

Actors with Multi-Headed Message Receive Patterns

Martin Sulzmann¹, Edmund S. L. Lam² and Peter Van Weert^{3*}

¹ Programming, Logics and Semantics Group, IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S Denmark
`martin.sulzmann@gmail.com`

² School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
`lamsoonl@comp.nus.edu.sg`

³ Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
`Peter.VanWeert@cs.kuleuven.be`

Abstract. The actor model provides high-level concurrency abstractions to coordinate simultaneous computations by message passing. Languages implementing the actor model such as Erlang commonly only support single-headed pattern matching over received messages. We propose and design an extension of Erlang style actors with receive clauses containing multi-headed message patterns. Patterns may be non-linear and constrained by guards. We provide a number of examples to show the usefulness of the extension. We also explore the design space for multi-headed message matching semantics, for example first-match and rule priority-match semantics. The various semantics are inspired by the multi-set constraint matching semantics found in Constraint Handling Rules. This provides us with a formal model to study actors with multi-headed message receive patterns. The system can be implemented efficiently and we have built a prototype as a library-extension to Haskell.

1 Introduction

We all know the free lunch is over. We must write concurrent programs to take advantage of the next generation of multi-core architectures. But writing correct concurrent programs using the traditional model of threads and locks is inherently difficult and error-prone. Message-based concurrency provides the programmer the ability to exchange messages without relying on low-level locking and blocking mechanisms. A particular popular form of message-based concurrency is actor style concurrency [1] as implemented by the Erlang language [2].

In Erlang, an actor comes with an asynchronous message queue also known as mailbox. Erlang actors communicate by sending and receiving messages. Sending is a non-blocking (asynchronous) operation. Each sent message is placed in the actors mailbox and immediately returns to the sender. Messages are processed via receive clauses which resemble pattern matching clauses found in

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen)

functional/logic languages. Receive clauses are tried in sequential order. The receive operation is blocking. If none of the receive clauses applies we suspend until a matching message is delivered.

Receive clauses in Erlang are restricted to a *single-headed* message pattern. That is, each receive pattern matches at most one message, possibly constrained by a guard. There are situations where we wish to match against multiple messages. Via *multi-headed* message patterns we can give a direct encoding of such problems. But such patterns are not commonly supported in Erlang style languages. The programmer herself must therefore either explicitly keep track of the set of partial matches or resort to nested received clauses. This leads to clumsy and error-prone code as we will see later in Section 2.

In this paper, we make the following contributions:

- We propose and design an extension of Erlang style actors with receive clauses containing multi-headed message patterns. Patterns may be non-linear (i.e. have multiple occurrences of the same pattern variable) and be constrained by guards. There are several possibly ways how to define multi-head message matching, for example either first-match or rule priority-match. We explore both alternatives in detail (Section 4).
- We have implemented a library-based prototype in Haskell (Section 5).

We draw our inspiration from prior work in the concurrent constraint logic programming community. Specifically, we adopt the various multi-set constraint matching semantics found in Constraint Handling Rules [6]. Section 3 provides the necessary background information. We discuss related work in Section 6. Section 7 concludes and discusses some possible future work.

We assume that the reader has some basic familiarity with Erlang and functional languages such as Haskell. We will write example programs in Haskell syntax [15] extended with actors. The Haskell extension uses some minor syntactic sugar compared to our library-based extension described in Section 5. Throughout the paper whenever we refer to actors we mean Erlang style actors.

2 Motivating Example

We motivate multi-headed message receive patterns via a classic concurrency challenge, the Santa Claus problem [18].

Santa Claus First Match (Variant) Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa chooses the *first matching* group of either elves or reindeer waiting.

Single-Headed Solution We give a solution in Haskell extended with Erlang style actors. We omit some unimportant tasks such as “deliver toys” and “show study” and assume that initially ten elves and nine deer are sent to the Santa actor which appear in random order in Santa’s mailbox.

```
data SantaMsg = Deer Int | Elf Int

santa sanActor DeerAcc ElvesAcc =
  receive sanActor of
    Deer x -> if length (Deer x:DeerAcc) == 9
              then ‘‘Deliver toys etc’’
              else santa sanActor (Deer x:DeerAcc) ElvesAcc
    Elf x ->  if length (Elf x:ElvesAcc) == 3
              then ‘‘Show study etc’’
              else santa sanActor DeerAcc (Elf x:ElvesAcc)
```

The critical task for Santa is to check for nine reindeer and three elves. Santa will pick the group whichever arrives first. In Erlang, receive patterns are single-headed. Therefore, the Santa actor accumulates the set of deer and elves received so far. We are slightly more explicit compared to Erlang in that the `receive` primitive takes the actor as the first argument and the receive clauses as second argument (similar to case statements). The actual behavior is like in Erlang. Receive clauses are tried from top to bottom, for one message at a time. If a message does not match any of the clauses we try the next message. In our case, we first match the current message against the deer pattern. If the match fails, we check for an elf. If this match fails as well, we move on to the next message and the process repeats itself. If none of the messages match we block and wait for new messages to arrive. This case does not apply here because there are only deer or elf messages. The receive clauses are exhaustive. The point to note is that Erlang actors apply a *first-match semantics* which selects the first clause (starting from the top) that matches the messages as they come in (the actors mailbox).

