Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2017

λir : A language with intensional receive

Swarn Priya
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Priya, Swarn, "λir : A language with intensional receive" (2017). *Graduate Theses and Dissertations*. 16198.
https://lib.dr.iastate.edu/etd/16198

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

المنارة للاستشارات

www.manaraa.com

λ_{ir}: A language with intensional receive

by

Swarn Priya

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Gianfranco Ciardo, Major Professor
Samik Basu
Steven Kautz

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

Copyright © Swarn Priya, 2017. All rights reserved.

www.manaraa.com

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS vii

ABSTRACT viii

1. INTRODUCTION 1

1.1 Contributions 3

2. MOTIVATING EXAMPLES 5

2.1 Pipelining 5

2.2 Encoding State Machines 6

2.3 Chain of responsibility design pattern 7

2.4 Multiplexing 9

3. DESIGN OF INTENSIONAL RECEIVE: ABSTRACT SYNTAX 11

3.1 Single-state Processes 12

3.2 Intensional Design of the Receive Expression 12

4. DYNAMIC SEMANTICS OF λ_{ir} PROGRAMS 15

4.1 Dynamic Objects 15

4.2 Local and Global Semantics 16

4.3 Sequential Synchronous Local Semantics 16

4.4 Concurrent Asynchronous Global Semantics 18

5. STATIC SEMANTICS OF λ_{ir} PROGRAMS 20

5.1 Computing Potential Behavior of a Recipient 20

www.manaraa.com

5.2 Pipelining (revisited) 22

5.3 Chain of responsibility design pattern (revisited) 24

5.4 Subtyping, Value, and Communication Types 26

5.5 Type Rules 27

6. INTENSIONAL RECEIVE GENERATES NEW HAPPENS-BEFORE RELATIONS **30**

6.1 Actions: Conflict and Happens-Before Relations 30

6.2 Guaranteed Delivery 32

7. EXTENSIONS **35**

7.1 Broadcast 35

7.2 Multicast 36

7.3 Guarded Receive 38

7.4 Non-blocking Receive 39

7.5 Synchronization primitive, Wait 39

7.6 Extension in properties 41

8. FORMALIZATION **43**

8.1 Parallel Composition 44

8.2 Formalization of the extensions 46

9. RELATED WORK **49**

10.CONCLUSION **52**

BIBLIOGRAPHY **54**

APPENDIX. ADDITIONAL MATERIAL **59**

A.1 Dynamic Semantics of STLC expressions 59

A.2 Static Semantics of STLC expressions 60

المنارة للاستشارات

www.manaraa.com

A.3 Happens-Before Relations 60

A.4 Type Inference Algorithm inspired by Hindley and Milner Approach 63

المنارة للاستشارات

www.manaraa.com

LIST OF FIGURES

Page

Figure 2.1 An example of the receive expression supporting pipeline architecture. 6

Figure 2.2 An example of the receive expression encoding state machines. 7

Figure 2.3 An example of the receive expression representing chain of responsibility design pattern. 8

Figure 2.4 An example of the receive expression supporting multiplexing. 9

Figure 3.1 λ_{ir} Abstract Syntax. (In this thesis, Notation $\overline{\bullet}$ Represents a Set of $\bullet$ Elements.) 11

Figure 3.2 An example of the receive expression performing different tasks based on the type of the message. Note expressions on lines 3 and 5 that treat messages from the same mailbox differently. 13

Figure 3.3 Syntactic sugar for multiple receive and its desugaring 13

Figure 4.1 Definitions used in λ_{ir}'s Dynamic Semantics 15

Figure 4.2 Operational Semantics 17

Figure 5.1 Example showing the communication type of two interacting processes 21

Figure 5.2 Communication type of initstg and finstg 23

Figure 5.3 Communication type of midstg 23

Figure 5.4 Communication type of hndlr1 and hndlr2 24

Figure 5.5 Communication type of rqst 25

Figure 5.6 Subtyping relations for value types. 26

Figure 5.7 Subtyping relations for communication types. 26

Figure 5.8 Typing Rules for λ_{ir}. 27

المنارة للاستشارات

www.manaraa.com

Figure 5.9 Auxiliary Functions 28

Figure 6.1 Conflicting actions in the Message Passing Model based on type of message received. 31

Figure 6.2 Encoding vending machine and reasoning about its correctness. 31

Figure 7.1 Syntax, action, dynamic semantics, and static semantics for Broadcast. . . . 36

Figure 7.2 Syntax, action, dynamic semantics, and static semantics of Multicast. 37

Figure 7.3 Syntax, dynamic semantics, and static semantics for Guarded Receive. 38

Figure 7.4 Syntax, dynamic semantic and static semantic for Non-blocking receive. . . . 40

Figure 7.5 Syntax, dynamic semantics, and static semantics for Wait. 41

Figure 7.6 Conflicting actions in Message Passing Model for the extensions where # denotes conflict and their happens-before $\prec$ relation. 42

Figure 8.1 Parallel composition library 44

Figure 8.2 Structural Congruence Rules. 45

Figure 8.3 Theorems related to Congruence 46

Figure 8.4 Lines of code in Coq. (SYN : Syntax, SEM : Semantics, TYPE : Typing rules, GD : Guaranteed delivery, HB : Happens-before relation, Total : Total number of lines of code, + : Number of lines of code added in each of the module for each extension). 47

Figure A.1 Operational Semantics 59

Figure A.2 Static Semantics 60

Figure A.3 Conflicting actions in the Message Passing Model where # denotes conflict and their *happens-before* $\prec$ relation. 61

Figure A.4 Variable representing value type and communication type 63

المنارة للاستشارات

www.manaraa.com

ACKNOWLEDGEMENTS

I would like to thank a lot of people around me who helped me with various aspects of research and writing this thesis. First and foremost, special thanks to Dr. Gianfranco Ciardo for agreeing to be my major professor at very last moment and for his support towards my master's degree. I would also like to thank Dr. Kristin Yvonne Rozier for her valuable feedback and encouragement on this work. I would also like to thank my lab mates for helping me out in various ways in my research and thesis. I also want to thank my POS committee members, Dr. Samik Basu and Dr. Steve Kautz.

للاستشارات المنارة

www.manaraa.com

ABSTRACT

Message passing-based programming is one of the dominant concurrent programming models today in both research and practice. The major challenge in message passing concurrency is to reason about the type of message received by any process and its effect. We present λ_{ir}, a message passing-based language that incorporates an intensional design of the receive expression to solve this problem. Intensional design of receive expression integrates static and dynamic type checking and allows the effect of the message received to be intensionally inspected through a notion of dynamic typing. This enables reasoning about the effect of the message received from the head of the mailbox while retaining static type safety. We demonstrate the applications of intensional design of receive expression in various programming patterns like multiplexing, safe pipelining, encoding state machines and supporting the chain of responsibility pattern. In each of these applications, intensional receive helps in providing better safety. We have also formalized λ_{ir} using the Coq proof assistant and prove its soundness. λ_{ir} provides built-in proofs for guaranteed delivery of messages and encodes actions as an integral part that makes it possible to describe and prove properties about happens-before relations. λ_{ir} comes with a range of extensions like broadcasting, multicasting, guarded receive, non-blocking receive, and synchronization primitive "wait" that not only provide insights into the extensibility of the calculus, but also provide pedagogical examples of how it can be extended.

للاستشارات المنارة
www.manaraa.com

CHAPTER 1. INTRODUCTION

Programming support for concurrency has come full circle. Hoare's communicating sequential processes [17], Hewitt and Agha's actors [16, 1], Milner's Π calculus [23] and Bagherzadeh and Rajan's concurrent calculus [28] [3] promoted the ideas that programs can be formed by composing independently executing entities that communicate with each other via messages. This model was embraced by several language designs early on, e.g. Reppy's CML [30], Erlang [33], Pierce and Turner's PICT [26], but mainstream languages remained focused on developing "lightweight threads" as the dominant concurrency abstraction. Over the last decade, message passing concurrency has re-emerged as one of the dominant programming models for concurrency, as evidenced by the success of Scala actors [13], Akka [5], Panini [3] and Go [11] among others. The future of embedded systems depends on developments in message-passing architectures, including the Robot Operating System (ROS) [27] and NASA's new Autonomy Operating System (AOS) [22]. There is significant interest in understanding the theoretical properties of the message passing model further for enabling reasoning about behaviors of critical concurrent programs, e.g. for flight-certification of message-passing architectures [22], and safety properties of robotic systems [27].

To that end, this thesis introduces *intensional receive*, a novel formulation of the receive operation, to enable more precise, static modular type-checking of message passing programs. To illustrate the motivation behind intensional receive consider the following simple code snippet that has a traditional receive operation in the first and the second line that accepts an incoming message and binds it to the name x and y respectively to be used in the continuation of the receive operation. The continuation of the first receive operation is the entire expression in lines 2 and 3, and the continuation of the second receive operation is the addition operation on the third line.

```
1 (receive x
2  (receive y
```

www.manaraa.com

```
3   (add x y)
4  )
5 )
```

Static, modular type-checking of this simple code is difficult in the presence of a range of dynamic language features like reflection, dynamic loading, native code interfaces, etc., that eliminate all hopes of having static access to all message send locations.

Our proposal λ_{ir} is designed to support reasoning about the effect of receiving any message from the mailbox and statically type checking the message received. Inspired by Harper and Morrisett [14], we achieve this by incorporating an intensional type system design, where process can intensionally inspect the type of message at run time. Specifically, expression **receive** $x : T\ e_1\ e_2$ inspects whether the runtime (message) type of x is T, and evaluates e_1 if so, or e_2 otherwise.

