Join Diesel: Concurrency Primitives for Diesel

Peter-Michael Osera
psosera@cs.washington.edu

December 16, 2005

Abstract

We introduce a syntax and semantics for an adaption of the Join Calculus to the Diesel programming language, entitled Join Diesel. We describe the design decisions and trade-offs made in integrating these concurrency primitives into the Diesel language. We also give a typechecking algorithm for the function calls and method declarations of Join Diesel and then prove its soundness.

Acknowledgements

I would like to extend my deepest thanks to the WASP research group, especially Keunwoo Lee and Sorin Lerner, for their continuous feedback and advice in this endeavour, my advisor, Craig Chambers, for his guidance and infinite patience with me throughout the entire research process, and my friends at the Computer Science and Engineering at the University of Washington whom I bounced off countless ideas and prototypes.

I would also extend my sincerest apologizes to my wife, Victoria, whom has had to sleep alone for many nights while I burnt the midnight oil. I'll sleep one day, I promise!

1 Introduction

With the rise of distributed and asynchronous programming has come an increased need for safe, yet expressive concurrency primitives. Concurrency for the programmer is no longer dominated by problems in the realm of shared-memory access: ensuring that a bank account is not overdrawn by multiple threads or safeguarding a `List` from being modified during iteration. Instead, programmers are coordinating multiple computers to work on a problem across a latency-ridden network or masking I/O in interactive applications. Consequently, old shared-memory models of concurrency are giving way to new models that better represent these distributed and asynchronous views on concurrency [3].

Furthermore, many of these concurrency primitives, threading in particular, have been tucked away in external libraries rather than encoded into the programming language proper. By being in external libraries, the compiler does not have the opportunity to statically analyze code which can give way to a host of optimizations such as smart thread-switching and pooling. Putting concurrency primitives in external libraries can also lead to inelegant syntax, further burdening the programmer. By integrating our concurrency primitives into the language proper, we can thus save the programmer time and effort when dealing with these distributed and asynchronous systems.

1.1 The Diesel Programming Language

Our base language is Diesel [6], the successor to the Cecil programming language [5]. Diesel is an advanced object-oriented programming language featuring an extensible object model, multiple dispatch, and a constraint-based static typechecking system. In Diesel, class declarations do not define a particular module as in C++ and Java (Diesel has its own module system separate from the object system). They instead take the form of simple declarations.

```
class dog;
class boxer isa dog;
class poodle isa dog;
class boxerdoodle isa boxer, poddle;
```

Unlike other object-oriented languages such as Java and C++, method dispatch is not receiver-based in Diesel. Instead, as with CLOS [11], we define generic functions and implementing methods specialized on the parameters of the function.

```
fun play(:dog, :dog):void;
method play(@boxer, @boxer):void { ... }
method play(@poodle, @poodle):void { ... }
method play(@boxerdoodle, @boxer):void { ... }
```

With these multimethods, we dispatch on the runtime types of the arguments whom are specialized in the formal parameter lists of the methods.

However, with all of these advanced features, Diesel implements its concurrency with a traditional thread-lock model found in the standard library. This makes Diesel an excellent candidate to explore new concurrency primitives that support distributed and asynchronous programming integrated with advanced object-oriented features.

2 Related Work

Before tackling our language design proper, we briefly describe several major efforts towards asynchronous and distributed programming all of which have influenced our design directly or indirectly.

2.1 Other Concurrency Efforts

The π-calculus [15] by Robin Milner is an algebra that uses message-passing via channels to model distributed systems. In this minimal calculus, channels are first class and can be themselves sent across channels as messages. The Pict programming language [16] is an ML-style adaption of this algebra to concurrent programming.

Also related to the topic of distributed programming is the Tuple Space and its associated implementation language Linda [4]. A Tuple Space is a shared memory space that is used to coordinate actions between processes. A process can either read or write a tuple of values to and from Tuple Space. The read operation blocks if the requested tuple is not available, leading to a method of using Tuple Space as a common area by which worker processes can coordinate information. This idea has been extended with much success to the Java programming language as the JavaSpaces Services [13].

2.2 The Join Calculus

We base our concurrency extensions on the Join Calculus by Cédric Fournet et al. [10]. This programming model employs synchronous expressions and asynchronous processes coordinated by join patterns to support distributed programming. Fournet and others have also integrated this calculus into the Objective-CAML programming language, the JoCaml system [9], to aid in rapid development of distributed applications.

Others have also integrated the Join Calculus into their programming languages. We have looked greatly to the integration of the Join Calculus by Nick Benton et al. into the C# programming language, first dubbed Polyphonic C#, and then integrated into a larger effort called $C\omega$ [2]. They adapt the process-expression model of the Join Calculus into the object-method model of C#. Also, the Join Calculus has been integrated into the Java programming language by G Stewart von Itzstein et al., called Join Java [12]. In following with Join Java, we name our language extension Join Diesel.

3 Language Design of Join Diesel

We first survey the key features of Join Diesel by example, then move to an informal description of the semantics of these features.

3.1 Example: A Thread-Safe Buffer

To highlight the essential features of Join Diesel, we present a simple implementation of a thread-safe buffer adapted from [2].

```
class buffer;
fun get(:buffer):any;
fun async put(:buffer, :any):void;
method get(b:buffer):any and put(b:buffer, a:any):void { a }
```

The first thing to observe is the use of the **async** reserved word with the declaration of the put function. By declaring put as **async**, any call to put is guaranteed to act asynchronously. That is, the caller of put will not wait for the method body of put to complete; instead, the method body is scheduled for execution in another thread and the caller continues immediately.

Also worth noticing is the method which implements both get and put. In standard Diesel, as with most other languages, a single method implements a single function. However, in Join Diesel, we allow an arbitrary number of functions to be implemented by a single method. To execute the body of the method we thus require that all functions that appear in the header of the method must be called. Since only single calls to functions can be made, function calls are queued up until some subset of them can trigger a method to run. Those fulfilling calls are then removed from the queue — consumed, in other words — and the triggered method body runs. In our particular example, both get and put must be called before the declared method body runs, which returns the argument passed by put to the caller of get. (As we describe later, the return value of a method is actually returned to all callers that participate in the triggering of that method, but functions with a void return type ignore the returned value).