Multi-Headed Solution Via multi-headed message patterns we can omit the accumulation of partial matches entirely. Here is a solution using our proposed multi-head extension:

```
santa2 sanActor =
  receive sanActor of
    Deer x1, Deer x2, Deer x3, Deer x4, Deer x5,
    Deer x6, Deer x7, Deer x8, Deer x9 -> ‘‘Deliver toys etc’’
    Elf x1, Elf x2, Elf x3 -> ‘‘Show study etc’’
```

We explain the semantics for such an extension in terms of multi-set constraint matching semantics studied in the context of Constraint Handling Rules (CHR) [6]. CHR is a concurrent committed-choice constraint logic programming language to transform (rewrite) multi-sets of constraints into simpler ones. Constraints correspond to messages, and the left-hand side of a CHR rule corresponds to the pattern of a receive clause. Concretely, we adopt the refined CHR

semantics [3] which finds a match for the left-hand side of a CHR rule by processing constraints in sequential order and testing CHR rules from top to bottom. For single-headed CHR rules, this is essentially the first-match actor semantics. The refined CHR semantics provides for a formal basis to extend the first-match actor semantics with multi-headed message receive patterns involving guards.

Guarded Multi-Heads Suppose not every group of three elves is compatible. For example, either only odd or even numbered elves are willing to work together. Via guards we can easily impose this condition:

```
santa3 sanActor =
  receive sanActor of
    Deer x1, Deer x2, Deer x3, Deer x4, Deer x5,
    Deer x6, Deer x7, Deer x8, Deer x9 -> ‘‘Deliver toys etc’’
    Elf x1, Elf x2, Elf x3 when allOddorEven [x1,x2,x3] -> ‘‘Show study etc’’
  where
    allOddorEven xs = (and (map xs odd)) || (and (map xs even))
```

Multi-Heads are Unordered The message order in patterns is irrelevant. For example, the multi-headed receive clauses `Deer x, Elf y -> body` and `Elf y, Deer x -> body` are equivalent. The rationale behind this design choice is as follows. We treat the multiple messages in a pattern as an un-ordered multi-set because, for the user, the order in which messages arrive is not observable. A specific order among messages can be imposed using nested receive clauses. For instance, the following program text gives priority to the elf:

```
receive someActor of
  Elf y -> receive someActor of
    Deer x ->
```

Nested receive clauses in combination with `otherwise` statements (introduced shortly) can be essential to express priorities as we demonstrate next.

Santa Claus Priority-Match (Original) In the original specification of the Santa Claus problem [18], instead of choosing the *first* matching group of three elves or nine reindeer waiting, Santa needs to give *priority* to the reindeer if there are *matching* groups of both elves and reindeer waiting. Under a first-match semantics, our previous solutions `santa` and `santa2` do not obey the priority given to a group of deer. Suppose for example that at the moment the `receive` statement is executed, three elves and nine deer are waiting in Santa’s mailbox, with the elves appearing first in the mailbox. The first-match semantics of receive patterns then selects the three elves (hence execute “show study”), even though nine deer are waiting. The priority given to the deer has to be encoded explicitly:

```
santa4 sanActor =
  receive sanActor of
    Deer x1, Deer x2, Deer x3, Deer x4, Deer x5,
    Deer x6, Deer x7, Deer x8, Deer x9 -> ‘‘Deliver toys etc’’
    otherwise -> receive sanActor of
      Elf x1, Elf x2, Elf x3 -> ‘‘Show study etc’’
      otherwise -> santa4 sanActor
```

First, we check if there are nine deer (waiting) in Santa’s mailbox. Otherwise, we call a nested receive statement to check for three elves. Otherwise, the process repeats itself. The `otherwise` statement corresponds to ‘`after 0`’ in Erlang. This branch applies if none of the other branches could find a match. The outer `otherwise` for instance applies if there are fewer than nine deer in Santa’s mailbox. Enforcing priorities manually via `otherwise` and nested receive statements leads to clumsy code. For concurrency problems with priorities a different semantics is warranted in which receive clauses are executed in (textual) order. Incidentally, in the CHR literature a semantics has been recently suggested [12] in which rewrite rules can be executed in textual order. If we adopt such a semantics to the actor setting, solution `santa2` immediately solves the Santa Claus Priority-Match problem.

Summary Thanks to multi-headed message receive patterns the programmer is relieved from the tedious and non-trivial task of building the set of partial matches herself. In combination with guards this leads to more concise and maintainable code. Erlang style actors follow the first-match semantics. The refined CHR semantics [3] is a conservative extension of this semantics to the setting of multi-set matching involving guards. Certain concurrency problems, however, are more naturally solved using a rule priority-match semantics which has also been explored in the CHR context.

In the up-coming section, we provide background information on the first-match and rule priority-match semantics. In Section 4, we formalize an extension of actors with multi-headed message patterns which can be constrained by guards. The extension is parametric in terms of the underlying message match semantics for receive clauses. In case we adopt a first-match CHR style semantics for message patterns, we obtain a conservative extension of Erlang style actors.