This design has two distinct advantages. First, static typing can guarantee that the expression **receive** $x : T\ e_1\ e_2$ is well-typed regardless of where it is applied. Message received from a process is always type checked at static time. Static safety is achieved for the messages. Indeed this advantage can be seen in an intensional version of our example from above that is presented below.

```
1 (receive (x : nat)
2  (receive (y : nat)
3   (add x y)
4   unit
5  )
6  unit
7 )
```

The first receive operation `receive x : nat ... unit` says to evaluate the expression `receive y : nat ... unit` if the message received is of type `nat` otherwise evaluate to `unit` expression. The expression can be type-checked independent of its message send sites, a big win for modularity.

www.manaraa.com

Second, the use of dynamic typing provides more precise reasoning for the expression at run time. To illustrate consider a message send site like in line 9 in the code below that defies static reasoning but leads to process p computing 42 in λ_{ir}.

```
1 (let p =
2  (spawn
3   (receive (x : nat)
4    (receive (y : nat)
5     (add x y)
6     unit
7    )
8    unit
9   )
10  ) in
11  (send (if (< 1 0) #f 41) p) ;
12  (send 1 p)
13 )
```

The rest of this thesis describes this design of λ_{ir}, its advantages, and its challenges.

1.1 Contributions

In summary, this thesis makes the following contributions.

- It presents λ_{ir}, a calculus for message passing concurrency with intensional receive.
  - Like [10], we extend the simply-typed lambda calculus (STLC) to build λ_{ir} to leverage the strong meta-theory that has already been developed for STLC. As a result, our expression-based calculus has an STLC-like style and leverages familiarity.
  - λ_{ir} provides an intensional design of the receive primitive that leverages dynamic typing.

المنارة للاستشارات
www.manaraa.com

- λ_{ir} incorporates infrastructure to reason about both type soundness and concurrency-related properties that build on the happens before relation. To show the utility of λ_{ir} regarding its ability to support formalization of properties about message passing program, we state and prove two properties: *guaranteed delivery* and the *happens-before* relation between actions in the trace of the program [19].

- The design of λ_{ir} incorporates standard value types, plus we incorporate communication types that describe an upper bound on communication effects of the expressions, inspired by [18]. We show an illustrative example of using communication types to detect type incompatibilities between senders and recipients in the same program.

- It presents a mechanized realization of λ_{ir}. We have mechanized λ_{ir} using Coq proof assistant.

- It demonstrates the impact of the intensional receive for improving type-safety in application patterns such as pipelining, encoding state machines, chain of responsibility design pattern and multiplexing.

- five extensions of λ_{ir} (§7): broadcast, multicast, guarded receive, non-blocking receive, and wait. These new features have helped us understand the extensibility of our calculus. Several of these features required under 200 lines of Coq code, with multicast being the exception in that it required 306 lines of Coq code.

The rest of the thesis is organized as follows: Chapter 2 presents impact of the intensional design of receive expression; Chapter 3 presents the syntax of λ_{ir} with examples illustrating its use; Chapter 4 presents the operational semantics of λ_{ir}; Chapter 5 describes the static semantics of λ_{ir}; Chapter 6 describes the properties supported by λ_{ir}; Chapter 7 describes the extensions supported by λ_{ir}; Chapter 8 describes the noteworthy aspects of our Coq formalization; Chapter 9 discusses related ideas and Chapter 10 concludes.

المنارة للاستشارات
www.manaraa.com

CHAPTER 2. MOTIVATING EXAMPLES

In this section, we demonstrate the applicability of intensional design of receive expression for providing better safety for many programming patterns like pipelining, encoding state machines, chain of responsibility pattern and multiplexing. In each programming pattern, we demonstrate how intensional design of receive expression which incorporates both static and dynamic typing makes reasoning more precise and provides better safety.

In all the examples, we use `receive x :  T e` as a shorthand for `receive x : T e unit`, and we use `P` as the type of a process for simplicity. The full representation of a process type is discussed in §3.

2.1 Pipelining

Pipelining structure is widely used in both hardware and software where each component (called pipeline stage) has a set of inputs and a set of outputs. Each stage takes data on its input and produces data on its output which acts as an input for the next stage component. For the correct implementation of pipeline structure, there should be type compatibility between the various stages. Pipeline structure can go faulty due to incompatibility in types of the output produced by one stage which is consumed by another stage. The intensional design of the receive operation can help avoid type incompatibility between pipeline stages in a message passing-based pipeline architecture. An example λ_{ir} program is shown in Figure 2.1 that spawns three processes that are first class values in λ_{ir} and binds them to names `initstg`, `midstg` and `finstg`.

There can be two kinds of incompatibilities. The first kind of incompatibility can arise at pipeline setup time, e.g. at lines 25-27 in Figure 2.1. For example, `initstg` could be connected to a value that is not a pipeline stage process. These cases are immediately detected due to the newfound type information in the intensional receive. Somewhat more interestingly, `initstg` could

المنارة للاستشارات

www.manaraa.com

```
1  (let finstg =                      17     (spawn (initnext : P)
2   (spawn (s : nat)                  18      (receive (next : P)
3    (receive (a : nat)               19       (set next)
4     (set (add a a a a 2))           20       (receive (m : nat)
5    )                                21        (send (get)
6   ) in                              22              (pair 0 m))
7   (let midstg =                     23       )
8    (spawn (midnext : P)             24      )
9     (receive (next : P)             25     ) in
10     (set next)                     26     (send initstg midstg) ;
11     (receive (m : nat * nat)       27     (send midstg finstg) ;
12      (send (get) (snd m))          28     (send initstg 10)
13     )                              29    )
14    )                               30   )
15   ) in                             31  )
16   (let initstg =
```

Figure 2.1 An example of the receive expression supporting pipeline architecture.

be connected to a value that is a process, but of incompatible kind. We will see in §5.2 that these problems are also resolved by refining `P` further. The second kind of incompatibility can arise when the pipeline is processing data, e.g. at lines 3-5, 11-13, and 20-22. These kinds of incompatibilities can also be checked and eliminated. Notice that the `initstg` and `midstg` processes can receive messages of multiple kind and exhibits incompatible behavior in response to each kind of message. So, parametric polymorphism doesn't immediately work here.

2.2 Encoding State Machines

As another application of intensional design of receive expression, consider its usage in encoding state machines. Intensional receive expression can be used to receive messages in an order from the mailbox and perform computation on them. A state machine encoding can be faulty if the order of transition from one state to another is not maintained. For example `halfadder` in Figure 2.2 works only on two inputs of type `bool` provided in an order and produces sum and carry[1].

[1] For simplicity we do not encode the output, but extending this example to send messages to an output process would be straightforward.

www.manaraa.com

```
1 (let halfadder =
2  (spawn (p : bool * bool)
3   (fix (receive (x : bool)
4         (receive (y : bool)
5          (if (and (equal x false)
6                   (equal y false)
7              )
8           (set (pair false false))
9            else if (and (equal x true)
10                        (equal y true)
11                   )
12             (set (pair false true))
13              else (or (and (equal x true)
14                            (equal y false)
15                       )
16                        (and (equal x false)
17                             (equal y true)
18                       )
19                   )
20               (set (pair true false))
21           )
22          )
23     )
24    ) in
25    (send halfadder true) ;
26    (send halfadder false)
27  )
```

Figure 2.2 An example of the receive expression encoding state machines.

The use of intensional receive makes the state machine's behavior at the type-level explicit in the source code. Furthermore, incompatibilties between the type and order in which inputs are provided to the `halfadder` can also be detected during typechecking, e.g. at lines 25-26.

2.3 Chain of responsibility design pattern

The chain-of-responsibility patterns is a commonly-used software design pattern to organize components that make a request (requester) and other components that handle the request (handler) in a manner that avoids coupling requester with knowledge about which handler handles which request. This pattern eliminates the need to couple the sender to the compatible handler as

www.manaraa.com

the handler who cannot handle the message forwards the message to the next handler in the chain. The last (default) handler either raises error or provides default handling. This guarantees that there is at least a handler in the system that can handle the request from requester processes and hence provide completeness of the system.

Intensional receive can be used to implement this pattern in message passing systems and also ensures completeness of the system. By completeness, we mean that there is at least one process in the system that can handle the sent message of a particular type.

```
1  (let hndlr1 =
2   (spawn (next : P)
3    (receive (m1 : P)
4     (set m1)
5     (receive (m2 : bool * P)
6      (send (snd m2)
7            (neg (fst m2)))
8      (receive (f : Top)
9       (send get f)
10     )
11    )
12   )
13  ) in
14  (let hndlr2 =
15   (spawn (s : nat)
16    (receive (m3 : nat * P)
17     (send (snd m3)
18           (double (fst m3))
19    )
20   ) in
21   (let rqst =
22    (spawn (r : bool * nat)
23     (receive (m4 : nat)
24       (set (pair (fst get) m4)
25       )
26       (receive (m5: P)
27        (send m5
28              (pair (fst get)
29                    self)) ;
30        (send m5
31              (pair (snd get)
32                    self)) ;
33        (receive (b : bool)
34         (receive (c : nat)
35          (set (pair b c)
36          )
37         )
38       )
39      )
40     ) in
41     (send rqst 2) ;
42     (send rqst hndlr1) ;
43     (send hndlr1 hndlr2)
44    )
45   )
46 )
```

Figure 2.3 An example of the receive expression representing chain of responsibility design pattern.

In the example presented in Figure 2.3, there is a requester `rqst` and two other processes in the system `hndlr1` and `hndlr2`. `rqst` is sending two messages of type `bool * P` and `nat * P` to

www.manaraa.com

hndlr1 at line 27-32. hndlr1 is capable of handling the message of type bool * P and hence it performs computation on it and sends back the result to the rqst as we could see at lines 5, 6 and 7. But hndlr1 cannot handle the message of type nat * P and hence forward the request to hndlr2 as we could see at line 9. hndlr2 performs the computation on message of type nat * P and sends back the result to rqst at lines 17 and 18. By following the chain of intensional receive expression, a programmer can inspect the system for completeness. We revisit this example in §5.3 where we discuss checking completeness.