There is also one more constraint on the execution of the method body. We generally require that the formal parameter names in the headers of a method be unique, so we can unambiguously refer to them in the method body. However, between headers, we allow formals to have the same name to indicate that the values corresponding to the formals must be the same. In our buffer example, the buffer objects passed to get and put must be the same.

These two features, *asynchronous function declarations* and *join methods* power our buffer. To fill the buffer, we make repeated calls to put passing the objects we wish to store. These calls return immediately since put is marked **async**. We remove buffered objects by calling get which returns the objects queued up by put. Since the implementing method for get also implements put, calls to get block until there is a matching put call. In this manner, our buffer is thread safe: calls to get will never get elements that are not in the buffer since the join method requires calls to get and put to be paired up.

An unusual features of this code is that elements of our buffer are not stored in a particular object, but instead in the built up queue of calls to put. This use of the unresolved calls to store state is a common idiom in the Join Calculus style of programming concurrent and distributed systems.

3.2 Asynchronous Function Declarations

A function call can be thought of as a synchronous event with respect to the caller in that the caller must wait for the callee to return before continuing. In our proposed syntax (extended from the Diesel grammar found in [6] [1]), we optionally allow function calls to be asynchronous events by declaring the corresponding function **async**.

```
fun_decl ::= [type_cxt] [privacy] "fun" ["async"] fun_name
             [formal_params] "(" [formal_ps] ")" [type_decl_p}
             [type_cons] {pragma} fun_body
```

Note that we allow a function, rather than an implementing method, to be annotated with the **async** reserved word. This is because **async** modifies only the interface of a function, the **fun** declaration, rather than its implementation, the **method** declaration. The implementing method, the callee, does not care if it executes in a separate thread or if its return value is packaged into a `future`; these are, instead, concerns of the caller of the function.

Calls to asynchronous functions do not block waiting for the method body to complete. Because of this, the value returned to the call site will usually not be immediately available. To remedy this, we instead return a `future` [7] whose contents are filled in by the method body upon its completion. Thus all **async** functions that have return type `T` implicitly have return type `future[T]`.

```
-- process the provided GUI event
fun async process_event(:event):event_result;
```

For example, we typically wish to process GUI events such as mouse clicks without unnecessarily blocking the client. This declaration of an event-processing function ensures that all implementing methods take place in a separate thread from the caller. Also, if we wished to return some result to the caller of `process_event`, a `future[event_result]` is provided.

3.2.1 Futures

A `future` is an additional class to the Diesel standard library. It acts as a wrapper around a value that may not be computed yet, blocking processes that try to access that value until it becomes available. To access the contents of a `future`, we call the `value` function:

```
let future_result: future[event_result] := process_event(e);
let result: event_result := future.value();
```

If the callee has not yet completed, then the call to `value` blocks until the return value is made available by the callee. In this manner, the function call behaves asynchronously yet retains the capacity to receive a value from the callee when necessary.

Alternatively, we could have made the `future` object transparent with respect to the value it contains:

```
let result: event_result := process_event(e);
```

[1]In this EBNF notation, brackets surround optional strings, braces surround strings that may be repeated zero or more times, and quotes surround literal tokens.

In this style, we treat the `future[T]` as if it were of type `T`. However, the semantics remain the same. If we try to use `result` and the callee has not yet completed, then we block until the value is available. Otherwise, access to `result` behaves like a normal `event_result` object.

This has the benefit of making **async** and non-**async** return values interchangeable. However, the caller then cannot distinguish between values that are available and those that are not available and thus cause blocking. Furthermore, there are times when the programmer may wish to have explicit access to the futures. For example, we may wish to coordinate access among values returned from multiple asynchronous function calls which would not be possible without having explicit handles on the `future` objects. Because of this, we elected to expose the `future` interface to programmers rather than hide it.

3.3 Method Join Patterns

In Diesel, a single method implements a single function. However, in Join Diesel, a method may implement multiple functions by specifying their signatures — henceforth called headers to avoid confusion with the Diesel `signature` declaration — joined together by the **and** reserved word.

```
method_decl ::= [type_cxt] [privacy] "method" ["signature"]
                method_headers {pragma} method_body
method_headers ::= method_header { "and" method_header }
method_header ::= method_name [formal_params] "(" [meth_formals] ")"
                [type_decl_p] [type_cons]
```

The method body executes only when this join pattern has been fulfilled, that is, when all of these functions have been called with the appropriate arguments. For example, the following method implements a printing buffer that prints its contents only when someone requests for them:

```
public method echo(s@string):void and next():void {
  s.print_line();
}
```

Both `echo` and `next` must be called before the argument to `echo` is printed (which must be a `string` or a subtype thereof).

Calls to functions that do not complete a join method are queued up. For example, making the calls

```
echo("hello");
echo("beautiful");
echo("world");
```

will queue up three `echo` calls, each with different string arguments. Three separate calls to `next` will be required to print these strings. Furthermore, these strings will be printed in the order received:

```
next();
```

```
> "Hello"
next();
> "beautiful"
next();
> "world"
```

3.3.1 Parameter Matching

Our previous example of a buffered printer is, in fact, a global, singleton buffered printer. This is because any call to `echo` can be matched up to any call to `next`.

However, we may want to have multiple buffered printers, each with their own "state" of queued-up `echo` calls. We do so by creating a tie-in class `buffered_printer` and constraining the method to only be invoked when `echo` and `next` pass the same `buffered_printer` object:

```
class buffered_printer;
public method echo(b@ buffered_printer, s@string):void
   and next(b@ buffered_printer):void {
  s.print_line();
}
```

By using the same formal name for the `buffered_printer` argument, we say that the `buffered_printer` values passed in by `echo` and `print` must be the same. In general, we use the same formal name for several parameters in a join method's headers to indicate that those values corresponding to those parameters must also be the same object.