3 Constraint Handling Rules Matching

We review the essentials of the multi-set constraint matching semantics of Constraint Handling Rules (CHR). The actual CHR framework is much richer than presented here. CHR also supports constraint propagation and built-in constraints such as unification constraints. We ignore these additional features.

Figure 1 introduces some basic syntactic categories. Constraints are terms built via constructors K . Constraints carry a distinct number to distinguish multiple appearances of a constraint c . Rule patterns consist of a head and guard component. A CHR rule also consists of a rule body which we ignore here. We are only interested in the multi-set constraint match semantics of CHR and not in CHR execution. In the actor context, a rule pattern corresponds to a receive pattern and a rule body corresponds to the body of a receive clause. The guard must evaluate to a Boolean value. Constraints in rule heads have distinct, increasing occurrences with respect to their textual order in a program. For example, the rule heads derived from the `santa3` function in the previous section are

$$Deer\ x_1 : 1 \wedge \dots \wedge Deer\ x_9 : 9$$

Constraints

	K	Constructor name
c	$::= K\ c\dots c$	Constraint
	$ x$	Constraint variable
cn	$::= c\#n$	Numbered constraint
co	$::= c : j$	Occurrence constraint
cno	$::= c\#n : j$	Active constraint

Substitution

$$\theta ::= [c_1/x_1, \dots, c_n/x_n]$$

Rule patterns

H	$::= co \mid H \wedge H$	Head
G	$::= e$	Guard
RP	$::= H \text{ when } G$	Rule pattern
	$ H$	
\mathcal{RP}	$::= \{RP_1, \dots, RP_n\}$	Set of rule patterns

Executables

$$M ::= N \mid [cno|N]$$

$$N ::= [] \mid [cn|N]$$

Store

$$St ::= [] \mid [cn|St]$$

Matching States

$\langle M, St \rangle$	Intermediate
$\langle M, St, \theta, RP \rangle$	Successful
$\langle [], St \rangle$	Failure

Fig. 1. Constraint Handling Rules Essential Syntax

and

$$Elf\ x_1 : 10 \wedge Elf\ x_2 : 11 \wedge Elf\ x_3 : 12$$

Thus, we can perform a systematic search for a match.

The idea is that all (numbered) constraints are initially stored in a list M . We use Prolog syntax to denote a list $[x|xs]$ with first element x and tail xs . The symbol $[]$ denotes the empty list and $++$ denotes list concatenation. In the CHR context, M is referred to as the execution stack. Here, it is more appropriate to view M as a list corresponding to the actors mailbox. Constraints in M are executed in sequential order to find a match with a rule pattern. We execute constraints by activating them with the initial occurrence number 1. We will increase the occurrence until a match is found. Otherwise, we deactivate the current active constraint by putting it into the store and start with a new active constraint. In case of the first-match semantics, this strategy guarantees that constraints are processed in sequential order and rules are tried from top to bottom. Below are the formal details.

Matching reduction: $\langle M, St \rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St \rangle$ and $\langle M, St \rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St, \theta, RP \rangle$

$$\begin{array}{l}
\text{(Activate)} \quad \langle [c\#n|M], St \rangle \longrightarrow_{First-\mathcal{RP}} \langle [c\#n : 1|M], St \rangle \\
\\
\text{(Match)} \quad \frac{
\begin{array}{l}
H_1, c' : j, H_2 \text{ when } G \in \mathcal{RP} \\
\theta(G) \text{ evaluates to } True \quad St_1 ++ St_2 ++ St' =_{set} St \\
\theta(c') = c \quad \theta(H_1) = St_1 \quad \theta(H_2) = St_2 \quad \text{for some } \theta
\end{array}
}{
\langle [c\#n : j, M], St \rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St', \theta, H_1 \wedge c' : j \wedge H_2 \text{ when } G \rangle
} \\
\\
\text{(Continue)} \quad \frac{j < maxOccur(\mathcal{RP})}{\langle [c\#n : j|M], St \rangle \longrightarrow_{First-\mathcal{RP}} \langle [c\#n : j + 1|M], St \rangle} \\
\\
\text{(Deactivate)} \quad \frac{j \geq maxOccur(\mathcal{RP})}{\langle [c\#n : j|M], St \rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St ++ [c\#n] \rangle} \\
\\
\text{(Step1)} \quad \frac{\langle M, St \rangle \longrightarrow_{First-\mathcal{RP}} \langle M', St' \rangle}{\langle M, St \rangle \longrightarrow_{First-\mathcal{RP}}^* \langle M', St' \rangle} \\
\\
\text{(Step2)} \quad \frac{\langle M, St \rangle \longrightarrow_{First-\mathcal{RP}} \langle M', St', \theta, RP \rangle}{\langle M, St \rangle \longrightarrow_{First-\mathcal{RP}}^* \langle M', St', \theta, RP \rangle} \\
\\
\text{(Trans)} \quad \frac{\langle M_1, St_1 \rangle \longrightarrow_{First-\mathcal{RP}}^* \langle M_2, St_2 \rangle \quad \langle M_2, St_2 \rangle \longrightarrow_{First-\mathcal{RP}}^* \langle M_3, St_3 \rangle}{\langle M_1, St_1 \rangle \longrightarrow_{First-\mathcal{RP}}^* \langle M_3, St_3 \rangle}
\end{array}$$