2.4 Multiplexing

In this section, we demonstrate how intensional design of receive expression helps us in reasoning about the effects and types of multiple messages sent to a single process by different sources. In another mechanism to deal with multiple messages the messages cannot be statically type checked. Our intensional design of receive expression help us to statically type check different type of messages sent by different sources to a single process. We illustrate an example in the Figure 2.4.

```
1 (let mult =
2  (spawn (addr : P)
3   (fix (receive (m1 : P)
4         (set m1)
5         (receive (m2 : nat)
6          (receive (m3 : bool)
7           (send (get)
8                 (pair m2 m3))
9           (receive (m4 :
10                    bool)
11            (receive (m5 :
12                     nat)
13             (send (get)
14                   (pair m5
15                         m4))
16            )
17           )
18          )
19         )
20        )
21   )
22  ) in
23  (let output =
24   (spawn (a:nat*bool)
25    (receive (m6:nat*bool)
26     (set m6)
27    )
28   ) in
29   (send mult output);
30   (send mult 4) ;
31   (send mult false)
32  )
33 )
```

Figure 2.4 An example of the receive expression supporting multiplexing.

www.manaraa.com

There is a process `mult` which forwards the pair of message received from different sources to process `output` where the first element of the pair is always a natural and the second element is always a bool irrespective of in which order message is received. In the body of the process `mult`, it multiplexes the input provided to it in line 30 and 31 as pair of `nat` and `bool` and forwards it to process `output` as we could see in the line 13, 14 and 15. Intensional receive expression help us to statically type check the messages from different sources and always keeps the effect same.

المنارة للاستشارات

www.manaraa.com

CHAPTER 3. DESIGN OF INTENSIONAL RECEIVE: ABSTRACT SYNTAX

This and the next four chapters describe the technical underpinnings of intensional receive. While intensional receive could also augment process calculi in the flavor of CSP [17] or Π calculus [23], we chose to built an expression-based calculus that we call λ_{ir}. λ_{ir} extends the simply-type lambda calculus in a style similar to [10]. This combination is arguably closer to the adoption of message-passing concurrency in practice [13, 5, 11, 27, 22], while remaining small enough to highlight the foundation nature of intensional receive.

$e ::=$	Exp	$T ::=$	Value Types
$\mid \boldsymbol{send}\ e\ e'$	Send	$\mid T \xrightarrow{C} T$	Arrow
$\mid \boldsymbol{spawn}\ x : T\ e$	Spawn	$\mid P(T\ \ C)$	Process
$\mid \boldsymbol{receive}\ x : T\ e\ e'$	Receive	$\mid unit$	Unit
$\mid \boldsymbol{self}$	Self	$\mid Top$	Top
$\mid \boldsymbol{set}\ e$	Asgn		
$\mid \boldsymbol{get}$	Read	$C ::=$	Comm Types
$\mid x : T\ .\ e$	Abs	$\mid 0$	Null
$\mid e\ e$	App	$\mid C_1\ \&\ C_2$	Choice
$\mid x$	Variable	$\mid C_1 :: \ldots :: C_n$	Seq
$\mid \boldsymbol{unit}$	Unit	$\mid ![T]$	Send
$\mid \boldsymbol{fix}\ e$	Fix	$\mid ?[T]$	Receive

Figure 3.1 λ_{ir} Abstract Syntax. (In this thesis, Notation $\bar{\bullet}$ Represents a Set of $\bullet$ Elements.)

Figure 3.1 shows the expression-based core syntax and set of types, which includes both value types and communication types. We borrow standard terms and types from the simply typed lambda calculus. To that, we add new expressions for sending a message (**send**), spawning a new process (**spawn**) and receiving a message (**receive**). An expression of the form *send* $e\ e'$ sends a message represented by the e' to the process represented by the e. We also have support for

المنارة للاستشارات
www.manaraa.com

recursive functions (**fix**) and λ_{ir} also has two nullary expressions for retrieving the identity of a process (**self**). Our examples also freely use let, if, booleans, and numbers that have the standard desugaring.

The typing for unit values and functions is standard, except function types also encode information about "*latent*" communication that can be performed if that function is evaluated. λ_{ir} distinguishes between value types T and communication types C. We write $T \xrightarrow{0} T$ as $T \rightarrow T$ when the use is unambiguous. We discuss types in more detail when presenting the static semantics of λ_{ir} in §5.

3.1 Single-state Processes

Somewhat non-traditionally, each process in λ_{ir} can have a single state that can be read (using **get**) and written (using **set**). Supporting multiple states can be encoded using records and pairs without causing fundamental difficulties.

An expression of the form *spawn* $x : T\ e$ creates a new process with state x of type T and e as the body of the new process. The body e can use the name x to refer to its own state, but other processes may not access the state.

The type of a process is a 2-tuple that represents the type of the value produced by the body of the process and the communication type of the process (Comm Types).

3.2 Intensional Design of the Receive Expression

A typical message passing calculus includes a nullary receive expression of the form *receive* that retrieves a message from the mailbox such that the value of the expression is the value of the received message. While this design is flexible, the continuation of the receive expression is well typed only if the sender sends a compatible message. A second design of the receive expression would expect incoming messages to be of a certain type T and the program would fail to type check if there are message sends of an incompatible type. This second design has the advantage of static reasoning. However, the increasing use of dynamic features such as reflection, dynamic linking/loading, native

المنارة للاستشارات
www.manaraa.com

```
1 (spawn x : unit
2  (receive (y : unit)
3   (set y)
4   (receive (z : unit -> unit)
5    (set z get)
6    (stop)
7   )
8  )
9 )
```

Figure 3.2 An example of the receive expression performing different tasks based on the type of the message. Note expressions on lines 3 and 5 that treat messages from the same mailbox differently.

code interfaces, and meta-programming in practice makes it increasing difficult to carry out this static reasoning precisely [21].

In λ_{ir}, building on the large body of work combining static checking and dynamic checking (for instance [9, 31, 12, 21]), and inspired by [14], we incorporate a design of the receive expression that integrates static and dynamic reasoning.

An expression of the form $receive\ x : T\ e\ e'$ checks the dynamic type of the current message in the mailbox. If the type of the message received matches the type T the message is retrieved bound to x and the expression reduces to e. Otherwise, the expression reduces to e' without retrieving the message from the mailbox. An example appears in Figure 3.2.

This design provides two benefits. First, the body of the process lends itself to static reasoning independent of the sender processes in the system. For example, expressions on lines 3 and 5 in Figure 3.2 can be statically checked regardless of the sender processes. Second, the use of dynamic typing provides more precise reasoning. In the examples we also freely use syntactic sugar for

```
1    receive x : T1 e1
2    T2 e2
3    T3 e3
4    e4
```

```
1    receive x : T1 e1
2    receive x : T2 e2
3    receive x : T3 e3
4    e4
```

Figure 3.3 Syntactic sugar for multiple receive and its desugaring

المنارة للاستشارات
www.manaraa.com

receive with multiple types as shown in Figure 3.3 (left) whose desugaring is shown on the right in Figure 3.3.

www.manaraa.com

CHAPTER 4. DYNAMIC SEMANTICS OF λ_{ir} PROGRAMS

We define λ_{ir}'s operational (dynamic) semantics as a small-step semantics. The definitions used by the operational semantics are defined in Figure 4.1. In λ_{ir}'s operational semantics each concurrently running process instance owns its state and mailbox which stores messages sent to it and uses only one thread of execution to execute its body. Two processes can only communicate through sending messages to each other. The state of a process can be only accessed and changed by the process itself.

4.1 Dynamic Objects

The operational semantics of λ_{ir} transition from one global (program) configuration to another is shown in Figure 4.2. A global configuration $\mathscr{P}$ is a concurrent composition $\parallel$ of process instance configuration $\sum$. Concurrent composition $\parallel$ is commutative, i.e. $\sum \parallel \sum'$ is equal to $\sum' \parallel \sum$.

Evaluation contexts :

$\mathscr{E}$ ::= $\bullet$ | **send** e $\mathscr{E}$ | **send** $\mathscr{E}$ v | **set** $\mathscr{E}$ | $\mathscr{E}$ e | v $\mathscr{E}$

Domains :

$\mathscr{P}$ ::= $\bullet$ | $\sum \parallel \mathscr{P}$	Global config
$\sum$ ::= $\langle st, e, M\rangle_{id}$	Local config
st ::= $T\ x\ v$	State
M ::= $\bullet$ | $v.M$	Mailbox
v ::=	Values
| $x : T . e$	Abs
| *unit*	Unit
| *process* id	Process

a ::=	Actions
| *send (v, id, id′)*	Send
| *receive (v, T, id)*	Receive
| *spawn (id, id′)*	Spawn
| *get (id)*	Get
| *set (id)*	Set
| *local (id)*	Local
| *self*	Self

Figure 4.1 Definitions used in λ_{ir}'s Dynamic Semantics

المنارة للاستشارات
www.manaraa.com

A process configuration $\sum$ consists of a unique process identifier *id*, and the state declared in the process *st*, expression to be evaluated in the body of the process, and a mailbox. The mailbox stores the message sent to a process and messages are received in sequential order from the mailbox. *st* contains a type of the state, the name of the state, and the value it provides.

In λ_{ir}, a value can be an abstraction, the unit expression, or a process value.

The execution of a λ_{ir} program produces a trace of observable actions. Actions are basic units of execution and each action represents execution of a single *indivisible* (atomic) instruction. Figure 4.1 shows a core set of actions observed during the execution of a λ_{ir} program. An action can be: send a message v from the process instance id to process instance id', receive a message v of type T, create a new process instance id', get the value of state by the process instance id, set the value of state by the process instance id, to get its own address and the local action observed during an execution.

4.2 Local and Global Semantics

The operational semantics of λ_{ir} consists of two sets of evaluation rules which includes its local and global semantics. A local evaluation $\overset{a}{\rightsquigarrow}$ denotes a transition from a process configuration to another performing the action a. A local transition in turn causes a global transition $\overset{a}{\hookrightarrow}$ from one program configuration to another in which processes run concurrently. A process instance is chosen nondeterministically by a *preemptive* scheduler in the program configuration for evaluation at each point of time. The dynamic semantic rules are of the form $\mathscr{P} \overset{a}{\hookrightarrow} \mathscr{P}'$, to be read as "program $\mathscr{P}$ reduces to program $\mathscr{P}'$" or "program $\mathscr{P}$ takes a step to program $\mathscr{P}'$." The reduction rules are given in Figure 4.2.

