
fun async captain():int;
fun async tactician():int;
fun async grunt():int;
method captain():int and tactician():int and grunt():int {
    id := id + 1; id }
```

- Problem: `async` calls do not block, but we may need to wait for a return value.
- Solution: `async` calls return a `future[T]`.
  - Object that wraps values that may not be computed yet.
  - Attempts to access the value (via the `value` method) **block** until the value is available.
Two Design Considerations with Async

- Why not make futures transparent? (e.g., f.value() ≡ f).
  - Caller could not distinguish between future and the value.
  - Could not query the state of the future without blocking.
- Why not declare methods `async`?
  - Interface/implementor issue: only the caller cares if it’s call spawns a new thread or returns a future.
  - Consistency issue: dynamic dispatch leads to some calls spawning threads/returning futures and others not. Ow.
Two Design Considerations with Async

- Why not make futures transparent? (e.g., \( f\.value() \equiv f \)).
  - Caller could not distinguish between future and the value.
  - Could not query the state of the future without blocking.

- Why not declare methods `async`?
  - Interface/implementor issue: only the caller cares if it’s call spawns a new thread or returns a future.
  - Consistency issue: dynamic dispatch leads to some calls spawning threads/returning futures and others not. Ow.
Two Design Considerations with Async

- Why not make futures transparent? (e.g., \( f\.value() \equiv f \)).
  - Caller could not distinguish between future and the value.
  - Could not query the state of the future without blocking.

- Why not declare methods \texttt{async}?
  - Interface/implementor issue: only the caller cares if it’s call spawns a new thread or returns a future.
  - Consistency issue: dynamic dispatch leads to some calls spawning threads/returning futures and others not. Ow.
Two Design Considerations with Async

- Why not make futures transparent? (e.g., \( f \).value() \( \equiv \) \( f \)).
  - Caller could not distinguish between future and the value.
  - Could not query the state of the future without blocking.
- Why not declare methods `async`?
  - Interface/implementor issue: only the caller cares if it’s call spawns a new thread or returns a future.
  - Consistency issue: dynamic dispatch leads to some calls spawning threads/returning futures and others not. Ow.
Example: A Global Thread-Safe Buffer

```plaintext
fun get():string;
fun put(:string):void;
method get():string and put (s@string):void { s }
```

- Problem: implementation of a *global* buffer.
  - Any pair of calls to `put` and `get` will enable the method.
  - Need a facility to “instantiate” and refer to multiple buffers.
How Its Done in Cω

Example: A Thread-Safe Buffer in Cω

```java
public class Buffer {
    public async Put(String s);
    public String Put(String s) & Get() {
        return s;
    }
}
```

```c
// bad style in Cω ...
```

- $C_\omega$ is a receiver-based object-oriented language.
- Calls to some object are matched only with calls to that same object.
- i.e., The receivers are compared for identity.
Example: An Instantiable Thread-Safe Buffer

class buffer;
fun get(:buffer):string;
async fun put(:buffer, :string):void;
method get(b@buffer):string and
    async put (b@buffer, s@string):void { s }
let b1 := new buffer;
let b2 := new buffer;

- Formals with the same name → corresponding actuals must be the same object.
- Natural generalization of the single-dispatch case.
  - Can constrain more than just the “receiver” argument.
Method Overriding in Diesel

Example: Overriding Methods