Fig. 2. CHR Multi-Set First Match Semantics

3.1 First-Match Semantics

Our presentation largely follows the CHR description [3], which we adapt to our specialized setting. Figure 2 describes the CHR multi-set first-match semantics as a transition system $\longrightarrow_{First-\mathcal{RP}}^*$ among states $\langle M, St \rangle$ where M represents the constraints to be executed and St holds the already processed constraints. The set \mathcal{RP} holds the rule patterns. Initially, we start in the state $\langle M, [] \rangle$. The goal is to reach a successful state $\langle M', St, \theta, RP \rangle$ where RP is the (first) rule pattern matched by a (sequentially processed) sequence of constraints in M , θ is the matching substitution, St holds the already processed constraints that did not contribute to the match, and M' are the remaining constraints. State $\langle [], St \rangle$ indicates failure: none of the constraints in the initial M trigger a rule pattern.

The search for a match is performed by activating the leading constraint in M by assigning it the occurrence number 1. See rule (Activate). Rule (Match) checks whether the active constraint matches a constraint in the head of a rule pattern at the respective position. We consult the store to find constraints St_1 and St_2

Matching reduction: $\langle M, St \rangle \longrightarrow_{Priority-\mathcal{RP}}^* \langle M, St \rangle$ and $\langle M, St \rangle \longrightarrow_{Priority-\mathcal{RP}}^* \langle M, St, \theta, RP \rangle$

$$\begin{array}{c}
\mathcal{RP} = \{RP_1, \dots, RP_n\} \\
\text{(Succ)} \quad \frac{\forall 1 \leq j < i \langle M, St \rangle \longrightarrow_{First-\{RP_j\}}^* \langle [], St'_j \rangle \quad \langle M, St \rangle \longrightarrow_{First-\{RP_i\}}^* \langle M', St', \theta, RP_i \rangle}{\langle M, St \rangle \longrightarrow_{Priority-\mathcal{RP}}^* \langle M', St', \theta, RP_i \rangle} \\
\mathcal{RP} = \{RP_1, \dots, RP_n\} \\
\text{(Fail)} \quad \frac{\forall 1 \leq j \leq n \langle M, St \rangle \longrightarrow_{First-\{RP_j\}}^* \langle [], St'_j \rangle}{\langle M, St \rangle \longrightarrow_{Priority-\mathcal{RP}}^* \langle [], St_n \rangle}
\end{array}$$

Fig. 3. CHR Multi-Set Rule Priority Match Semantics

which match the remaining constraints H_1 and H_2 in the head. The symbol $=_{set}$ denotes set equality among lists. The statement $St_1 ++ St_2 ++ St' =_{set} St$ holds if each element in $St_1 ++ St_2 ++ St'$ appears in St and vice versa. This implies that the order of constraints in patterns does not matter which is a sensible choice for our (actor) setting as argued in Section 2. The equality test among constraints ignores numbering of constraints and occurrences. If the guard can be satisfied as well, we report the successfully found match. Otherwise, we continue our search by incrementing the occurrence number of the active constraint. See rule (Continue). This is only sensible if the maximum occurrence in any constraint in \mathcal{RP} , computed via function $maxOccur(\cdot)$, is smaller than the current occurrence number. Otherwise, we deactivate the constraint by putting it into the store. See rule (Deactivate). The order among messages is retained. That is, for any initial state $\langle M, [] \rangle$ and intermediate state $\langle M', St \rangle$ we have that $M = St ++ M'$. We keep repeatedly applying rules (Activate), (Match), (Continue) and (Deactivate), in that order, until we either reach a successful or failure state.

To summarize, the first-match semantics finds a match by processing constraints in sequential order and checking for a matching rule pattern from top to bottom (in the textual order).

3.2 Rule Priority-Match Semantics

We consider a rule priority-match semantics which guarantees that rule patterns are executed in (textual) order. Figure 3 contains the details. We apply the first-match semantics on each rule pattern and select the first successful match in textual order. We assume that RP_j appears before RP_{j+1} in the program which can be specified via occurrences associated to head constraints.

Next, we consider some examples to illustrate the differences between both semantics.

Receive clause:

```

receive act of
  A,A -> "RP1"    -- RP1 = A : 1 ∧ A : 2
  B   -> "RP2"    -- RP2 = B : 3

```

$$\mathcal{RP} = \{RP_1, RP_2\}$$

First-Match reduction:

```

                                ⟨[A#1, B#2, A#3], []⟩
(Act-Cont-Deact)  →First- $\mathcal{RP}$  ⟨[B#2, A#3], [A#1]⟩
(Activate)       →First- $\mathcal{RP}$  ⟨[B#2 : 1, A#3], [A#1]⟩
(Continue ×2)    →First- $\mathcal{RP}$  ⟨[B#2 : 3, A#3], [A#1]⟩
(Match)          →First- $\mathcal{RP}$  ⟨[A#3], [A#1], identSubst, RP2⟩