4.3 Sequential Synchronous Local Semantics

Local evaluation relation $\overset{a}{\rightsquigarrow}$ within the context of a process instance denotes evaluation of an expression e at the body of the process to another expression e' and performing the action a. This evaluation causes the transition from a process configuration to another with a (potentially)

www.manaraa.com

Local evaluation rule $\overset{a}{\leadsto}$**:** $\langle st, \mathscr{E}[e], M\rangle_{id} \;||\; \mathscr{P} \overset{a}{\leadsto} \langle st', \mathscr{E}[e'], M'\rangle_{id} \;||\; \mathscr{P}$

$$(Asgn)\frac{}{\langle T\ x\ v, \mathscr{E}[\textbf{set}\ v'], M\rangle_{id} \,||\, \mathscr{P} \overset{set(id)}{\leadsto} \langle T\ x\ v', \mathscr{E}[v'], M\rangle_{id} \,||\, \mathscr{P}}$$

$$(Read)\frac{}{\langle T\ x\ v, \mathscr{E}[\textbf{get}], M\rangle_{id} \,||\, \mathscr{P} \overset{get(id)}{\leadsto} \langle T\ x\ v, \mathscr{E}[v], M\rangle_{id} \,||\, \mathscr{P}}$$

$$(ReceiveT)\frac{\Gamma \vdash v : T,\ 0}{\langle st, \mathscr{E}[\textbf{receive}\ x : T\ e\ e'], v.M\rangle_{id} \,||\, \mathscr{P} \overset{receive(v,T,id)}{\leadsto} \langle st, \mathscr{E}[[v/x]e], M\rangle_{id} \,||\, \mathscr{P}}$$

$$(ReceiveF)\frac{\Gamma \vdash v : T',\ 0}{\langle st, \mathscr{E}[\textbf{receive}\ x : T\ e\ e'], v.M\rangle_{id} \,||\, \mathscr{P} \overset{receive(v,T',id)}{\leadsto} \langle st, \mathscr{E}[e'], v.M\rangle_{id} \,||\, \mathscr{P}}$$

$$(Self)\frac{}{\langle st, \mathscr{E}[\textbf{self}], M\rangle_{id} \,||\, \mathscr{P} \overset{self}{\hookrightarrow} \langle st, \mathscr{E}[id], M\rangle_{id} \,||\, \mathscr{P}}$$

$$(AppAbs)\frac{}{\langle st, \mathscr{E}[x : T.e\ \ v], M\rangle_{id} || \mathscr{P} \overset{local(id)}{\leadsto} \langle st, \mathscr{E}[[v/x]\ e], M\rangle_{id} || \mathscr{P}}$$

$$(Fix)\frac{}{\langle st, \mathscr{E}[\textbf{fix}\ x : T\ e], M\rangle_{id} \,||\, \mathscr{P} \overset{local(id)}{\hookrightarrow} \langle st, \mathscr{E}[[\text{fix}\ x :\ T\ e/x]e], M\rangle_{id} \,||\, \mathscr{P}}$$

Global evaluation rule $\overset{a}{\hookrightarrow}$**:** $\langle st, \mathscr{E}[e], M\rangle_{id} \;||\; \mathscr{P} \overset{a}{\hookrightarrow} \langle st', \mathscr{E}[e'], M'\rangle_{id} \;||\; \mathscr{P}'$

$$(Send)\frac{\langle st', e', M'\rangle_{id'} \in \mathscr{P} \qquad \mathscr{P}' = \langle st', e', M'.v'\rangle_{id'} \uplus \mathscr{P}}{\langle st, \mathscr{E}[\textbf{send}\ id'\ v'], M\rangle_{id} \,||\, \mathscr{P} \overset{send(v',id,id')}{\hookrightarrow} \langle st, \mathscr{E}[v'], M\rangle_{id} \,||\, \mathscr{P}'}$$

$$(Spawn)\frac{\begin{array}{c} fresh(id', \langle st, \mathscr{E}[\textbf{spawn}\ x\ T\ e], M\rangle_{id} \,||\, \mathscr{P}) \qquad v' = default(T) \\ \mathscr{P}' = \langle T\ x\ v', e, \bullet\rangle_{id'} \,||\, \mathscr{P} \end{array}}{\langle st, \mathscr{E}[\textbf{spawn}\ x\ T\ e], M\rangle_{id} \,||\, \mathscr{P} \overset{spawn(id,id')}{\hookrightarrow} \langle st, \mathscr{E}[id'], M\rangle_{id} \,||\, \mathscr{P}'}$$

$$(Congruence)\frac{\langle st, \mathscr{E}[e], M\rangle_{id} \overset{a}{\leadsto} \langle st', \mathscr{E}[e'], M'\rangle_{id}}{\langle st, \mathscr{E}[e], M\rangle_{id} \,||\, \mathscr{P} \overset{a}{\hookrightarrow} \langle st', \mathscr{E}[e'], M'\rangle_{id} \,||\, \mathscr{P}'}$$

Figure 4.2 Operational Semantics

modified mailbox and updated state. In local-semantics, a process instance can access its state,

المنارة للاستشارات
www.manaraa.com

update the value of its state, and receive a message from its mailbox. The rules for state assignment, state read, and self are self-explanatory and presented in Figure 4.2.

The two rules of receive *(ReceiveT)* and *(ReceiveF)* model the cases where the top of the mailbox contains a matching message, and otherwise. The rule looks up the dynamic type of the message, and performs substitution in e in the former case, and reduces to e' in the latter.

All of the rules record actions. The set expression performs a *set (id)* action. The get expression performs a *get (id)* action, the receive expression performs a *receive (v, T, id)* action and the self-expression performs a *self* action.

4.4 Concurrent Asynchronous Global Semantics

Global evaluation $\overset{a}{\hookrightarrow}$ denotes concurrent evaluation of process instances. The rule *(Congruence)* plays the role of a preemptive scheduler that chooses a process instance id in global configuration $\mathscr{P}$ nondeterministically to take an atomic action at each point in time, according to the operational semantic rules.

A process can send a message to another process through the send expression. A send expression represented as *send* id' v' where the id' represents the address of the process to which message is sent and the second expression represents the content of the message. The evaluation of send expression appends the message v' to the end of the mailbox of the receiver of the message. The notation $\langle st', e', M'.v' \rangle_{id'} \uplus \mathscr{P}$ denotes overriding the process configuration of process represented by the address id', where $\uplus$ is an overriding union operation. The send expression performs a *send* (v', id, id') action.

A process can create another new process to carry the rest of its computation or perform some subtasks. A new process is created using *spawn* x T e where e is the body of the process, and x is the state in the process with type T. After the evaluation of *spawn* x T e, a new process configuration is added to the set of configurations. The address of the new process should be a fresh address. The value v' is the value assigned to the state x which is a default value and which has the same type as the type of state. The auxiliary function *default* produces a default value based on the type of

المنارة للاستشارات
www.manaraa.com

the process's state. The notation $\langle T\ x\ v', e, M' \rangle_{id'} \ || \ \mathscr{P}$ denotes appending $\langle T\ x\ v', e, M' \rangle_{id'}$ with the concurrent composition. The spawn expression performs a *spawn (id, id′)* action.

www.manaraa.com

CHAPTER 5. STATIC SEMANTICS OF λ_{ir} PROGRAMS

We now discuss the type system of λ_{ir} that ranges over expressions and types defined in Figure 3.1.

The typing judgment $\Gamma \vdash e : (T, C)$ says that in the typing environment Γ, expression e has value type T and communication type C which describe an upper bound on the communication performed during the reduction of the expression. The idea of communication types is inspired from [18] which calls them "process types", whereas our communication types describe the communication behavior of both expressions and processes (thus the general name). The set of value types and communication types are defined in the Figure 3.1. 0 is the type of the null process. The type C_1 & C_2 represents an internal choice irrespective of what communication are provided by the environment. $C_1 :: ... :: C_n$ represents the sequence of the communication types $C_1, ... , C_n$. $?[T]$ represents the receive communication type and $![T]$ represents the send communication type. The value types are basic simple lambda types except that function types also encode the information about the latent communication that could be produced due to the function evaluation. Also, we include a process type that includes the value type and communication type of the body of the process which helps us reason about the process.

5.1 Computing Potential Behavior of a Recipient

From the communication type of a process, we know whether it is capable of receiving a message of a certain type. However, statically we do not know which state the recipient will be at run time when the message is sent. Depending on the state of the recipient, it can exhibit different communication behavior.

To illustrate this, consider the example in Figure 5.1 that has two processes `a` and `b`. Process `a` is triggered by sending it `b`'s address and a number `5` at line 20. We are reasoning about the

المنارة للاستشارات
www.manaraa.com

```
1 (let a =
2  (spawn (s : nat)
3   (receive (y : P * nat)
4    (if (gt (snd y) (0))
5     then (receive (x :
6                     P * nat)
7           (if (gt (snd x)
8                    (snd y))
9            then (set
10                 (snd x))
11            else (set
12                  (snd y)))
13           (x)
14          )
15      else y)
16     (send b
17           (pair (self) 2)
18   )
19  ) in
20  (let b =
21   (spawn (s : nat)
22    (receive (y : P * nat)
23     (set (snd y))
24    )
25   ) in
26   (send (a) (pair b 5))
27  )
28 )
```

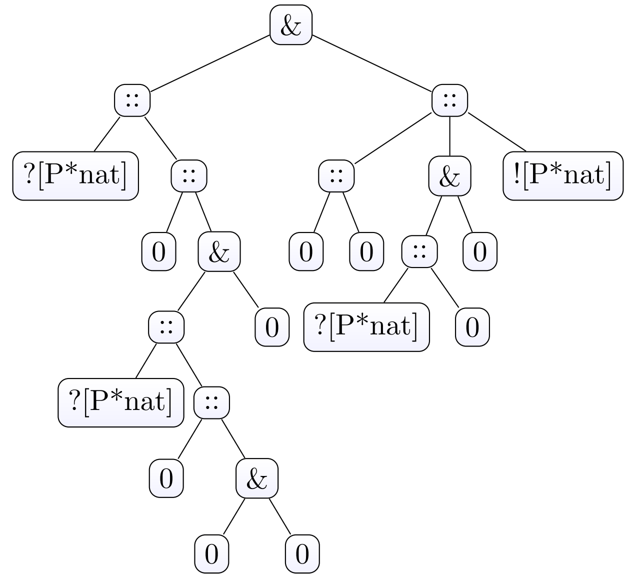

Figure 5.1 Example showing the communication type of two interacting processes

المنارة للاستشارات
www.manaraa.com

communication behavior of process `a`. The communication type of process `a` is represented by an abstract syntax tree (AST) in Figure 5.1 (right).