```plaintext
method draw(r@Rectangle, w@XWindow):void { ... } -- method 1
method draw(r@Rhombus, w@XWindow):void { ... } ---- method 2
method draw(r@Square, w@XWindow):void { ... } ----- method 3
... 
```

draw(new Square, new XWindow)

- Method 3 is **more-specific** than methods 1 and 2.
  - Each argument of method 3 is a subtype of the corresponding arguments of method 1 and 2.
- The function call applies to all three methods, but we invoke the **most-specific** method: method 3.
Method Overriding in Join Diesel

Definition

Method $m_1$ overrides method $m_2$ ($m_1 <: m_2$) if:

- Each header of $m_2$ is overridden by a header in $m_1$.
  - $\text{draw}(s@\text{Square}, w@\text{XWindow}) <: \text{draw}(r@\text{Rectangle}, w@\text{XWindow})$
- $m_1$ has at least as many headers as $m_2$.
  - $\text{draw}(s@\text{Square}, w_1@\text{XWindow})$ and $\text{ready}(w_2@\text{XWindow}) <: \text{draw}(s@\text{Square}, w_2@\text{XWindow})$
- $m_1$ has at least all of the constraints of $m_2$.
  - $\text{draw}(s@\text{Square}, w_1@\text{XWindow})$ and $\text{ready}(w_1@\text{XWindow}) <: \text{draw}(s@\text{Square}, w_1@\text{XWindow})$ and $\text{ready}(w_2@\text{XWindow})$

- Corresponds to width and depth subtyping of records.

Peter-Michael Santos Osera

Join Diesel
Method Overriding in Join Diesel

Definition

Method $m_1$ overrides method $m_2$ ($m_1 <: m_2$) if:

- Each header of $m_2$ is overridden by a header in $m_1$.
  - `draw(s@Square, w@XWindow) <: draw(r@Rectangle, w@XWindow)`
- $m_1$ has at least as many headers as $m_2$.
  - `draw(s@Square, w1@XWindow) and ready(w2@XWindow) <: 
    draw(s@Square, w2@XWindow)`
- $m_1$ has at least all of the constraints of $m_2$.
  - `draw(s@Square, w1@XWindow) and ready(w1@XWindow) <: 
    draw(s@Square, w1@XWindow) and ready(w2@XWindow)`
- Corresponds to width and depth subtyping of records.
Method Overriding in Join Diesel

Definition

Method $m_1$ overrides method $m_2$ ($m_1 <: m_2$) if:

- Each header of $m_2$ is overridden by a header in $m_1$.
  - `draw(s@Square, w@XWindow) <: draw(r@Rectangle, w@XWindow)`

- $m_1$ has at least as many headers as $m_2$.
  - `draw(s@Square, w1@XWindow) and ready(w2@XWindow) <: draw(s@Square, w2@XWindow)`

- $m_1$ has at least all of the constraints of $m_2$.
  - `draw(s@Square, w1@XWindow) and ready(w1@XWindow) <: draw(s@Square, w1@XWindow) and ready(w2@XWindow)`

- Corresponds to width and depth subtyping of records.
Method Overriding in Join Diesel

**Definition**

Method $m_1$ overrides method $m_2$ ($m_1 <: m_2$) if:

- Each header of $m_2$ is overridden by a header in $m_1$.
  - draw(s@Square, w@XWindow) <: draw(r@Rectangle, w@XWindow)
- $m_1$ has at least as many headers as $m_2$.
  - draw(s@Square, w1@XWindow) and ready(w2@XWindow) <: draw(s@Square, w2@XWindow)
- $m_1$ has at least all of the constraints of $m_2$.
  - draw(s@Square, w1@XWindow) and ready(w1@XWindow) <: draw(s@Square, w1@XWindow) and ready(w2@XWindow)

- Corresponds to width and depth subtyping of records.
Example of Method Overriding in Join Diesel

Example: Waiting on a Window Before Drawing

```plaintext
...  
fun ready(:Window):void;  
method draw(s@Square, w@XWindow):void { ... }  
method draw(s@Square, w@XWindow):void and
    ready(w@XWindow):void { ... }
let s:Square := new Square;
let w:XWindow := new XWindow;
```

- The sequence of calls `draw(s, w)` applies to one method.
- The sequence of calls `ready(w), draw(s, w)` applies to both methods.
  - The second method is more-specific than the first, so invoke it.
Outline

Introduction
- Concurrency and Diesel
- Related Work

Language Design and Rationale
- Method Join Patterns
- Asynchronous Function Declarations
- Parameter Matching
- Join Method Overriding

Dynamic Semantics and Typechecking
- Motivation
- Approaches

Future Work
Typechecking Motivation

Outcomes From a Function Call

At run-time, a function call can:

- Invoke a method if the call, with other unresolved calls, fulfills that most-specific method.
- Enter the queue of unresolved calls if the call applies to a header in a method otherwise.
- Give a “Message Not Understood” error if the call applies to no header.
- Give an “Ambiguous Message” error if the call, with other unresolved calls, fulfills more than one most-specific method.

- If a program typechecks, the program should never give these errors.
Outcomes From a Function Call

At run-time, a function call can:

- Invoke a method if the call, with other unresolved calls, fulfills that most-specific method.
- Enter the queue of unresolved calls if the call applies to a header in a method otherwise.
- Give a “Message Not Understood” error if the call applies to no header.
- Give an “Ambiguous Message” error if the call, with other unresolved calls, fulfills more than one most-specific method.

- If a program typechecks, the program should never give these errors.
Outcomes From a Function Call

At run-time, a function call can:

- Invoke a method if the call, with other unresolved calls, fulfills that most-specific method.
- Enter the queue of unresolved calls if the call applies to a header in a method otherwise.
- Give a "Message Not Understood" error if the call applies to no header.
- Give an "Ambiguous Message" error if the call, with other unresolved calls, fulfills more than one most-specific method.

- If a program typechecks, the program should never give these errors.
Ambiguous Messages without Join Patterns

Example: Sets of Methods that Cause Ambiguity

...  
fun draw(:Shape, :Window):void;
method draw(@Rhombus, @XWindow):void { ... }
method draw(@Rectangle, @XWindow):void { ... }

- Consider the call draw(new Square, new XWindow):void.
  - Both methods apply to the call, but neither are most-specific.
  - Could resolve by arbitrary choice, textual ordering, etc.
  - Diesel typechecking requires that the resolving method draw(@Square):void is declared.
Ambiguous Messages with Join Patterns

Example: More Sets of Methods that Cause Ambiguity