```

Rule Priority-Match reduction:

```

                                ⟨[A#1, B#2, A#3], []⟩
(Act-Cont-Deact)  →First- $\{RP_1\}$  ⟨[B#2, A#3], [A#1]⟩
(Act-Cont-Deact)  →First- $\{RP_1\}$  ⟨[A#3], [A#1, B#2]⟩
(Activate)        →First- $\{RP_1\}$  ⟨[A#3 : 1], [A#1, B#2]⟩
(Match)           →First- $\{RP_1\}$  ⟨[], [B#2], identSubst, RP1⟩

```

Fig. 4. Example 1

3.3 Examples

The first example is given in Figure 4. We assume that A and B are constant messages. Therefore, each (Match) reductions make use of the identity (matching) substitutions $identSubst$. Each reduction step is annotated with the corresponding reduction rule. For brevity, we shorten reduction steps. For example, we write

```

                                ⟨[A#1, B#2, A#3], []⟩
(Act-Cont-Deact)  →First- $\mathcal{RP}$  ⟨[B#2, A#3], [A#1]⟩

```

as a short-hand for

```

                                ⟨[A#1, B#2, A#3], []⟩
(Activate)        →First- $\mathcal{RP}$  ⟨[A#1 : 1, B#2, A#3], []⟩
(Continue ×3)     →First- $\mathcal{RP}$  ⟨[A#1 : 4, B#2, A#3], []⟩
(Deactivate)      →First- $\mathcal{RP}$  ⟨[B#2, A#3], [A#1]⟩

```

The first-match reduction applies RP_2 . We sequentially process constraints, searching for the first match for a rule pattern from top to bottom. Starting with the initial list of executables $[A\#1, B\#2, A\#3]$, we find that $B\#2$ form the first match for rule pattern RP_2 . On the other hand the rule priority-match reduction applies RP_1 . We strictly apply rule patterns in (textual) order. Based on the priority of rules, $A\#1, A\#3$ form a match for the first rule pattern RP_1 .

In the second example in Figure 5, we apply the first-match and rule priority-match on the initial list of executables $[A\#1, B\#2]$. In both cases only rule

Receive clause:

receive act of

$$\begin{array}{ll} A, A \rightarrow "RP_1" & \text{-- } RP_1 = A : 1 \wedge A : 2 \\ B \rightarrow "RP_2" & \text{-- } RP_2 = B : 3 \end{array}$$

$$\mathcal{RP} = \{RP_1, RP_2\}$$

First-Match reduction:

$$\begin{array}{ll} & \langle [A\#1, B\#2], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1, RP_2\}} \langle [B\#2], [A\#1] \rangle \\ \text{(Activate)} & \longrightarrow_{First-\{RP_1, RP_2\}} \langle [B\#2 : 1], [A\#1] \rangle \\ \text{(Continue } \times 2) & \longrightarrow_{First-\{RP_1, RP_2\}} \langle [B\#2 : 3], [A\#1] \rangle \\ \text{(Match)} & \longrightarrow_{First-\{RP_1, RP_2\}} \langle [], [A\#1], identSubst, RP_2 \rangle \end{array}$$
Rule Priority-Match reductions:

$$\begin{array}{ll} & \langle [A\#1, B\#2], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1\}} \langle [B\#2], [A\#1] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1\}} \langle [], [A\#1, B\#2] \rangle \\ \\ & \langle [A\#1, B\#2], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_2\}} \langle [B\#2], [A\#1] \rangle \\ \text{(Activate)} & \longrightarrow_{First-\{RP_2\}} \langle [B\#2 : 1], [A\#1] \rangle \\ \text{(Continue } \times 2) & \longrightarrow_{First-\{RP_2\}} \langle [B\#2 : 3], [A\#1] \rangle \\ \text{(Match)} & \longrightarrow_{First-\{RP_2\}} \langle [], [A\#1], identSubst, RP_2 \rangle \end{array}$$
Fig. 5. Example 2

pattern RP_2 applies. The first-match reduction is almost identical to Example 1 where we additionally find constraint $A\#3$ in the initial M . But this constraint does not contribute to the first match. In case of the rule priority-match reduction we first try RP_1 which fails and then we try RP_2 which leads to success.

4 Actors with Multi-Headed Message Patterns

Figure 6 introduces the syntax and Figures 7 and 8 introduce the semantics of an elementary actor language which supports multi-headed message patterns. In example programs, we use “,” (comma) to separate multi-headed message patterns whereas in our (internal) syntax we use \wedge . We assume a distinct pattern variable *otherwise* to support otherwise statements. The *otherwise* pattern (if present) does not appear anywhere else in the program but in the last pattern of a receive statement. We assume that the variables in a guard statement e'_i appear in the associated pattern p_i .