The body of the process `a` has a receive expression which receives a message of type `P * nat` and evaluates a conditional expression or sends a message to the process `b`. The receive type `?[P * nat]` must be present in the communication type of the process `b` in order for these processes to be compatible.

The AST of the communication type shows that process `a` contains multiple states where it can receive a message of type `?[P * nat]` (e.g., lines 3 and 4 in the code). But the process `a` can be in any of those states. Furthermore, process `a` will exhibit a different communication behavior when it is in either of those states. For example, process `a` can either receive a message of type `?[P * nat]` and exhibit no communication behavior (0) (right most subtree). Alternatively, it can receive a message of type `?[P * nat]` and then possibly be ready to receive another message of type `?[P * nat]` (left most subtree). λ_{ir}'s static semantics models this possibility by composing the subtree starting with `?[P * nat]` nodes using a non-deterministic choice operator $\ominus$ (details in §5.5).

5.2 Pipelining (revisited)

As we presented an example in Figure 2.1 the components connected in pipeline structure should be compatible with each other in terms of message being sent and consumed. From the communication type of a process, we know whether a process is capable of receiving a message of a certain type. Figure 2.1 shows that `initstg` is sending a message of type `![nat * nat]` to `midstg` and `midstg` is sending message of type `nat` to `finstg`. So from the communication type of processes, we can argue that sender and receiver are compatible with each other or not. Lets look at the AST representing communication type of `initstg`, `midstg` and `finstg` in the Figure 5.2 and Figure 5.3. As we could see AST representing communication type of `initstg` has a state where it can send message of type `![nat * nat]`, which means AST representing communication type of `midstg` must have a state where it can receive message of type `?[nat * nat]` to guarantee

المنارة للاستشارات
www.manaraa.com

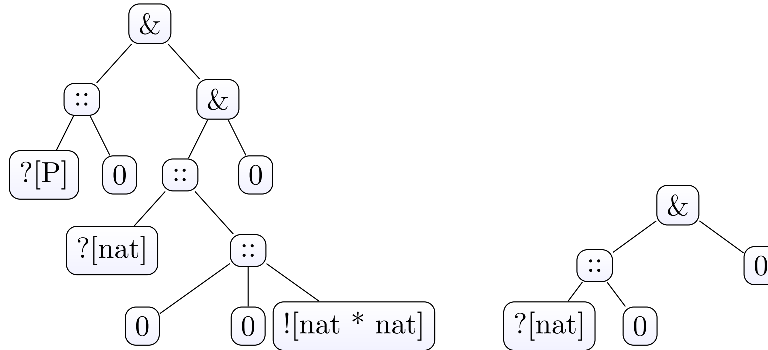

Figure 5.2 Communication type of initstg and finstg

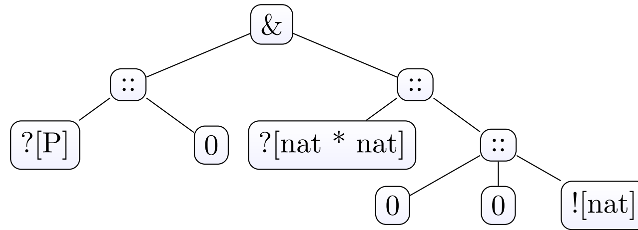

Figure 5.3 Communication type of midstg

compatibility between them. We could see in the Figure 5.3, there is node representing the state where it can receive message of type `?[nat * nat]`. Also in AST representing communication type of `midstg`, we could see it can be in a state where it can send message of type `![nat]`. For `midstg` and `finstg` to be compatible, AST representing communication type of `finstg` must have a state where it can receive message of type `?[nat]`. We could see in the right side of Figure 5.2 which represents communication type of `finstg`, there is node representing the state where it can receive message of type `?[nat]`. Hence `midstg` and `finstg` are compatible with each other. Hence from communication type, we can argue about the compatibility between various stages which could not be possible without the intensional design of receive. Because of the design of intensional receive, we have incorporated types in receive and send communication types.

www.manaraa.com

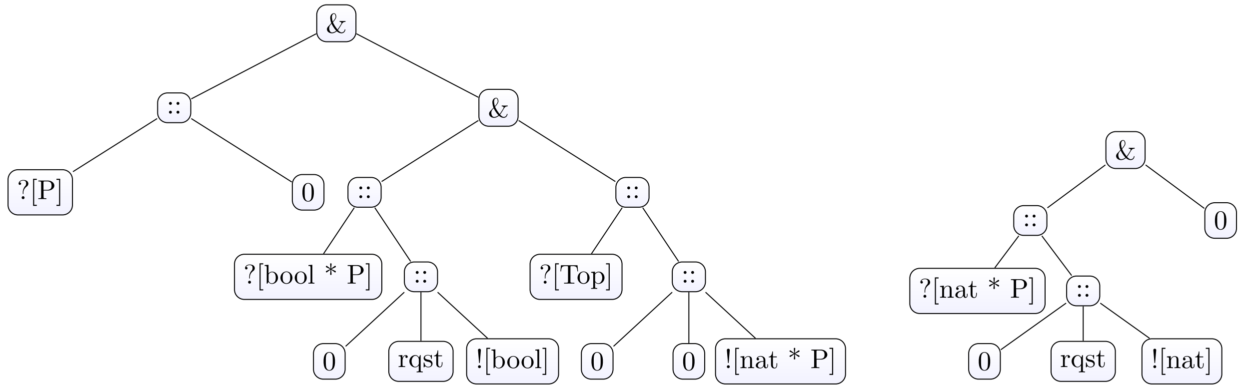

Figure 5.4 Communication type of hndlr1 and hndlr2

5.3 Chain of responsibility design pattern (revisited)

In the example in Figure 2.3, we claim that intensional receive helps us to prove the completeness of the system. From the communication type of the handlers and requesters, we could argue that if there is a send communication type of a particular type in the system then there must be a receive communication type of that particular type in the system. As we could see in the AST representing communication type of `rqst` in Figure 5.5, it can be in states where it can send message of type `![bool * P]` and `![nat * P]`. Also we could see in the AST representing communication type of `hndlr1` and `hndlr2` in Figure 5.4, `hndlr1` can be in a state where it can receive message of `?[bool * P]`. Also as `hndlr1` is not capable of handling message of type `nat * P`, hence it forwards the request to `hndlr2`. In the AST representing communication type of `hndlr1` in the left side of Figure 5.4, it can be in a state where it can send a message of type `![nat * P]`. In the AST representing communication type of `hndlr2` in the right side of Figure 5.4, it can be in a state where it can receive a message of type `?[nat * P]`. So we could see that both the type of messages sent by the `rqst` are handled by at-least one handler. Hence from the communication type of all the processes in the system, we can prove completeness of the system by showing that there exists at-least a receiver which is capable of handling message of any particular type.

www.manaraa.com

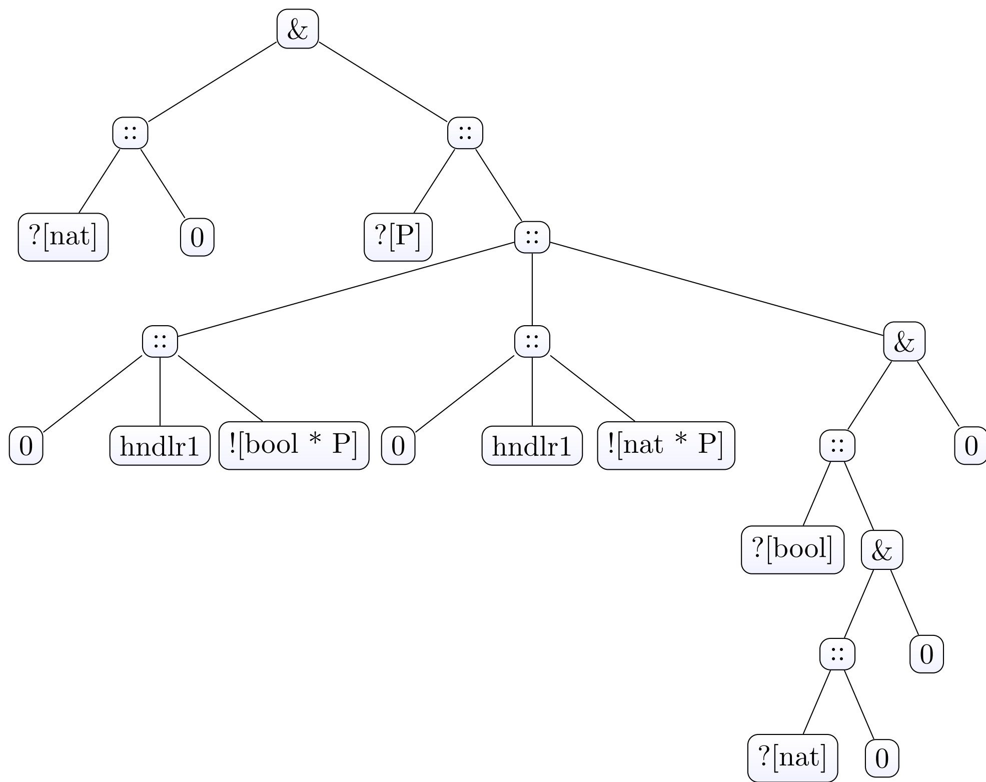

Figure 5.5 Communication type of rqst

المنارة للاستشارات
www.manaraa.com

Subtyping for Value Types: T $\preceq T'$

$$(\textit{Reflexive})\frac{}{T \preceq T} \qquad (\textit{Transitive})\frac{T \preceq T' \quad T' \preceq T''}{T \preceq T''}$$

$$(\textit{Arrow})\frac{T_2 \preceq T_1 \quad T_1' \preceq T_2' \quad C \preceq C'}{T_1 \xrightarrow{C} T_1' \preceq T_2 \xrightarrow{C'} T_2'} \qquad (\textit{Top})\frac{}{T \preceq \textit{Top}}$$

Figure 5.6 Subtyping relations for value types.