Note that the sort of parameter matching illustrated in the above example is implicit in the receiver-based language Cω:

```
public class BufferedPrinter {
  public async Next();
  public void Echo(string s) & Next() {
    System.Console.WriteLine(s);
  }
}
```

We may instantiate multiple `BufferedPrinter`s to call the `Echo` and `Next` methods. However, calls to one `BufferedPrinter` are not matched up with calls to another. Under this system, the "zeroth" argument to the method calls, the receiver, is checked for identity. Thus we can think of our version of parameter matching as a generalization, in the spirit of multimethods, of the single-dispatch case in which we allow arbitrary matching of values across the headers of a method.

3.3.2 Return Types

In a join method, the returned value is sent to all call sites that participated in the enabling of that particular method. We may wish to break up a large computation into several independent parts, but return a success indicator to the contributors indicating if their contributions were compatible.

```
fun contribute_state(:state):bool;
fun contribute_action(:action):bool;
method contribute_state(s@state):bool
      and contribute_action(a@action):bool
   { ... }
```

The type of the method body's result must be a subtype of each of the non-`void` return types of the headers. This way, the return value is applicable to all relevant call sites. As another example, we may wish to extend the factory pattern to create objects that are the compositions of a pair of objects.

```
class set;
class iterable;
class iterable_set isa set, iterable;
fun compose_set(): set;
fun compose_iterable(): iterable;
method compose_set(s:set):set
     and compose_iterable():iterable {
  new iterable_set(d, c);
}
```

The resulting `iterable_set` is treated as a `set` at the `compose_set` call site and as an `iterable` at the `compose_iterable` call site.

Many functions in a join method do not care about the return value or do not wish to reveal it to the call site. To ensure that a function call does not receive a return value, we say that calls to functions declared with a `void` return type do not receive a value, even if they are part of a join method which returns a value. For example, in our original buffer:

```
method get(b:buffer):any and put(b:buffer, a:any):void { a }
```

The caller of `put` ignores the value returned to `get`, as it should.

4 Formalization and Typechecking

4.1 Introduction

An important part of Diesel's typechecking system is its ability to ensure that no message sends result in an "ambiguous message" error by way of calculating the "intersection" of methods. For example, consider the hierarchy of shape classes and corresponding methods to draw them:

```
abstract class shape;
class rectangle isa shape;
class rhombus isa shape;
class square isa rectangle, rhombus;
```

```
fun draw(:shape):void;
method draw(@rhombus):void { ... }
method draw(@rectangle):void { ... }
```

Calling the `draw` function with a `square` argument is ambiguous because such a message send fulfills both implementations of `draw`, neither of which is most-specific. Typechecking in Diesel ensures that the resolving method

```
method draw(@square):void { ... }
```

is present by calculating the most-general child shared by `rhombus` and `rectangle` in the inheritance hierarchy; this third `draw` method is the intersection of the two other `draw` methods.

With join patterns, it is much easier to arrive at a "ambiguous message" error:

```
method get_color(@color):void and draw(@rhombus):void { ... }
method draw_next():void and draw(@rhombus):void { ... }
```

If we call `draw` first, then whatever one of `get_color` or `draw_next` is called first will determine which method body is invoked. However, if both `get_color` and `draw_next` are called before `draw`, then the call to `draw` will fulfill both methods. Because of these factors, designing the typechecking semantics for Join Diesel has proven to be a challenge.

4.1.1 Prior Approaches

In other implementations of the Join Calculus, such as Cω, if a method call enables multiple chords, then one is chosen nondeterministically. Since C# (and Cω, by extension) does not support multiple inheritance, the nondeterministic choice can only happen if multiple chords are being enabled. However, our previous example illustrates that, in Diesel, even regular methods can cause ambiguity problems. We would like to preserve Diesel's strong typechecking semantics under this degenerate case, so resolving ambiguities simply by nondeterministic choice is not an option.

Our original semantics made the guarantee that any series of calls would enable at most one most-specific method. To do this, we extended the notion of formal and header intersection with method intersection that covered the cases where more than one most-specific chorded method would be enabled. We initially thought that the set of methods the programmer would need to provide (the method intersection) would be small, but a resulting bug in our typechecking proof showed us that the set was wrong in many cases.

For example, if the join methods

```
method clone(@rectangle):void and draw(@rectangle):void { ... }
method clone(@rhombus):void and draw(@rhombus):void { ... }
```

were declared, then these old semantics would have required that the method

```
method clone(@square):void and draw(@square):void { ... }
```

be declared, but the call sequence `clone(@rectangle), clone(@rhombus), draw(@square)` would not be covered by this method. The only way to resolve this sequence is to require a method that implemented both `clone(@rectangle)` and `clone(@square)`, but this approach would allow multiple header names to appear in a join method which would introduce even more ambiguity issues.

4.1.2 Current Approach

In light of this discovery, we chose to change our dynamic semantics to agree with what our typechecking rules were checking. The key insight is that we really want each subsequence of calls (of our sequence of unresolved calls) to enable at most one method. In the above example, the subsequence `clone(@rectangle), draw(@square)` would enable the first method, and the subsequence `clone(@rhombus), draw(@square)` would enable the second. From these enabled methods, we choose the method who utilizes the earliest sequence of calls.

This approach handles the degenerate case outlined above as the only call `draw(@square)` would enable both `draw` methods. Our dynamic semantics would consider this an ambiguous message send. Thus we would require that the method intersection, `draw(@square)`, be present to cover the ambiguity.

Likewise, this approach also handles the set of join methods outlined above. The only subsequence of calls that would cause an "ambiguous message" error under these new semantics is `clone(@square):void, \lstinline`draw(@square):void! (and its reverse) is covered by the calculated method intersection.