We define the semantics in terms of a small-step Wright/Felleisen style semantics [20]. We assume a fixed set of actors, each identified by a unique actor identification number, *aid* for short. Each actor has a mailbox M and the actor's

Expressions

$e ::= x$	Variable
$K e \dots e$	Message
$\lambda x.e \mid e e$	Function and application
receive $[p_i \text{ when } e'_i \rightarrow e_i]_{i \in I}$	Message receive
send $aid e$	Message send
$()$	Don't care
$p ::= p' \mid p \wedge p$	Single-head and multi-head pattern
$p' ::= x$	Variable pattern
otherwise	Otherwise pattern
$K p' \dots p'$	Message pattern

Actor

$a ::= (aid, M, e)$	
aid	Actor identification
M	Mailbox
e	Behavior

Fig. 6. MiniActor Language

behavior is specified by an expression e . We execute actors in random order. See rule (Schedule) in Figure 8. We simply evaluate the actor expression k number of steps. Evaluation affects the actors mailbox and has as a side effect the sending of messages. We append sent messages to the appropriate mailboxes via the operations $S@a$ and $S@AP$. See rules (AS1-4). In rule (AS3), we attach a unique number to the message to distinguish multiple occurrences of the same message.

Evaluation of expressions is described in Figure 7. Rule (Send) yields a don't care expression but has the side effect of sending a message. Side effects are collected in a multi-set of constraints. We may send the same message twice to the same actor. The symbol \uplus denotes multi-set union. We do not care much about the order of sent messages which may be random. Evaluation of receiving of messages is parametric in terms of the match semantics described earlier. We first describe the general receive rules in terms of a generic-match reduction $\longrightarrow_{X-\mathcal{RP}}^*$ before we consider the impact of a specific matching policy.

Matching starts in the initial state $\langle M, [] \rangle$ where M is the actor's current mailbox. In rule (Receive) we have found a successful match. From the successful state $\langle M', St, \theta, p_j \text{ when } e'_j \rightarrow e_j \rangle$ we collect the list St of already processed messages which have not been involved in the matching. We put these messages back into the actor's mailbox in their original order (see also rule (Deactivate) in Figure 2). We then continue executing the successful receive body $\theta(e_j)$. In the (Otherwise) case, we leave the mailbox unchanged. There is no rule for covering failure which means that evaluation of a receive clause will block until a successful match is found.

In case we instantiate $\longrightarrow_{X-\mathcal{RP}}^*$ with the first-match reduction relation from Section 3, we obtain a conservative extension of Erlang-style actors with multi-

Values

$$v ::= \lambda x.e \mid K v_1 \dots v_n \mid ()$$

Send effects

$$S ::= \emptyset \mid \{\text{send aid } K v_1 \dots v_n\} \mid S \uplus S$$

Evaluation contexts:

$$E ::= [] \mid E v \mid K E \dots E \mid \text{receive } [p_i \text{ when } E \rightarrow e_i]_{i \in I} \mid \text{send aid } E$$

Expression reduction: $\langle M, e \rangle \xrightarrow{S} \langle M, e \rangle$

$$\text{(Beta)} \quad \langle M, (\lambda x.e) v \rangle \xrightarrow{\emptyset} \langle M, [v/x]e \rangle$$

$$\text{(Send)} \quad \frac{S = \{\text{send aid } K v_1 \dots v_n\}}{\langle M, \text{send aid } K v_1 \dots v_n \rangle \xrightarrow{S} \langle M, () \rangle}$$

$$\text{(Receive)} \quad \frac{\begin{array}{l} \mathcal{RP} = \{p_1 \text{ when } e'_1 \rightarrow e_1, \dots, p_n \text{ when } e'_n \rightarrow e_n\} \\ \langle M, [] \rangle \xrightarrow{*}_{X-\mathcal{RP}} \langle M', St, \theta, p_j \text{ when } e'_j \rightarrow e_j \rangle \\ p_j \neq \text{otherwise for some } j \in \{1, \dots, n\} \\ M'' = St \upuparrows M' \end{array}}{\langle M, \text{receive } [p_i \text{ when } e'_i \rightarrow e_i]_{i \in \{1, \dots, n\}} \rangle \xrightarrow{\emptyset} \langle M'', \theta(e_j) \rangle}$$

$$\text{(Otherwise)} \quad \frac{\begin{array}{l} \mathcal{RP} = \{p_1 \text{ when } e'_1 \rightarrow e_1, \dots, p_n \text{ when } e'_n \rightarrow e_n\} \\ \langle M, [] \rangle \xrightarrow{*}_{X-\mathcal{RP}} \langle M', St, \theta, p_n \text{ when } e'_n \rightarrow e_n \rangle \\ p_n = \text{otherwise} \end{array}}{\langle M, \text{receive } [p_i \text{ when } e'_i \rightarrow e_i]_{i \in \{1, \dots, n\}} \rangle \xrightarrow{\emptyset} \langle M, e_n \rangle}$$

$$\text{(Context)} \quad \frac{\langle M, e \rangle \xrightarrow{S} \langle M', e' \rangle}{\langle M, E[e] \rangle \xrightarrow{S} \langle M', E[e'] \rangle} \quad \text{(Step)} \quad \frac{\langle M, e \rangle \xrightarrow{S} \langle M', e' \rangle}{\langle M, e \rangle \xrightarrow{*} \langle M', e' \rangle}$$

$$\text{(k-Step)} \quad \frac{\langle M_1, e_1 \rangle \xrightarrow{S_1} \langle M_2, e_2 \rangle \dots \langle M_{k-1}, e_{k-1} \rangle \xrightarrow{S_{k-1}} \langle M_k, e_k \rangle}{S_k = S_1 \uplus \dots \uplus S_{k-1}}{\langle M_1, e_1 \rangle \xrightarrow{S_k} \langle M_k, e_k \rangle}$$

$$\text{(Trans)} \quad \frac{\langle M_1, e_1 \rangle \xrightarrow{S_1} \langle M_2, e_2 \rangle \quad \langle M_2, e_2 \rangle \xrightarrow{S_2} \langle M_3, e_3 \rangle}{\langle M_1, e_1 \rangle \xrightarrow{S_1 \uplus S_2} \langle M_3, e_3 \rangle}$$