5.4 Subtyping, Value, and Communication Types

The sub-typing relations between the types are shown in Figure 5.6 and Figure 5.7. The subtyping for value types is standard except subtyping relation for function types is augmented to state subtyping requirements for latent communication type.

The subtyping relations for communication types is interesting. The basic intuition behind these rules is that a super-communication-type defines an upper bound on the communication behavior, and the sub-communication-type also defines an upper bound on the communication behavior that is contained within the upper bound defined by the super-communication-type.

Subtyping for Communication Types: C $\preceq C'$

$$(\textit{Reflexive})\frac{}{C \preceq C} \qquad (\textit{Transitive})\frac{C \preceq C' \quad C' \preceq C''}{C \preceq C''}$$

$$(\textit{Internal Choice})\frac{C_i \preceq C_i'}{C_1 \ \&\ C_2 \preceq C_i' \ (i \in \{1,2\})} \qquad (\textit{Send})\frac{T \preceq T'}{![T] \preceq ![T']}$$

$$(\textit{Receive})\frac{T \preceq T'}{?[T] \preceq ?[T']} \qquad (\textit{Seq})\frac{C_i \preceq C_i' \textit{ for each } i \in \{1, \ldots, n\}}{C_1 :: \ldots :: C_n \preceq C_1' :: \ldots :: C_n'}$$

$$(\textit{Seq})\frac{C_i \in C_1' :: \ldots :: C_n'}{C_i \preceq C_1' :: \ldots :: C_n'}$$

Figure 5.7 Subtyping relations for communication types.

المنارة للاستشارات
www.manaraa.com

The subtyping relations for send and receive are devised to avoid surprising the recipient.

5.5 Type Rules

Type Checking: $\Gamma \vdash$ e : T, C

$$(Send)\frac{\Gamma \vdash e' : T',\ C' \qquad \Gamma \vdash e : P\ (T\ C),\ C'' \qquad ?[T'] \in C \qquad \ominus(?[T'], C) = C'''}{\Gamma \vdash \textbf{send}\ e\ e' : T',\ C' :: C'' :: ![T'] :: C'''}$$

$$(Receive)\frac{\Gamma\ ,\ x : T \vdash e : T',\ C' \qquad \Gamma \vdash e' : T',\ C''}{\Gamma \vdash \textbf{receive}\ x : T\ e\ e' : T',\ ?[T] :: C'\ \&\ C''}$$

$$(Spawn)\frac{\Gamma,\ self_st : T,\ self_id : P(T'\ C') \vdash e : T',\ C'}{\Gamma \vdash \textbf{spawn}\ x : T\ e : P(T'\ C'),\ 0} \qquad (Self)\frac{\Gamma(self_id) = P(T\ C),\ 0}{\Gamma \vdash \textbf{self} : P(T\ C),\ 0}$$

$$(Set)\frac{\Gamma \vdash e : T',\ C' \qquad \Gamma \vdash self_st : T,\ 0 \qquad T' \preceq T}{\Gamma \vdash \textbf{set}\ e : T,\ C'} \qquad (Get)\frac{\Gamma \vdash self_st : T,\ 0}{\Gamma \vdash \textbf{get} : T,\ 0}$$

$$(Abs)\frac{\Gamma\ ,\ x : T \vdash e : T',\ C}{\Gamma \vdash x : T\ e : T \xrightarrow{C} T',\ 0} \qquad (App)\frac{\Gamma \vdash e : T \xrightarrow{C''} T',\ C \qquad \Gamma \vdash e' : T,\ C'}{\Gamma \vdash e\ e' : T',\ C :: C' :: C''}$$

$$(Fix)\frac{\Gamma \vdash e : T \xrightarrow{C} T,\ 0}{\Gamma \vdash \textbf{fix}\ e : T,\ C} \qquad (Subsume)\frac{\Gamma \vdash e : T',\ C \qquad T' \preceq T}{\Gamma \vdash e : T,\ C}$$

$$(Subsume)\frac{\Gamma \vdash e : T,\ C' \qquad C' \preceq C}{\Gamma \vdash e : T,\ C}$$

Figure 5.8 Typing Rules for λ_{ir}.

Figure 5.8 shows the typing rules for expressions of λ_{ir} which assigns the value type and communication type to each expression in the typing context Γ.

Intensional design of receive expression help us to gain compatibility between the sender and receiver in terms of their flow of communication which we explain further by describing our typing rules.

المنارة للاستشارات

www.manaraa.com

$$\ominus(?[T], C) = \begin{cases} 0 & \textit{if } C = 0, \\ 0 & \textit{if } C = ![T], \\ ?[T] & \textit{if } C = ?[T], \\ \ominus(?[T], C_1) \ \& \ominus (?[T], C_2) & \textit{if } C = C_1 \ \& \ C_2, \\ \ominus(?[T], C_1) \ \& \ \ldots \ \& \ominus (?[T], C_n) & \textit{if } C = C_1 \ :: \ \ldots \ :: \ C_n \end{cases}$$

$$\frac{\langle st, e, M \rangle_{id} \notin \mathscr{P}}{fresh(id, \mathscr{P})}$$

Figure 5.9 Auxiliary Functions

The type rule for the send expression is most interesting because it involves both verifying compatibility between the sender and the receiver and computing the potential behavior of the receiver in response to receiving a message. This rule type checks the send expression by ensuring (a) the message e' is well-typed and has value type T' and reducing e' could lead to communication behavior as defined by the communication type C' in the typing context Γ, and (b) expression e which evaluates to the identity of the recipient process is well-typed and has value type $P(T\ C)$ where T is the value type of the body of the recipient process and C is the communication type of the body of the recipient process and reducing e could lead to the communication behavior represented by communication type C'' in the typing context Γ. Hence the send expression represented as *send* $e\ e'$ has value type T' and communication type $C' :: C'' :: ![T'] :: C'''$ where receive communication type $?[T']$ should be present in C. This ensures a compatibility between the receiver and the message being sent.

When a message is sent, the recipient could be in one of the potentially many states (see §5.1). Given a receive type $?[T]$ and a communication type, the function $\ominus$ creates a composite communication type that represents a non-deterministic composition of the communication behavior starting with the states that can receive the message of type T.

The rule for receive expression models that upon receiving a message of type T the process would exhibit both the receiving behavior as well as the communication behavior exhibited by the expression e, whereas upon receiving a message that is not of type T the process would only exhibit the communication behavior of the expression e'.

www.manaraa.com

The rule for spawn expression produces the process type that encodes both the value type produced by the body of that process and its entire communication behavior. Spawning a process itself doesn't lead to any communication. Creating a function value records the communication type in the function type, and applying a function exhibits the communication behavior or evaluating the argument, the function, and the latent communication behavior.

المنارة للاستشارات
www.manaraa.com

CHAPTER 6. INTENSIONAL RECEIVE GENERATES NEW HAPPENS-BEFORE RELATIONS

Our intensional receive expression helps us produce a new *happens-before* relation which help us in proving correctness of the system. A system is correct if it meets its specification. If a system is expecting to receive messages in an order of certain types, to prove the correctness of the system we need to argue about the receive actions caused by the execution of the system. As we could see due to our intensional design of receive expression, receive action incorporates type denoted by T. This addition helps us state that there can be ordering between various receive actions in the trace which can be utilized to argue about the order of message types that are received by any program. The new *happens-before* relations are produced by arguing about the incorporated types in the receive action which is the direct impact of our intensional design of receive. Also we can see that incorporated type in the receive action makes the receiver action a non-mover action [20]. Any receive action incorporating type T cannot be switched with either its left or right actions because that will not be compatible with the corresponding intensional receive expressions which is expecting to receive message of type T before or after any other type T'. Complete proof of this property is presented in the Appendix §A.3.

6.1 Actions: Conflict and Happens-Before Relations

Evaluation of a message passing program, with its nondeterministic preemptive scheduler, results in a trace of interleaved actions (defined in Figure 4.1), performed by different process instances of the program. To build the core of λ_{ir}, we define the execution trace of a message passing program, define the conflict and happens-before relations, and prove the mover properties of the set of actions using Lipton's reduction theory [20]. While definitions and detailed proof of λ_{ir}'s mover properties are presented in §**??**, we discuss the essence below.

www.manaraa.com

Conflicting actions # and their *happens-before* relation: $\prec$

$$\frac{}{receive(v,\ T,\ id')\ \#\ receive(v,\ T',\ id')} \qquad \frac{\langle st, \mathscr{E}[\textbf{\textit{receive}}\ x : T\ e\ e'], v.M\rangle_{id'} \in \mathscr{P}}{receive(v,\ T,\ id')\ \#\ send(v,\ id,\ id')}$$

$$\frac{\langle st, \mathscr{E}[\textbf{\textit{receive}}\ x : T\ e\ e'], v.M\rangle_{id'} \in \mathscr{P}}{send(v,\ id,\ id')\ \prec receive(v,\ T,\ id')} \qquad \frac{send(v,\ id,\ id') \prec send(v',\ id,\ id')}{receive(v,\ T,\ id') \prec receive(v',\ T',\ id')}$$

$$\frac{\begin{array}{c} send(v, id, id_1) \prec send(v_1, id, id_2) \wedge\ send(v_1, id, id_2) \prec send(v_2, id_2, id_3)\ \ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots \prec send(v_n, id_{n-1}, id_n) \prec send(v', id_n, id_1) \end{array}}{receive(v, T_1, id_1)\ \prec\ receive(v', T_n, id_n)}$$

Figure 6.1 Conflicting actions in the Message Passing Model based on type of message received.

```
1  (let VM =
2   (spawn (s : nat)
3    (fix (receive (x : nat * P)
4      (receive (c :
5                 red * P)
6       (send (snd c)
7             (red-drink))
8       (receive (c :
9                  orange *
10                 P)
11       (send (snd c)
12             (orange-drink))
13      )
14     )
15    )
16   )
17  ) in
18  (let C =
19   (spawn (a : P)
20    (receive (addr : P)
21     (set addr)
22   (receive (coin : nat)
23     (receive (type : red)
24      (send get
25          (pair coin self)) ;
26      (send get
27          (pair type self));
28      (receive (d1 :
29                     red-drink)
30       (unit)
31       (receive (d2 :
32                      orange-drink)
33          (unit)
34         )
35        )
36       )
37      )
38     )
39    ) in
40    (send C VM) ;
41    (send C 5);
42    (send C r)
43   )
44  )
```

Figure 6.2 Encoding vending machine and reasoning about its correctness.