```plaintext
fun ready(:Window):void; fun available(:Shape):void;
method draw(s@Square, w@XWindow):void and
    ready(w@XWindow):void { ... }
method draw(s@Square, w@XWindow):void and
    available(s@Square):void { ... }
let s:Square := new Square; let w:XWindow := new XWindow;
```

- Both sequences `draw(s, w), ready(w)` and `draw(s, w), available(s)` apply to one method.
- The sequence `ready(w), available(s), draw(s, w)` applies to both methods!
  - Even more ambiguities with multiple inheritance, multimethods, yarg.

Peter-Michael Santos Osera
Join Diesel
Our Approaches

Non-deterministic Choice
If more than one method is enabled, chose one at random.

Only One Method Enabled
Typechecking ensures that any given sequence of unresolved calls will enable at most one most-specific method at runtime.

Final Try
Typechecking ensures that every subsequence of calls enables at most one method. From those methods, choose the one with the earliest sequence of calls to invoke.
Our Approaches

Non-deterministic Choice

If more than one method is enabled, chose one at random.

Too permissive!

Only One Method Enabled

Typechecking ensures that any given sequence of unresolved calls will enable at most one most-specific method at runtime.

Final Try

Typechecking ensures that every subsequence of calls enables at most one method. From those methods, choose the one with the earliest sequence of calls to invoke.
Our Approaches

Non-deterministic Choice

If more than one method is enabled, chose one at random. Too permissive!

Only One Method Enabled

Typechecking ensures that any given sequence of unresolved calls will enable at most one most-specific method at runtime.

Final Try

Typechecking ensures that every subsequence of calls enables at most one method. From those methods, choose the one with the earliest sequence of calls to invoke.
Our Approaches

Non-deterministic Choice
If more than one method is enabled, chose one at random. **Too permissive!**

Only One Method Enabled
Typechecking ensures that any given sequence of unresolved calls will enable at most one most-specific method at runtime. **Too restrictive!**

Final Try
Typechecking ensures that every subsequence of calls enables at most one method. From those methods, choose the one with the earliest sequence of calls to invoke.
Our Approaches

Non-deterministic Choice
If more than one method is enabled, chose one at random.
Too permissive!

Only One Method Enabled
Typechecking ensures that any given sequence of unresolved calls will enable at most one most-specific method at runtime.
Too restrictive!

Final Try
Typechecking ensures that every subsequence of calls enables at most one method. From those methods, choose the one with the earliest sequence of calls to invoke.
Our Approaches

**Non-deterministic Choice**

If more than one method is enabled, chose one at random. **Too permissive!**

**Only One Method Enabled**

Typechecking ensures that any given sequence of unresolved calls will enable at most one most-specific method at runtime. **Too restrictive!**

**Final Try**

Typechecking ensures that every subsequence of calls enables at most one method. From those methods, choose the one with the earliest sequence of calls to invoke. **Just Right!**
Resolving Ambiguities with Join Diesel

Example: More Sets of Methods that Cause Ambiguity

```plaintext
fun ready(:Window):void; fun available(:Shape):void;
method draw(s@Square, w@XWindow):void and
    ready(w@XWindow):void { ... }
method draw(s@Square, w@XWidow):void and
    available(s@Square):void { ... }
let s:Square := new Square; let w:XWindow := new XWindow;
```

- Consider the sequence
  ```plaintext
  ready(w), available(s), draw(s, w).
  ```
  - ready(w), draw(s, w) applies to the first method.
  - available(s), draw(s, w) applies to the second method.
  - ... but ready(w) was called before available(s) so invoke the first method!
Definition (Typechecking Algorithm)

For every pair of declared methods, calculate their set of intersecting methods. If any of these methods are not declared, then the generating pair of methods are *ambiguous*.

- Intuitively
  - Ambiguities occur if the same subsequence of unresolved calls enables two distinct methods.
  - “Intersecting methods” cover these cases by giving the subsequence of calls a more-specific method to enable.
- Process detailed and **proven sound** in thesis work.
Outline

Introduction
  - Concurrency and Diesel
  - Related Work

Language Design and Rationale
  - Method Join Patterns
  - Asynchronous Function Declarations
  - Parameter Matching
  - Join Method Overriding

Dynamic Semantics and Typechecking
  - Motivation
  - Approaches

Future Work
Implementation and Viability

- Take the next step: getting an interpreter up and running.
  - Analyze what optimizations and run-time system support are possible.
  - Streamline the typechecking algorithm.
- Are Joins actually worth it?
  - Try out the system on some “real” programs.
  - Look to Tuple Spaces for ideas . . .
Extensions with Predicate Dispatching

- Expand our limited predicate dispatching.
  - Can condition on object identity between headers of a method.
  - Can condition on arbitrary object state within individual headers (via predicate classes).
  - Cannot condition on arbitrary object state between individual headers.

- Take the next-next step:
  - All expressions are of the form \( b : e \) where \( e \) is an expression and \( b \) is a **boolean guard**.
    - \( b \) can consist of bool expressions and “headers”.
    - A “header” is considered true if an unresolved call applies to it.
  - \( e \) is evaluated only when \( b \) is true.
Acknowledgements and More Information

- Thanks to Craig Chambers, Keunwoo Lee, Sorin Lerner, the WASP research group, and others that have helped me brainstorm countless ideas and designs.
- For more information
  - Email me: psosera@cs.washington.edu.