To this end, we formalize the dynamic semantics for function calls, omitting constructs such as resends and even return types in favor of analyzing the behavior of function calls in Join Diesel. We also formalize a set of typechecking rules for both function calls and method declarations that utilize the idea of method intersections to ensure that no ambiguities arise when calling functions. Finally, we prove our typechecking rules sound by showing that they ensure that no function call produces a runtime error if typechecking for all function calls and method declarations pass.

4.2 Mathematical Preliminaries

We first formalize our language constructs in terms of mathematical set notation.

Formal. Say that a *formal* a of a method header is simply a type t representing the type of its static specializer.

To refer to the type of a formal, let the function $\textsc{type}((n, t)) = t$.

Method Header. If a method header has name f and formals $a_1, \dots, a_n$, then let the *method header* h for this declaration be a pair consisting of the name and a tuple of formals, $(f, (a_1, \dots, a_n))$, where the name of each formal is unique. That is, for all a_i and a_j in $a_1, \dots, a_n$ where $i \neq j$, $\textsc{name}(a_i) \neq \textsc{name}(a_j)$.

To refer to the name of a method header, let the function $\textsc{name}((f, (a_1, \dots, a_n))) = f$.

Actual. Say that an *actual* b of a dynamic function call is a pair (v, t) representing the value passed in v and its dynamic type t. Since we only care about the value insofar as to compare it to other values for identity we take $v \in \mathcal{Z}^+$ and do equality checks on v in the usual manner for integers.

To refer to the identity of an actual, let the function $\textsc{id}((v, t)) = v$. To refer to the type of an actual, let the function $\textsc{type}((v, t)) = t$.

Static Function Call. If a static function call has name f and argument types $t_1, \ldots, t_n$, then let the *static function call c* be a pair consisting of the name and argument type tuple, $(f, (t_1, \ldots, t_n))$.

To refer to the name of a static function call, let the function $\textsc{name}((f, (t_1, \ldots, t_n))) = f$.

Dynamic Function Call. If a dynamic function call has name f and actuals $b_1, \ldots, b_n$, then let the *dynamic function call d* be a pair consisting of the name and an actual tuple, $(f, (b_1, \ldots, b_n))$.

To refer to the name of a static function call, let the function $\textsc{name}((f, (a_1, \ldots, a_n))) = f$.

Method. If a method declaration consists of a set of method headers $\{h_1, \ldots, h_k\}$ and set of constraints on pairs of argument positions (locations) l_i in those headers $((f_1, l_1), \ldots, (f_n, f_n))$, then let a *method m* be defined as a pair (H, C) consisting of possibly-empty set of constraints C and the non-empty set of headers H. That is, for all h_i and h_j in $\{h_1, \ldots, h_k\}$ where $i \neq j$, $\textsc{name}(h_i) \neq \textsc{name}(h_j)$.

Methods. Let M be the set of all declared methods.

Call-Header Fulfillment. Say that a static call $c = (f, (a_1, \ldots, a_m))$ *fulfills* a header $h = (f, (a'_1, \ldots, a'_n))$ if $n = m$, and $\textsc{type}(a_i) <: \textsc{type}(a'_i)^{\forall i \in 1, \ldots n}$.

Likewise, say that a dynamic call $d = (f, (b_1, \ldots, b_m))$ *fulfills* a header $h = (f, (a_1, \ldots, a_n))$ if $n = m$, and $\textsc{type}(b_i) <: \textsc{type}(a_i)^{\forall i \in 1, \ldots n}$.

Calls-Method Fulfillment. Say that a sequence of dynamic calls $d_1, \ldots, d_n$ *fulfills* a method (H, C) if $d_1, \ldots, d_n$ is the smallest sequence, where for all $h_i \in H$, d_i fulfills h_i and for each $((f_1, l_1), (f_2, l_2)) \in C$, $\textsc{id}(a_{l_1}) = \textsc{id}(a'_{l_2})$ for $d_i = (f_1, (a_1, \ldots, a_{l_1}, \ldots, a_n))$ and $d_j = (f_2, (a'_1, \ldots, a_{l_2}, \ldots, a'_m))$.

Appending Unresolved Calls. Say that we *append* dynamic call d to a sequence of unresolved calls U, written $U :: c$, so that, if $U = d_1, \ldots, d_i$, then $U :: d = d_1, \ldots, d_i, d$.

Earlier Calls. Say that in a sequence of unresolved calls $U = d_1, \ldots, d_n$ a call $d_i \in U$ is *earlier* than a call $d_j \in U$ if $i < j$. That is, higher indices indicate later function calls.

Subsequences. Let $D' = d'_1, \ldots, d'_n$ be a subsequence of a sequence of dynamic calls $D = d_1, \ldots, d_n$, written $D' \sqsubseteq D$, if $D = \ldots, d'_1, \ldots, d'_i, \ldots, d'_n, \ldots$. That is, $D' \sqsubseteq D$ if the sequence D' occurs within D, not necessarily contiguously.

Earliest Subsequence. Say that a sequence of unresolved calls $U = d_1, \ldots, d_n$ is *earliest with respect to U* if $d_1, \ldots, d_n \sqsubseteq U$ and, for all $d_i = (f, (b_1, \ldots, b_m))$, there does not exist a $d'_i \in U$ where $d'_i = (f, (b'_1, \ldots, b'_m))$ is earlier than d_i and $\textsc{type}(b_j)' <: \textsc{type}(b_j)$ for $j \in 1, \ldots, m$.

Lexicographic Ordering of Subsequences of Calls. Let U be a sequence of unresolved calls. Then $d_1, \ldots, d_n \sqsubseteq U \prec d'_1, \ldots, d'_m \sqsubseteq U$, the first subsequence is lexicographically less than the second, if there exists an i where d_i is earlier than d'_i and for all $j < i$, $d_i = d'_j$.