المنارة للاستشارات
www.manaraa.com

Consider the example in Figure 6.2 where intensional receive is used to model a simple vending machine (VM) which serves customers and take a coin first and a choice of drink next as input and dispense the drink to the customer. In the example, we use descriptive type names such as red, orange, red-drink and orange-drink for clarity that can be encoded in terms of existing types in the language. Execution of the program will produce a trace of actions where there will be receive actions due to intensional receive expression at various receiving sites like lines 8, 9 and 12. There are possibilities of inefficient order in which customer operates the VM. A customer can press the choice first and then insert the coin but VM cannot operate in this order. Hence we need happens-before relation between the type of messages received by VM. Our intensional receive design help us to get new happens-before relation. The receive action produced due to execution of receive expression at lines 8 and 9 are *receive ((5, C), (nat * P), VM)* and *receive ((r, C), (red * P), VM)*. As we know according to the specification of vending machine that it accepts coin first and then the choice of drink, we can have a new *happens-before* relation which states that *receive ((5, C), (nat * P), VM)* $\prec$ *receive ((r, C), (red * P), VM)*. Also both the receive actions are non-mover and cannot be switched with each other. *receive ((5, C), (nat * P), VM)* cannot be moved to the right or left of any other receive action present in the program. *receive ((r, C), (red * P), VM)* is one of them because always message of type `nat` should be received before message of type `red`. Also by looking at the trace of the program, we can verify that vending machine meets the specifications or not. Similarly we can reason about the other case. Intensional receive help us to produce new *happens-before* relation which help us in validation and verification of the system.

6.2 Guaranteed Delivery

One of the important characteristics of message passing model is guaranteed delivery of the messages sent to a process. Processes can communicate with each other only by sending messages to each other. As explained in our dynamic semantics presented in §4, every message sent to a process instance is appended to the end of the mailbox. Whenever a receive expression is evaluated within the process, a message from the head of the mailbox is dequeued and processed depending

المنارة للاستشارات
www.manaraa.com

on the type of the message. By "message delivery", we mean that the message is dequeued from the mailbox and processed by the process.

Guaranteed delivery is stated as follows *"Every message send to a process is eventually received (processed) at some point of time"*. An expression present in a process configuration can either be a value or can make "progress" by stepping to some other expression. If expression becomes a value then it can no longer make any progress. We call a state in which there does not exists a process configuration $\mathscr{P}'$ such that process configuration $\mathscr{P}$ takes step to process configuration $\mathscr{P}'$ a *normal_form state*. We define a *missed delivery state* as a process state that cannot take any further step, is not in stuck state, and its mailbox is not empty.

Lemma 6.2.1. ***(Guaranteed delivery)*** *Let $\Sigma = \langle st, e, M \rangle_{id}$ be an arbitrary process configuration for a λ_{ir} program where the expression e is well-typed in the typing environment Γ which takes a multi step to $\Sigma' = \langle st, e', M' \rangle_{id}$ then Σ' is not a missed delivery state.*

Proof. The proof is based on the dynamic semantics and static semantics of λ_{ir} presented in Figure 4.2 and Figure 5.8. Because soundness of λ_{ir} is proved as a separate theorem we know that well-typed program in λ_{ir} can never get stuck. Semantics of send expression states that messages sent to a process are appended to the receiver's mailbox. The operational semantics of intensional receive expression dequeues the message from the mailbox if the message at the head is of expected type. The process configuration $\Sigma = \langle st, e, M \rangle_{id}$ takes multiple step to reach a configuration $\Sigma' = \langle st, e', M' \rangle_{id}$ from where no further execution is possible and it is not in stuck state. If a particular configuration is not in a stuck state and it cannot take any further step that means the expression at its body is a value. Our proof goal says that the mailbox of such configuration is empty. The proof proceeds by performing the case analysis on M'. Either M' is empty or it is of the form $v :: M''$. The first case where the mailbox is empty is a trivial case. The second case which states that M' is of the form $v :: M''$ is not true because the expression in the body of the process configuration is a value, hence there is no expression left to handle the message. The above case is never possible in our language design. Our multi-step relation covers all the possible transition of a process configuration which supports reflexive and transitivity property. For guaranteed delivery

www.manaraa.com

of a message, we look into the possible transition of process configuration in terms of receiving a message. A process agrees to receive the message until it is actually processed by it. If a message is sent to any process, it will be eventually received when the process is ready to be at the state of receiving it at some future point. Our intensional receive is designed in such a way that it prevents the process to enter into a block state. Hence the process configuration with non-empty mailbox can either take a step further which can be possible by various possibilities of multi-step relation or the mailbox is empty. □

المنارة للاستشارات
www.manaraa.com

CHAPTER 7. EXTENSIONS

We demonstrate the utility of λ_{ir} as a foundation for formalizing message passing concurrent architectures through five extensions that build on different parts of the λ_{ir} core. Two of the extensions, *broadcast* and *multicast*, show a broader use of the *send* operation of the message passing model. Another extension, *guarded receive* adds a conditional feature regulating when a *receive* operation can occur. Meanwhile, *non-blocking receive* demonstrates a different extension to the *receive* expression that allows progress to be made even if the mailbox is empty. Lastly, we extend our calculus to include the additional synchronisation primitive *wait*.

7.1 Broadcast

A process can send messages over computer networks by three different methods: unicast, multicast, and broadcast. The core calculus presented in §3 supports unicast. By the unicast method, one process can send the message to only one other process. If a process wants to send the same message to different processes, it has to use the *send* expression multiple times. Also to use the *send* expression to send messages to every other process, the process has to know the address of every other process in the program. The *broadcast* expression enables broadcasting messages to all the processes present in the program without knowing the addresses of these processes.

Figure 7.1 details an extension of λ_{mpc} with the *broadcast* feature, which sends a message represented by expression e to every process present in the program. We extend the core λ_{ir} actions by adding a *broadcast* action which are observed during the execution of *broadcast* expression. Action can be broadcasting message v by the process instance id. A process can send message v to every other process in the program through the *broadcast* expression. A *broadcast* expression is represented as *broadcast* v where v represents the message send to all other processes in the global configuration $\mathscr{P}$. The evaluation of *broadcast* expression as shown in the Figure 7.1 appends the

المنارة للاستشارات
www.manaraa.com

$e ::= \ldots$	Exp	$a ::= \ldots$	Actions
$\mid$ ***broadcast*** e	Broadcast	$\mid$ *broadcast (v, id)*	Broadcast

$$(Broadcast)\frac{\begin{array}{c}\forall \langle st_i, e_i, M_i\rangle_{id_i} \in \mathscr{P} \\ \mathscr{P}' = \langle st_1, e_1, M_1.v\rangle_{id_1} \uplus \langle st_2, e_2, M_2.v\rangle_{id_2} \uplus \ldots \uplus \langle st_n, e_n, M_n.v\rangle_{id_n} \uplus \mathscr{P}\end{array}}{\langle st, \mathscr{E}[\textbf{broadcast}\ v], M\rangle_{id} \mid\mid \mathscr{P} \overset{broadcast(v,id)}{\hookrightarrow} \langle st, \mathscr{E}[v], M\rangle_{id} \mid\mid \mathscr{P}'}$$

$$(Broadcast)\frac{\Gamma \vdash e : T', C'}{\Gamma \vdash \textbf{broadcast}\ e : T', C' :: ![T']}$$

Figure 7.1 Syntax, action, dynamic semantics, and static semantics for Broadcast.

message v to the end of the mailbox of the receiver of the message which is basically all the other processes present in the global configuration. The notation $\langle st_i, e_i, M_i.v\rangle_{id_i} \uplus \mathscr{P}$ denotes overriding the process configuration of process represented by the address id_i, where $\uplus$ is an overriding union operation. The configuration of all the other processes is updated by appending the mailboxes of each process with message v. The *broadcast* expression performs a *broadcast (v, id)* action.

The rule *(Broadcast)* presented in Figure 7.1 checks the broadcast expression by ensuring that the message e is well-typed and has value type T' and communication type C' in the typing context Γ. Hence the *broadcast* expression represented as *broadcast* e has value type T' and communication type $C' :: ![T']$ which is sequence of communication type of e and communication type send.

7.2 Multicast

The *multicast* extension enables sending messages to a group of processes found in the program, specified by a group of addresses. Figure 7.2 details an extension of λ_{mpc} with *multicast* feature, which sends message represented by e' to a group of processes represented by e in the program. We extend the λ_{ir} actions by *multi* action which are observed during the execution of *multicast* expression. Action can be sending message v' to a group of processes $\overline{v}$ by the process instance id.

المنارة للاستشارات
www.manaraa.com

$e ::= \ldots$	Exp	$a ::= \ldots$	Actions
$\mid$ ***multicast*** $e\ e'$	Multicast	$\mid$ *multicast* $(v',\ id,\ \overline{v})$	Multi

$$(Multicast)\frac{\overline{v} = (v1, v2, v3, \ldots) \qquad (\forall v_i \in \overline{v}) \wedge \forall \langle st_i, e_i, M_i\rangle_{v_i} \in \mathscr{P} \quad \mathscr{P}' = \langle st_i, e_i, M_i.v'\rangle_{v_i} \uplus \mathscr{P}}{\langle st, \mathscr{E}[\textbf{multicast}\ \overline{v}\ v'], M\rangle_{id} \mid\mid \mathscr{P} \overset{multicast(v',id,\overline{v})}{\hookrightarrow} \langle st, \mathscr{E}[v'], M\rangle_{id} \mid\mid \mathscr{P}'}$$

$$(Multicast)\frac{e = (e_1, \ldots, e_n) \quad \Gamma \vdash e' : T', C' \qquad \Gamma \vdash e : P(T_1\ C_1'), \ldots P(T_n\ C_n'),\ C_1 :: \ldots :: Cn \qquad ?[T'] \in C_i' \quad \ominus(?[T'], C_1' :: \ldots C_n') = C'''}{\Gamma \vdash \textbf{multicast}\ e\ e' : T_1, \ldots, T_n, C' :: C_1 :: \ldots :: C_n :: ![T'] :: C'''}$$

Figure 7.2 Syntax, action, dynamic semantics, and static semantics of Multicast.