Fig. 7. Expression Semantics

headed message receive patterns. The first-match semantics guarantees the following **Monotonicity Property**:

$$\text{If } \langle M, [] \rangle \xrightarrow{*}_{\text{First-}\mathcal{RP}} \langle M', St, \theta, p_j \text{ when } e'_j \rightarrow e_j \rangle \\ \text{then } \langle M \upuparrows M'', [] \rangle \xrightarrow{*}_{\text{First-}\mathcal{RP}} \langle M' \upuparrows M'', St, \theta, p_j \text{ when } e'_j \rightarrow e_j \rangle$$

Actor pool

$$AP ::= \emptyset \mid \{a\} \mid AP \cup AP$$

Actor send: $S@a$ and $S@AP$

$$(AS1) \quad \emptyset@(aid, M, e) = (aid, M, e)$$

$$(AS2) \quad \frac{aid \neq aid'}{\{\text{send } aid \ K \ v_1 \dots v_n\} \uplus S@(aid', M, e) = S@(aid', M, e)}$$

$$(AS3) \quad \frac{aid = aid' \quad \text{unique number } m}{\{\text{send } aid \ K \ v_1 \dots v_n\} \uplus S@(aid', M, e) = S@(aid', M ++ [K \ v_1 \dots v_n \ #m], e)}$$

$$(AS4) \quad S@\{a_1, \dots, a_n\} = \{S@a_1, \dots, S@a_n\}$$

Actor reduction: $AP \longrightarrow AP$

$$\begin{aligned} & AP = \{(aid, M, e)\} \cup AP' \\ \text{(Schedule)} \quad & \frac{\langle M, e \rangle \xrightarrow{S}^k \langle M', e' \rangle}{AP'' = \{S@(aid, M', e')\} \cup S@AP'} \\ & \frac{}{AP \longrightarrow AP''} \\ \text{(Step)} \quad & \frac{AP \longrightarrow AP'}{AP \longrightarrow^* AP'} \quad \text{(Trans)} \quad \frac{AP_1 \longrightarrow^* AP_2 \quad AP_2 \longrightarrow^* AP_3}{AP_1 \longrightarrow^* AP_3} \end{aligned}$$

Fig. 8. MiniActor Semantics

This property says that any successful match remains valid if further messages arrive in the actor's mailbox. This is a fairly important property and shows that we can treat the actor's mailbox as a "lazy" structure. That is, the mailbox represents a stream of incoming messages.

If we employ the rule priority-match semantics from Section 3, however, newly arrived messages can invalidate earlier (match) choices. For instance, consider the rule priority-match reduction in Figure 5. Suppose that at some later stage the message $A\#5$ arrives (already attached with a unique number). The rule priority-match reduction in Figure 4 shows that in this case, a different rule pattern may be applied.

On the other hand, under a rule priority-match semantics we can read off the priorities directly from the receive clauses. Under a first-match semantics, we need to explicitly program priorities via **otherwise** and nested receive statements. This often leads to clumsy and hard to maintain code. See the discussion in Section 2. In summary, we believe that both semantics represent interesting, alternative design choices for an actor language, and it will depend on the application which semantics is the better choice.

5 Implementation

We have implemented a prototype as a library extension in Haskell using the Glasgow Haskell Compiler [7]. GHC supports light-weight threads. Therefore, our implementation scales well to many actors. The latest version including examples can be downloaded via [9].

We briefly highlight the main features of our implementation. We support strongly typed actors in the sense that an actor’s mailbox can only holds messages of a certain (data) type. The actual mailbox consists of two parts. A buffer for recently sent messages is represented as a transacted channel to manage conflicts among multiple writers. A transacted channel is a linked list in shared memory where access is protected by Software Transactional Memory. In the future we plan to support distributed channels to support sending of messages across the network. The second part of the mailbox is a linked list to process $\langle M, St \rangle$ by a single reader. We use pointers to indicate the start of M and St .

Our implementation applies the first-match scheme outlined in Section 3. We process messages in M in sequential order. If M is empty, we check the buffer and transfer any recently sent messages to M . If the buffer is empty, we wait for new messages to arrive. Our current prototype [9] performs a sequential search for matching messages. To improve the performance, one possible optimization is message indexing. For example, consider rule pattern `Sell x, Buy x`. Suppose the active message `Sell SomeObject` in M matches part of the rule head. We then need to find a matching partner `Buy SomeObject` in St . We can achieve a faster lookup of candidate partners by using (hash)-indexing. Another optimization is the early scheduling of guards. For example, consider the rule pattern `Foo x, Bar y, Erk z when x > y`. Suppose that `Foo 1` is our active message and what remains is to find matching partners `Bar y` and `Erk z` for some y and z such that $x > y$. As soon as we have found a possible candidate `Bar y` we should schedule (i.e. test) the guard $1 > y$ to reduce the search space. We plan to integrate the above and other common CHR optimizations [10, 16]. into later versions of our system.