4.3 Overriding Rules

The following inference rules define method and header overriding.

$$\frac{\textsc{type}(a_1) <: \textsc{type}(a'_1) \cdots \textsc{type}(a'_n) <: \textsc{type}(a'_n)}{(f, (a_1, \ldots, a_n)) <: (f, (a'_1, \ldots, a'_n))} \quad \text{(Header Overriding)}$$

$$\frac{h_1 <: h'_1 \cdots h_n <: h'_n \qquad C' \subseteq C}{(\{h_1, \ldots, h_{n+k}\}, C) <: (\{h'_1, \ldots, h'_n\}, C')} \quad \text{(Method Overriding)}$$

Note that, like records, larger methods (those with more headers), subsumes smaller methods with exactly a subset of the headers of the larger method.

4.4 Dynamic Semantics

Let $\textsc{am}(d)$ be the *set of all applicable methods* to a dynamic call d where

$$\textsc{am}(d) = \{m \in M \mid \exists h \in m.\ d \text{ fulfills } h\}.$$

Let $\textsc{fsc}(m, U, d)$ be the unique *fulfilling subsequence of calls* for a method m, sequence of unresolved calls U, and dynamic call d where

$$\textsc{fsc}(m, U, d) = \begin{cases} d_i, \ldots, d_j \sqsubseteq U :: d & d_i, \ldots, d_j \text{ fulfills } m \text{ and } d_i, \ldots, d_j \text{ is} \\ & \text{earliest with respect to } U :: d\} \\ \emptyset & \text{if no such } d_i, \ldots, d_j \text{ exists.} \end{cases}$$

Then define $\textsc{fm}(U, d)$ to be the *set of methods fulfilled by dynamic call d and sequence of unresolved calls U* as

$$\textsc{fm}(U, d) = \{m \in \textsc{am}(d) \mid \textsc{fsc}(m, U, d) \neq \emptyset\}$$

We can extend this definition to $\textsc{msm}(U, d)$, the *set of fulfilled, most-specific methods* for a dynamic call d and sequence of unresolved calls U to be

$$\textsc{msm}(U, d) = \{m \in \textsc{fm}(U, d) \mid \nexists m' \in \textsc{fm}(U, d).\ m' \neq m \text{ and } m' <: m\}.$$

With these definitions, we can thus define the function VALID-CALL as taking the dynamic call to be resolved d and the sequence of unresolved calls U as

$$\textsc{resolve-call}(U,d) = \begin{cases} \text{“message not understood”} & \text{if } \textsc{am}(d) = \emptyset, \\ \text{“ambiguous message”} & \text{if } \exists m_1, m_2 \in \textsc{msm}(U,d). \\ & \textsc{fsc}(m_1,U,d) = \textsc{fsc}(m_2,U,d), \\ (m, U') & m \in \textsc{msm}(U,d), U' = U - \textsc{fsc}(m,U,d) \text{ where,} \\ & \nexists m' \in \textsc{msm}(U,d).\ \textsc{fsc}(m',U,d) \prec \textsc{fsc}(m,U,d), \\ (\emptyset, U :: d) & \text{if } |\textsc{msm}(U,d)| = 0. \end{cases}$$

“Message not understood” implies that the call d did not contribute to the enabling of any declared method. “Ambiguous message” implies that there exists a subsequence of dynamic calls of U that enabled more than one method. Otherwise, each subsequence of dynamic calls enables at most one method, so we choose the method whose enabling subsequence of calls occurs earliest, by our defined lexicographical ordering for subsequences of calls, relative to U. Otherwise, no methods are enabled (but d still fulfills some header in some method), so we add d to the sequence of unresolved calls.

4.5 Static Typechecking

4.5.1 Function Call Typechecking

The static typechecking of function calls follows naturally from the dynamic rules. Define the *set of all applicable methods* to a static call c where

$$\textsc{am}(c) = \{m \in M \mid \exists h \in m.\ c \text{ fulfills } h\}.$$

Let the function $\textsc{wt-call}$ (well-typed call) take the static call c to be checked as

$$\textsc{wt-call}(c) = \begin{cases} \text{“message not understood”} & \text{if } \textsc{am}(c) = \emptyset, \\ \text{“valid”} & \text{otherwise.} \end{cases}$$

A function call typechecks as long as it could fulfill some header in some method.

4.5.2 Method Declaration Typechecking

We define intersections successively of arguments, then headers, and then methods to define when a method declaration is type-safe.

Define the *intersection of two formals*, written $a_1 \sqcap a_2$, to be

$$a_1 \sqcap a_2 = \{a \in S \mid \nexists a' \in S.\ a' \neq a \text{ and } \textsc{type}(a) <: \textsc{type}(a')\}$$

where $S = \{(t) \mid t <: \textsc{type}(a_1) \text{ and } t <: \textsc{type}(a_2)\}$, the set of children common to both a_1 and a_2.

Next, define the *intersection of two headers* $h_1 = (f, (a_1, \dots, a_n))$ and $h_2 = (f', (a'_1, \dots, a'_m))$, written $h_1 \sqcap h_2$, as the set of headers where

$$h_1 \sqcap h_2 = \begin{cases} \emptyset & \text{if } f \neq f' \text{ or } m \neq n, \\ \{h \mid h = (f, A) \text{ where } A \in a_1 \sqcap a'_1 \times \cdots \times a_n \sqcap a'_m\} & \text{otherwise.} \end{cases}$$

Then define the intersection of declared methods $m_1 = (H, C')$ and $m_2 = (H', C')$, written $m_1 \sqcap m_2$, as

$$m_1 \sqcap m_2 = \begin{cases} (\emptyset, \emptyset) & \text{if } \exists h \in m_1.\ \nexists h' \in m_1.\ \textsc{name}(h) = \textsc{name}(h') \text{ or} \\ & \exists h' \in m_2.\ \nexists h \in m_1.\ \textsc{name}(h) = \textsc{name}(h'), \\ (\{h_1, \dots, h_n\}, C \cup C') \mid h_i \in t_i \text{ for each } t_i \in I\} & \text{otherwise,} \end{cases}$$

where I is the collection of all header intersections between the headers of m_1 and m_2 of the same name,

$$I = \{h_1 \sqcap h_2 \mid h_1 \in H, h_2 \in H', \text{and } \textsc{name}(h_1) = \textsc{name}(h_2)\}.$$

Finally, we define the function WT-METHOD (well-typed method) taking the method m to be typechecked as

$$\textsc{wt-method}(m) = \begin{cases} \text{"valid"} & \text{if } \forall m' \in M.\ m \sqcap m' \subseteq M, \\ \text{"ambiguous declaration"} & \text{otherwise.} \end{cases}$$

"Ambiguous declaration" implies that there exists some sequence of unresolved calls U and a single additional dynamic function call c that, at runtime, will cause both the method m and some other unrelated method m' to be enabled without enabling a more-specific overriding method.

