Implementing Joins using Extensible Pattern Matching

Philipp Haller, EPFL

joint work with

Tom Van Cutsem, Vrije Universiteit Brussel

Introduction

- Concurrency is indispensable: multi-cores, asynchronous, distributed applications
- Two interesting pieces of the puzzle:
  - Joins: high-level, declarative specification of synchronization constraints (origin: join calculus)
  - Extensible pattern matching constructs of modern languages (e.g., Scala, F#)
- **Idea**: integrate join synchronization using extensible pattern matching

Example

Simple buffer in Cω:

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

Asynchronous methods

- Never block
- Return unit/**void**

Join operator **&**

- Definition of join pattern
- Exactly one sync method (Get)
- Multiple async methods (Put)
- Result returned to caller of sync method

Example using Scala Joins

```
class Buffer extends Joins {
  val Put = new AsyncEvent[Int]
  val Get = new SyncEvent[Int]
  join { case Get() & Put(x) => Get reply x }
}
```

enable join patterns

Event types:

- Asynchronous/synchronous
- Types of argument(s)/result

- Multiple sync events per join pattern allowed
- Reply to sync events explicitly

Reader/Writer Lock

```
class ReaderWriterLock extends Joins {
  val Exclusive, ReleaseExclusive = new NullarySyncEvent
  val Shared, ReleaseShared = new NullarySyncEvent
  private val Sharing = new AsyncEvent[Int]

  join {
    case Exclusive() & Sharing(0) => Exclusive.reply()
    case ReleaseExclusive() =>
      Sharing(0); ReleaseExclusive.reply()

    case Shared() & Sharing(n) =>
      Sharing(n+1); Shared.reply()

    case ReleaseShared() & Sharing(n) =>
      Sharing(n-1); ReleaseShared.reply()
  }
  Sharing(0)
}
```

Nested pattern

Invoking (private) events inside bodies/constructor

Reader/Writer Lock: C# Joins

```
public class ReaderWriter {
  public Synchronous.Channel Exclusive, ReleaseExclusive;
  public Synchronous.Channel Shared, ReleaseShared;
  private Asynchronous.Channel Idle;
  private Asynchronous.Channel<int> Sharing;
  public ReaderWriter() {
    Join j = Join.Create(); ... // Boilerplate omitted

    j.When(Exclusive).And(Idle).Do(delegate {});
    j.When(ReleaseExclusive).Do(delegate{ Idle(); });

    j.When(Shared).And(Idle).Do(delegate{ Sharing(1); });
    j.When(Shared).And(Sharing).Do(delegate(int n) {
      Sharing(n+1); });

    j.When(ReleaseShared).And(Sharing).Do(delegate(int n) {
      if (n==1) Idle(); else Sharing(n-1); });

    Idle(); } }
```

Expressiveness of Scala Joins

1. Nested patterns
   - ✓ Avoid if-else inside join bodies
   - ✓ Less redundancy, fewer messages
2. Guards
   - ✓ No need for internal events to maintain state
3. Dynamic joins
   - ✓ Beyond compiled schemes, such as Cω
   - ✓ Sequence matching
     - Deconstruct sequence inside pattern

Pattern Matching in Scala

```
{
  case pat_1 => body_1
  ...
  case pat_n => body_n
}
```

is compiled to (anonymous) class that extends

```
class PartialFunction[A,B] extends Function[A,B] {
  def isDefinedAt(x: A): Boolean }

class Function[A,B] {
  def apply(x: A): B }
```

Join Patterns as Partial Functions

```
join { case Get() & Put(x) => Get reply x }
```

(is compiled to)

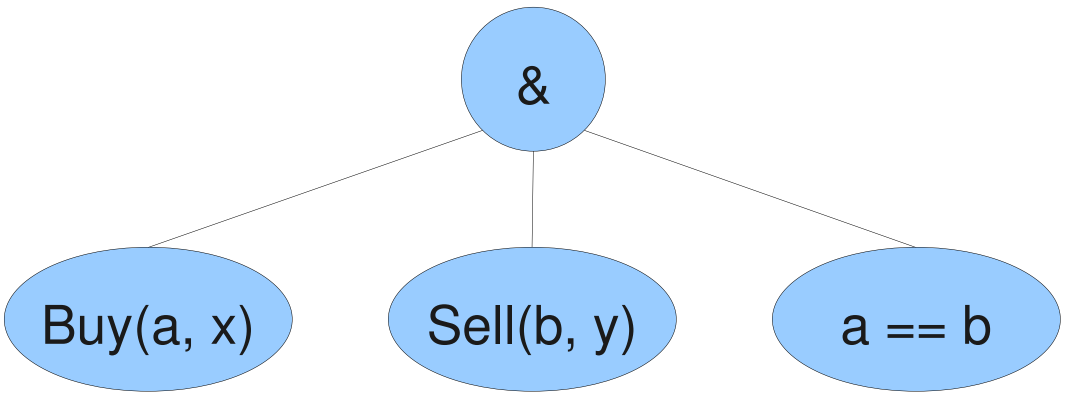
```
val pats = new PartialFunction[?, Unit] {
  def isDefinedAt(arg: ?): Boolean = ...
  def apply(arg: ?): Unit = ...
}
join(pats)
```

- isDefinedAt: check whether a join pattern matches
- apply: execute body of first matching join pattern

Matching Join Patterns: Example

```
join { case Buy(a, x) & Sell(a, y) => ... }
```

```
patFun :=
```



Invocations:

```
(B, U)    (A, V)
(A, W)    (C, V)
```

Matching:

(1, 1) **FAIL**
(1, 2) **FAIL**
(2, 1) **MATCH**

Matching Problem

- Problem: outcome of matching depends on history of event invocations
- Ex.: `join { case Get() & Put(x) => Get reply x }`
  - When invoking Get, join pattern matches *iff* Put has been invoked previously
- Solution
  - Buffer event invocations
  - Consult these buffers during matching
    - **For this, use extensible pattern matching!**

Extensible Pattern Matching

- Extractors (Scala), Active Patterns (F#)
- Implicitly apply type conversions
- Ex.:

```
(x: Int) match {
  case Twice(y) => println("x = 2*"+y)
  case _        => println("x uneven")
}
```

(is compiled to)

```
Twice.unapply(x) match {
  case Some(y) => println("x = 2*"+y)
  case None    => println("x uneven")
}
```

Matching Join Patterns

```
join { case &(Get(), Put(x)) => Get reply x }
```

(is compiled to)

```
new PartialFunction[?, Unit] {
  def isDefinedAt(arg: ?) =
    &.unapply(arg) match {
      case Some((Get(), Put(x))) => true
      case _ => false }
```

Matching Join Patterns (2)

```
join { case &(Get(), Put(x)) => Get reply x }
```

(is compiled to)

```
new PartialFunction[?, Unit] {
  def isDefinedAt(arg: ?) =
    &.unapply(arg) match {
      case Some((u, v)) =>
        Get.unapply(u) match {
          case true =>
            Put.unapply(v) match {
              case Some(x) => true
```

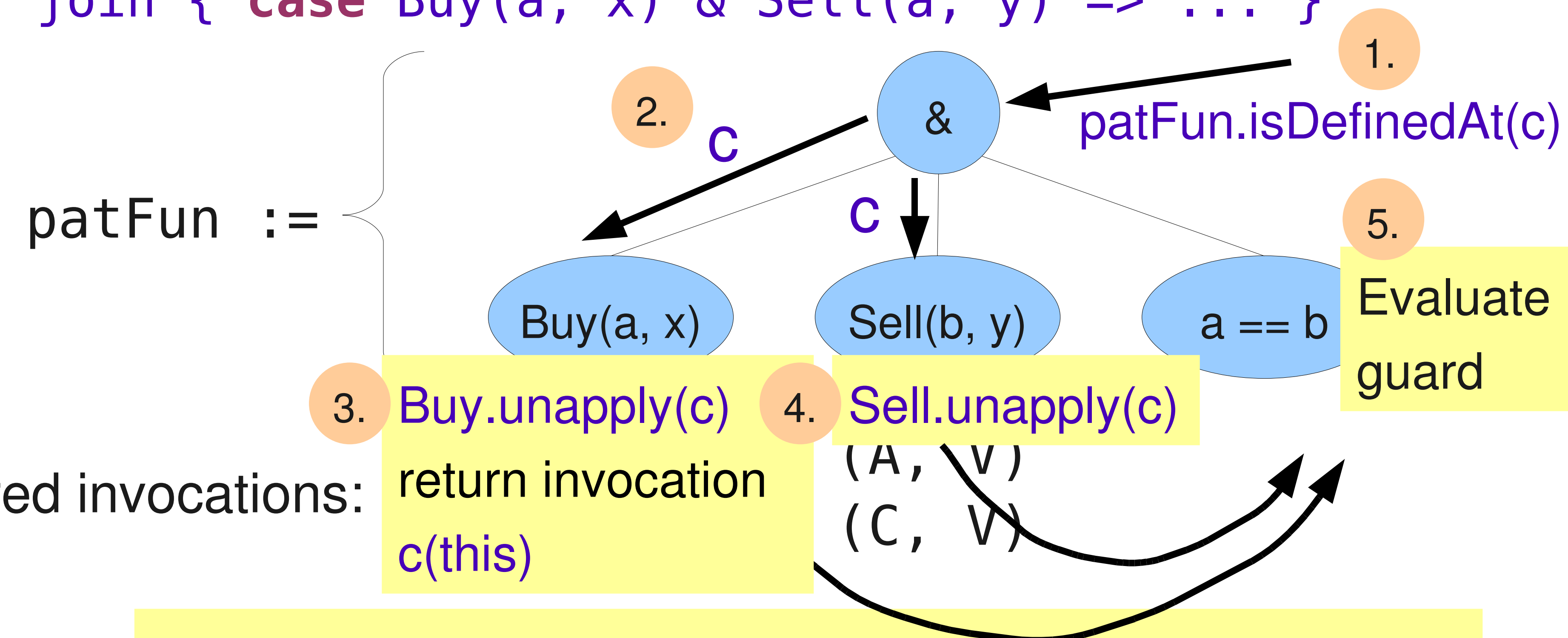
Permits information to be passed down from & to the events!

-> Control matching of events

Matching Join Patterns: Example

```
join { case Buy(a, x) & Sell(a, y) => ... }
```

Buffered invocations:



Matching:

For each combination c:

- call patFun.isDefinedAt(c)
- until match found or all combinations have been tried

Joins for Actors

- Enables join patterns **alongside normal message patterns**:

```
receive {
  case Put(x) & Get() => Get reply x
  case Some(other) => ...
}
```

- More general: **generalization of existing libraries** that use pattern matching
  - By sticking to simple PartialFunction interface
  - Compilation to locks would require adding compiler support for actors!

Ongoing and Future Work

- Generalization of programming model
  - e.g., mobility of events/join messages
- Search strategies (e.g., first match)
- Optimizations
- Experimental evaluation
- Compiler plug-in
  - More consistency checks
  - Offline optimization

Scala Joins

- Exploits extensible pattern matching
- High expressiveness
  - Nested patterns and guards
  - Dynamic joins
- Library-based design
  - Enables generalization of existing libraries
- Strong consistency checks
- Download: http://lamp.epfl.ch/~phaller/joins/