6 Related Work

In their foundational work, Kahn and Saraswat [11] establish connections between the actor programming model and concurrent constraint logic programming. Our work follows their footsteps by providing a formal model for multi-headed message receive patterns with guards based on CHR style multi-set constraint matching semantics.

The basic motivation and idea of multi-headed message receive patterns can already be found in the earlier work [5]. The concepts of receptionists and activators introduced in [5] correspond to what we called heads and rules respectively. However, there are noticeable differences. In [5], receptionists and activators are first class entities that can be communicated. Also, activators may be combined using both conjunction and disjunction. Whilst in [5], non-determinism issues are resolved by introducing fairness conditions, we consider a conservative extension

of Erlang style actors, and propose a (in the context of actors) novel rule-priority match semantics. Our prototype also shows that the language extension can be implemented efficiently.

The main advantage of multi-headed message receive patterns is the ability to expression complex synchronization patterns. In [5] two types of synchronization are considered: input and reply synchronization; [19] only considers the latter. For reply synchronization, we will consider some syntactic sugar, as in [5, 19], that allows actors to reply on messages. We believe that in our system we can express all synchronization patterns found in these works. Reply delegation (cf. [5, 19]) for instance is already possible by communicating the requesting actor's address. Further work is needed to support this claim.

Closest to our work is some recent work by Haller and Van Cutsem [8]. Like us they use the abstractions in the language (they use Scala we use Haskell) to avoid non-sensical pattern specifications. They focus on the implementation of join patterns [4] by means of extensible pattern matching. There are close connections between the join and actor model, and their system has support for join-style actors (what we call multi-headed message receive patterns). However, their approach appears to be more limiting. They can only support a limited form of guards, it is unclear whether they can support our non-linear patterns at all. Furthermore, they do not specify the semantics of their system.

7 Conclusion

We have studied an extension of Erlang style multi-headed message receive patterns with guards. Such an extension is useful as supported by a number of examples. We have explored two possible semantics by adapting previously studied CHR multi-set matching semantics. The first-match semantics gives us a conservative extension of Erlang style actors to the setting of multi-headed message receive patterns with guards. We have also explored a rule priority-match semantics which guarantees that rule patterns are executed in (textual) order. For certain applications this semantics is the better choice. The original CHR semantics proposed in [12] is more general and can express more complicated, even dynamic, user-defined priorities. How to exploit more complex priority-based execution control in the actor setting is subject of future work.

Both semantics can be implemented efficiently as shown by previous work [3, 13, 16]. Our library-based prototype exploits some of these methods. We plan to integrate further optimizations, and conduct more experimentations in future work.

In another line of work, we will enrich join patterns with CHR style guards and non-linear patterns. In the join context, patterns can be executed concurrently. We wish to parallelize the concurrent execution of join patterns. Contrast this to actor receive patterns which are executed sequentially (either following the first-match or rule priority-match semantics). We have already started work on the parallelization of CHR [14], and explored a CHR style enriched join pattern language [17]. We plan to report more detailed results in the future.

Acknowledgments

We thank the reviewers for their helpful comments.

References

1. G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
3. G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaaur. The refined operational semantics of Constraint Handling Rules. In *Proc. of ICLP'04*, volume 3132 of *LNCS*, pages 90–104. Springer-Verlag, 2004.
4. C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000*, pages 268–332, Caminha, Portugal, 2002. Springer-Verlag.
5. S. Frølund and G. Agha. Abstracting interactions based on message sets. In *Proc. of ECOOP '94: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 107–124. Springer-Verlag, 1995.
6. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
7. Glasgow haskell compiler home page. <http://www.haskell.org/ghc/>.
8. P. Haller and T. Van Cutsem. Implementing Joins using extensible pattern matching. In this volume.
9. HaskellActor. <http://code.google.com/p/haskellactor/>.
10. C. Holzbaaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *TPLP*, 5(4-5):503–531, 2005.
11. K. Kahn and Vijay A. Saraswat. Actors as a special case of concurrent constraint (logic) programming. In *Proc. of OOPSLA/ECOOP*, pages 57–66. ACM, 1990.
12. L. De Koninck, T. Schrijvers, and B. Demoen. User-definable rule priorities for CHR. In *Proc. of PPDP'07*, pages 25–36. ACM, 2007.
13. L. De Koninck, P.J. Stuckey, and G.J. Duck. Optimizing compilation of CHR with rule priorities. In *To appear in Proc. of FLOPS'08*, 2008.
14. E. S. L. Lam and M. Sulzmann. A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory. In *Proc. of DAMP'07*, pages 19–24. ACM Press, 2007.
15. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
16. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
17. M. Sulzmann and E. S. L. Lam. Haskell – Join – Rules. In Draft Proc. of IFL'07, September 2007.
18. J. A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.
19. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.
20. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.