4.6 Typechecking Soundness

To prove that our formulation is sound — methods and function calls declared typesafe do not cause runtime errors — we show that the function call typechecking ensures that we do not receive a "message not understood" error at runtime and method declaration typechecking ensures that we do not receive an "ambiguous message" error at runtime. Namely, we wish to prove the two theorems

Theorem 1. *(Well-Typed Calls do not Produce Errors). Let* $c = (f, (a_1, \dots, a_n))$ *be a static call. Then* $\textsc{wt-call}(c) =$ *"valid"* $\Rightarrow \forall (b_1, \dots, b_n)$ *where* $\textsc{type}(b_i) <: \textsc{type}(a_i)^{\forall i \in 1, \dots, n}$. $\forall U.\ \textsc{resolve-call}((U, d))$ $\neq$ *"message not understood" where* $d = (f, (b_i, \dots, b_n))$.

Theorem 2. *(Well-Typed Methods do not Produce Errors).* $(\forall m \in M.\ \textsc{wt-method}(m) =$ *"valid"*$) \Rightarrow$ $(\forall U.\ \forall c.\ \textsc{resolve-call}(U, d) \neq$ *"ambiguous message"*$)$.

The proof of theorem 1 is straightforward.

Proof of Theorem 1. Given that $\textsc{wt-call}(c)$ = "valid", then $\textsc{am}(c) \neq \emptyset$. This implies that for some declared method m, there exists some header h where c fulfills h. If $c = (f, (a_1, \ldots, a_n))$ and $h = (f, (a'_1, \ldots, a'_n))$, then by definition of fulfillment, it follows that $\textsc{type}(a_i) <: \textsc{type}(a'_i)^{\forall i \in 1,\ldots,n}$.

Now consider a dynamic call $d = (f, (b_1, \ldots, b_n))$ and an arbitrary unresolved sequence of calls U. Assuming that the arguments of the dynamic call are well-typed, then it follows that $\textsc{type}(b_i) <: \textsc{type}(a_i)^{\forall i \in 1,\ldots,n}$ since the arguments of a dynamic call are subtypes of its static counterpart. By subtype transitivity, since $\textsc{type}(b_i) <: \textsc{type}(a_i)$ and $\textsc{type}(a_i) <: \textsc{type}(a'_i)^{\forall i \in 1,\ldots,n}$ then it follows that $\textsc{type}(b_i) <: \textsc{type}(a'_i)^{\forall i \in 1,\ldots,n}$. By the definition of fulfillment, this means that d fulfills h. And thus the method m containing h is in $\textsc{am}(d)$ as well. Since $\textsc{am}(d)$ is, therefore, non-empty, then $\textsc{resolve-call}(U, d) \neq$ "message not understood". □

The proof of theorem 2 is more complicated and first requires the following lemmas.

Lemma 1. *(Intersection Headers Override their Parents). If $h \in h_1 \sqcap h_2$ then $h <: h_1$ and $h <: h_2$.*

Proof of Lemma 1. Consider some $h \in h_1 \sqcap h_2$. By the definition of header intersection all of h, h_1, and h_2 must have the same number of arguments and the same name (otherwise such an h could not exist since $h_1 \sqcap h_2$ would be empty) and thus $h = (f, (a_1, \ldots, a_n))$, $h_1 = (f, (a'_1, \ldots, a'_n))$, and $h_2 = (f, (a''_1, \ldots, a''_n))$. Also, by the definition of header intersection, each $a_i \in a'_i \sqcap a''_i$. By the definition of formal argument intersection, $a_i <: a'_i$ and $a_i <: a''_i$ for all $i \in 1, \ldots n$. This implies, by our rule of header overriding, that $(f, (a_1, \ldots, a_n)) <: (f, (a'_1, \ldots, a'_n))$ and $(f, (a_1, \ldots, a_n)) <: (f, (a''_1, \ldots, a''_n))$. □

Lemma 2. *(Intersection Methods Override their Parents). If $m \in m_1 \sqcap m_2$ then $m <: m_1$ and $m <: m_2$.*

Proof of Lemma 2. Consider some $m \in m_1 \sqcap m_2$ where $m = (H, C)$, $m_1 = (H_1, C_1)$ and $m_2 = (H_2, C_2)$. By the definition of method intersection, there must exist some $h^* \in H$ where $h^* = h$ if there is no header $h' \in H_1 \cup H_2$ where $h' \neq h$ and $\textsc{name}(h) = \textsc{name}(h')$, or $h^* \in h \sqcap h'$ if such a header h^* exists. In the first case, clearly $h^* <: h$ since $h^* = h$. In the second case, by lemma 1, since $h^* \in h \sqcap h'$, then $h^* <: h$ and $h^* <: h'$.

Therefore, for all $h_1 \in H_1$, there exists some $h'_1 \in H$ where $h'_1 <: h_1$. Also for all $h_2 \in H_2$, there exists some $h'_2 \in H$ where $h'_2 <: h_2$. Furthermore, by the definition of method intersection $C = C_1 \cup C_2$ so clearly $C \subseteq C_1$ and $C \subseteq C_2$. By our rule of method overriding, this implies that $m <: m_1$ and $m <: m_2$. □

Lemma 3. *(Intersecting Methods are Enabled). For all dynamic calls d and sequences of unresolved calls U, if $\exists m, m' \in \textsc{msm}(U, d)$. $\textsc{fsc}(m, U, d) = \textsc{fsc}(m', U, d) = D$, then $\exists m^* \in m \sqcap m'$. $m^* \in \textsc{fm}(U, d)$ and $\textsc{fsc}(m^*, U, d) = D$.*

Proof of Lemma 3. Since $\textsc{fsc}(m, U, d) = \textsc{fsc}(m', U, d)$ and both are the smallest such sequences that fulfill m and m', if $m = (H, C)$, and $m' = (H', C')$, then by the definition of the fulfilling sequence of calls, each $d \in D$ fulfills $h \in H$ and $h' \in H'$ where $\textsc{name}(d) = \textsc{name}(h) = \textsc{name}(h')$, and each header of H and h'' is fulfilled in this manner. Let $d = (f, (b_1, \ldots, b_n))$, $h = (f, (a_1, \ldots, a_n))$, and

$h' = (f, (a'_1, \ldots, a'_n))$. By the definition of fulfillment, it follows that for each $d \in D$, $h \in H$, and $h' \in H'$ related in this manner, $\textsc{type}(b_i) <: \textsc{type}(a_i)$ and $\textsc{type}(b_i) <: \textsc{type}(a'_i)$ for all $i \in 1, \ldots, n$.

With this in mind, let $m^* \in m \sqcap m'$ where $m^* = (H^*, C^*)$. Since it holds that between each related h and h' that $\textsc{name}(h) = \textsc{name}(h')$, then by the definition of method intersection, there must exist $h^* \in m^*$ where $h^* \in h \sqcap h'$. If $h^* = (f, (a^*_1, \ldots, a^*_n))$, then by the definition of header intersection, $\textsc{type}(a^*_i) <: \textsc{type}(a_i)$ and $\textsc{type}(a^*_i) <: \textsc{type}(a'_i)$ for all $i \in 1, \ldots, n$.

However, each such a^*_i is, by definition of formal intersection, *the most-general type* where $\textsc{type}(a^*_i) <: \textsc{type}(a_i)$ and $\textsc{type}(a^*_i) <: \textsc{type}(a'_i)$. Since $a_i \sqcap a'_i$ is the set of all such most-general types, then $\exists a^*_i \in a_i \sqcap a'_i.\ \textsc{type}(b_i) <: \textsc{type}(a^*_i)$. Furthermore, by the definition of header intersection, we create the cartesian product of such a^*_is so $\exists h^* \in h \sqcap h'.\ \textsc{type}(b_i) <: \textsc{type}(a^*_i)$ for all $i \in 1, \ldots, n$. Finally, by the definition of method intersection, we create all combinations of these headers by name, so $\exists m^* \in m_1 \sqcap m_2$ where for each $d_i \in D$, d_i fulfills $h^* \in H^*$. Since our method intersection never introduces any new headers, it also follows that each $h^* \in H^*$ is fulfilled by a $d_i \in D$.

Also by the definition of method intersection, $C^* = C \cup C'$. Since $D = \textsc{fsc}(m, U, d) = \textsc{fsc}(m', U, d)$, then D satisfies the constraints of C and C'. Furthermore, $C \cup C'$ does not introduce any new constraints, so D must satisfy C^* as well.

Therefore, for this m^* it holds that $\textsc{fsc}(m^*, U, d) = D = \textsc{fsc}(m, U, d) = \textsc{fsc}(m', U, d)$ and therefore, $m^* \in \textsc{fm}(U, d)$. □

With these lemmas established, we can now prove theorem 2.

Proof of Theorem 2. If, for all declared methods $m \in M$, it holds that $\textsc{wt-method}(m) =$ "valid", then for all $m, m' \in M$, $m \sqcap m' \subseteq M$.

Assume for the sake of contradiction that some arbitrary dynamic call d and sequence of unresolved calls U caused an "ambiguous message" error. That is, $\textsc{resolve-call}(U, d) =$ "ambiguous message". Then, by definition, $\exists m_1, m_2 \in \textsc{msm}(U, d)$ where $\textsc{fsc}(m_1, U, d) = \textsc{fsc}(m_2, U, d)$.

By lemma 3, there exists an m^* where $m^* \in m_1 \sqcap m_2$ and $m^* \in \textsc{fm}(U, d)$. Also, by our typechecking assumption $m^* \in M$. Therefore, by the definition of $\textsc{msm}$, m_1 and m_2 can not be in $\textsc{msm}(U, d)$ since $m^* \in \textsc{fm}(U, d)$, $m^*, \neq m_1$, and by lemma 1 $m^* \neq m_2$, and $m^* <: m_1$ and $m^* <: m_2$. This contradicts the assumption that m_1 and m_2 were both most-specific, fulfilled methods, so under these conditions, m_1 and m_2 cannot exist and thus $\textsc{resolve-call}(U, d)$ cannot be "ambiguous message".

□

Since both theorems are proven, our typechecking semantics ensure that we do not encounter any errors at run time.

5 Further Work

In designing Join Diesel, we have discovered many new avenues of research to pursue. In closing, we describe these areas in more detail.

5.1 Implementation

We are currently working on an implementation of Join Diesel by extending the Diesel interpreter found in the Diesel standard library. The issue of performance, particularly between the dispatching on join methods versus regular methods, is important to justify the use of joins in practical programs. Benton et al. have run benchmarks on their *Cω* join system, which uses hashing and bit vectors to efficiently determine if a join pattern is enabled on a given call. Their tests show that their Join Patterns perform comparatively with normal methods and traditional monitors built into *Cω* [2].

One important area of exploration that has not been studied in depth is what static and runtime optimizations can be made with Join Patterns. Managing the thread pool based on the formation of Join Patterns can provide substantial gains in performance without the need to hand-manage the thread pool. Work can be done to develop algorithms to efficiently manage the thread pool.

In addition to these areas, we must also worry about the performance of typechecking. Our typechecking algorithm naively checks all methods against all other methods. There is no doubt room for optimization here, particularly in reducing the search space over which we typecheck.

5.2 Viability Assessment

The next question after an implementation of Join Diesel is complete is whether or not its extensions work in practice. The Join Calculus is relatively young, so many of its implementations are still research vehicles. Benton has demonstrated that the Join Calculus can elegantly tackle complex theoretical problems such as the Santa Clause problem [1]. However, there has been little in the way of the Join Calculus in industrial-style programs. One interesting avenue of research would be taking a Join Calculus implementation such as Join Diesel or *Cω* and applying it to a "real" distributed program.

However, another avenue of analysis can be found with a more established form of distributed programming in Tuple Spaces. There is an obvious correlation between Tuple Space and the Join Calculus. In the Join Calculus, function calls act like a Tuple Space tuple: tuples are ejected into space asynchronously and read back in, blocking the reading process if the tuple is not yet available. Function calls can also be made asynchronously and also block if the corresponding join method is not yet enabled. With this in mind, we can look to Tuple Spaces for possible programming idioms to assist in our viability assessment.

5.3 Extensions with Predicate Dispatching

One problem that Benton et al. cites for future research is enabling a kind of ML style of pattern matching to create conditional join patterns [2]. Diesel offers a form of predicate dispatching in the form of predicate objects which dynamically change types depending on the state of the object. This works well to condition on objects within individual headers in a join method. However, we still cannot condition on objects between headers in a join method. A form of predicate dispatching [8, 14] would be necessary to condition on any arbitrary combination of arguments within the headers of a join method.

One can generalize this even further in the following sense. We can associate with every expression e a boolean guard b, and require that e executes only when b is `true`. The boolean guard can take the form of any boolean expression which includes an arbitrary number of function headers. And like the Join Calculus, we can consider the function header portion of a boolean guard to be `true` only when all the headers specified by the guard have been called.

A number of interesting factors arise when considering such a system. One such example is that if we guard every expression with a boolean, what does it mean for the guard to be true? Does the expression execute once when the boolean condition changes from `false` to `true`? Or does the expression repeatedly evaluate until the guard becomes `false`? A formal calculus will go a long way towards clarifying these issues.

Afterwards

In reflection of my time as a researcher, the one thing I wish had more of was time. More time to sit down. More time to analyze. More time to program. More time to prove. More time to think. Juggling an intensive undergraduate curriculum and research has proven to be a grueling endeavour with many corners cut, days lost, and sleepless nights. However, I would not have had this experience any other way; the many things I have grown to appreciate along the way are priceless.

For one, I have a better understanding of the sheer amount of time it takes to conduct good research. While I knew prior that the process itself is long, I did not know exactly where those many hours went. Now I do: false starts, staring at a whiteboard for hours, trying to wedge that last detail into the proof before giving up and rewriting it from scratch. And these hours are not lost or wasted: it was a good thing that a nearly completed yet-broken proof showed us the flaw in our semantics and had us thinking for two more weeks. Because of this "wasted time", our results are all the better.

Another thing that struck me is the necessity of being able to discover and identify the possible future research topics that arise from your own work. On one hand, identifying these possible research areas is a responsibility of the academic researcher to other academic researchers. But on the other hand, identifying these areas gives you focus and understanding in your own work. For me, I found that the similarities between Tuple Spaces and the Join Calculus were critical in my comprehension of this design, which in turn helped me communicate my ideas to others. Furthermore, seeing all the possibilities that can arise from your work can be very motivating. Whenever I discovered a new area for further research, this discovery further affirmed the importance of my work and got me re-engaged into the project.

A related issue I have discovered is the necessity of being able to communicate your results to others. This has me particularly excited about the research process as I love teaching and education, especially about my own research and interests. However, this has also meant that I have needed to learn how to articulate my own thoughts and ideas, something which I am still actively working on.

Finally, the last thing I have grown to appreciate is the feeling of ownership involved in research. There is a certain joy involved with doing something that you know no one else is pursuing and then becoming an authority in that area. Consequently, this whole experience has reaffirmed my desire to become a Computer Science researcher. This combination of knowledge-producing, knowledge-owning and teaching is what I want to do with my life; my research experiences confirm it.

References

[1] Nick Benton. Jingle bells: Solving the santa claus problem in, March 24 2003.

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst*, 26(5):769–804, 2004.

[3] Luca Cardelli. Transitions in programming models. Talk given at the New University of Lisbon, November 13, 2003.

[4] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, November 1989.

[5] Craig Chambers. The Cecil language specification and rationale: Version 2.1. Available from `http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html`, March 1997.

[6] Craig Chambers. The Diesel language specification and rationale: Version 0.1. February 2005.

[7] Arunodaya Chatterjee. Futures: a mechanism for concurrency among objects. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 562–567, 1989.

[8] Michael Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP*, pages 186–211, 1998.

[9] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. The jocaml language beta release: Documentation and user's manual. Available from `http://pauillac.inria.fr/jocaml/htmlman/index.html`, Jan 2001.

[10] Cédric Fournet and Luc Maranget. The join-calculus language release 1.05: Documentation and user's manual. Available from `http://pauillac.inria.fr/join/manual/index.html`, 1997.

[11] Richard P. Gabriel, John L. White, and Daniel G. Bobrow. CLOS: Integrating object-orietned and functional programming. *CACM: Communications of the ACM*, 34, 1991.

[12] David Kearney and Stewart Itzstein. Applications of join java, Janurary 2002.

[13] Sun Microsystems. Javaspaces service specification, October 2000.

[14] Todd D. Millstein. Practical predicate dispatch. In *OOPSLA*, pages 345–364, 2004.

[15] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[16] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. 1997.