A *multicast* expression is represented as *multicast* $\overline{v}\ v'$ where $\overline{v}$ represents the addresses of the group of processes to which message is sent and v' represents the message to be sent. The set of processes to which message is sent is represented as pair of processes using the lambda expression *pair*. The evaluation of *multicast* expression as shown in the Figure 7.2 appends the message v' to the end of the mailbox of the receiver of the message which is basically all the processes present in the group represented by $\overline{v}$. The notation $\langle st_i, e_i, M_i.v\rangle_{id_i} \uplus \mathscr{P}$ denotes overriding the process configuration of process represented by the address id_i, where $\uplus$ is an overriding union operation. The configuration of all the processes present in the group $\overline{v}$ is updated by appending the mailboxes of each process in the group $\overline{v}$ with message v'. The *multicast* expression performs a *multi* $(v',\ id,\ \overline{v})$ action. The rule *(Multicast)* shown in the Figure 7.2 type checks the *multicast* expression by ensuring that (a) the message e' is well-typed and has value type T' and reducing e' could lead to communication behavior as defined by the communication type C' in the typing context Γ and (b) expression e evaluates to identities of the recipient processes are well-typed and have value type as pair type containing $P(T_i\ C_i')$ where T_i represents the value type of the body of the recipient processes and C_i' is the communication type of the body of the recipient and reducing e could lead to the sequence of communication behavior represented by sequence of communication type C_i in the typing context

www.manaraa.com

$e ::= \ldots$ Exp

$|\ \textbf{\textit{guard}}\ x :\ T\ e\ e'\ e''$ Guard

$$(Guarded)\frac{\Gamma \vdash v :\ T,\ 0}{\langle st, \mathscr{E}[\textbf{guard}\ x : T\ \mathit{true}\ e'\ e''], v.M\rangle_{id} \mid\mid \mathscr{P} \overset{receive(v,T,id)}{\rightsquigarrow} \langle st, \mathscr{E}[[x := v]e'], M\rangle_{id} \mid\mid \mathscr{P}}$$

$$(Guarded)\frac{\Gamma \vdash v :\ T,\ 0}{\langle st, \mathscr{E}[\textbf{guard}\ x : T, 0\ \mathit{false}\ e'\ e''], v.M\rangle_{id} \mid\mid \mathscr{P} \overset{receive(v,T,id)}{\rightsquigarrow} \langle st, \mathscr{E}[e''], v.M\rangle_{id} \mid\mid \mathscr{P}}$$

$$(Guarded)\frac{\Gamma \vdash v :\ T',\ 0}{\langle st, \mathscr{E}[\textbf{guard}\ x : T\ e\ e'\ e''], v.M\rangle_{id} \mid\mid \mathscr{P} \overset{receive(v,T',id)}{\rightsquigarrow} \langle st, \mathscr{E}[e''], v.M\rangle_{id} \mid\mid \mathscr{P}}$$

$$(Guarded)\frac{\Gamma \vdash e :\ bool,\ C''' \qquad \Gamma \vdash e'' :\ T',\ C'' \qquad \Gamma, x :\ T \vdash e' :\ T',\ C'}{\Gamma \vdash \textbf{guard}\ x :\ T\ e\ e'\ e'' :\ T',\ C'''\ ::\ ?[T]\ ::\ C'\ \&\ C''}$$

Figure 7.3 Syntax, dynamic semantics, and static semantics for Guarded Receive.

Γ. Hence the communication type of *multicast* expression has communication type C' :: C_1 ::...:: C_n :: $![T']$ :: C''' where receive communication type $?[T']$ must be present in C_i'. Given a receive type $?[T']$ and a sequence of communication type, the function $\ominus$ creates a composite communication type that represents a non-deterministic composition of the communication behavior starting with the states that can receive the message of type T that is represented as C'''.

7.3 Guarded Receive

We also extend our calculus to support a conditional feature in the receive expression, which ensures that the effect of the message received is executed only if the guard is true. Figure 7.3 details an extension of λ_{mpc} with the guarded receive feature, stipulating that a process can receive a message v from the mailbox by satisfying the guard. If the message is dynamically type-checked in the context Γ, the message is assigned value type T, the communication type is 0, and the guard is true, the message is retrieved and e' is executed. Otherwise, the message is not retrieved, and expression e'' is executed. The rule *(Guarded)* type-checks the guarded receive expression. The type of the guard is Boolean in the typing context Γ. The expressions e' and e'' are assigned value

www.manaraa.com

type T' and communication type C' and C''. The communication type of the guarded receive is sequence of C''' which represents the communication of the guarded expression and choice between the communication represented by type $?[T] :: C'$ and C'' in the context Γ.

7.4 Non-blocking Receive

Sometimes we want to allow progress of the configuration even when the mailbox is empty, rather than risking a livelock due to the receive expression. As the mailbox is empty we cannot dynamically type check the messages as there is no message present in the mailbox. The non-blocking receive extension handles cases when the *nreceive* expression checks the mailbox for a message of certain kind (type) but the mailbox is empty. If a message of the expected type is present in the mailbox then *nreceive* proceeds in the same way as the standard *receive* expression that is present in our syntax. Otherwise, a future value is substituted in place of the message value in the body of the *nreceive* expression. Accessing this future value blocks the computation until the message value is resolved. The future value takes an argument T, which is the type of message scheduled to arrive at the mailbox in future. If a message arrives at the mailbox, then the *receive* expression unblocks the computation and the message is substituted in place of the future value. The typing rule for non-blocking receive is similar to the receive expression and is presented in the Figure 7.4.

7.5 Synchronization primitive, Wait

The message passing model presented in this thesis supports asynchronous communication. Many other practical message passing models implement some level of synchronization. Therefore, we extend our calculus to add the synchronization primitive "wait." Wait is a synchronization method that allows some process id to block until a process id' evaluates to a value. This value is then returned directly to the process id, bypassing id's mailbox.

Figure 7.5 details an extension of λ_{mpc} with wait primitive, which blocks a process until a particular process's body is reduced to a value. A wait expression is represented as *wait* id' where id' represents

www.manaraa.com

$e ::= \ldots$	Exp	$v ::= \ldots$	Value
$\mid$ ***nreceive*** $x{:}\ T\ e\ e'$	Non-blocking	$\mid$ ***future*** T	Future

$$(NReceive)\langle st, \mathscr{E}[\textbf{\textit{nreceive}}\ x : T\ e\ e'],\ \bullet\rangle_{id} \mid\mid \mathscr{P} \overset{local\ (id)}{\rightsquigarrow} \langle st, \mathscr{E}[[future\ T/x]e],\ \bullet\rangle_{id} \mid\mid \mathscr{P}$$

$$(NReceive)\langle st, \mathscr{E}[\textbf{\textit{nreceive}}\ x : T\ e\ e'],\ \bullet\rangle_{id} \mid\mid \mathscr{P} \overset{local\ (id)}{\rightsquigarrow} \langle st, \mathscr{E}[e'],\ \bullet\rangle_{id} \mid\mid \mathscr{P}$$

$$(NReceive)\frac{\Gamma \vdash v : T,\ 0}{\langle st, \mathscr{E}[\textbf{\textit{nreceive}}\ x : T\ e\ e'], v.M\rangle_{id} \mid\mid \mathscr{P} \overset{receive(v,T,id)}{\rightsquigarrow} \langle st, \mathscr{E}[[v/x]e], M\rangle_{id} \mid\mid \mathscr{P}}$$

$$(NReceive)\frac{\Gamma \vdash v : T',\ 0}{\langle st, \mathscr{E}[\textbf{\textit{nreceive}}\ x : T\ e\ e'], v.M\rangle_{id} \mid\mid \mathscr{P} \overset{receive(v,T',id)}{\rightsquigarrow} \langle st, \mathscr{E}[e'], v.M\rangle_{id} \mid\mid \mathscr{P}}$$

$$(Bsend)\frac{future\ T \qquad \langle st', e', M'\rangle_{id'} \in\ \mathscr{P}}{\langle st, \mathscr{E}[\textbf{\textit{send}}\ id'\ future\ T], M\rangle_{id} \mid\mid \mathscr{P} \overset{send(f,id,id')}{\rightsquigarrow} \langle st, \mathscr{E}[receive\ x : T\ send\ id'\ x\ send\ id'\ future\ T], M\rangle_{id} \mid\mid \mathscr{P}}$$

$$(Bset)\ \langle st, \mathscr{E}[\textbf{\textit{set}}\ future\ T], M\rangle_{id} \mid\mid \mathscr{P} \overset{set(id)}{\rightsquigarrow} \langle st, \mathscr{E}[receive\ x : T\ set\ x\ set\ future\ T], M\rangle_{id} \mid\mid \mathscr{P}$$

$$(Bfst)\ \langle st, \mathscr{E}[\textbf{\textit{fst}}\ pair\ future\ T\ v], M\rangle_{id} \mid\mid \mathscr{P} \overset{local(id)}{\rightsquigarrow} \langle st, \mathscr{E}[receive\ x : T\ fst\ pair\ x\ v\ fst\ pair\ future\ T\ v], M\rangle_{id} \mid\mid \mathscr{P}$$

$$(Bsnd)\ \langle st, \mathscr{E}[\textbf{\textit{snd}}\ pair\ v\ future\ T], M\rangle_{id} \mid\mid \mathscr{P} \overset{local(id)}{\rightsquigarrow} \langle st, \mathscr{E}[receive\ x : T\ snd\ pair\ v\ x\ snd\ pair\ v\ future\ T], M\rangle_{id} \mid\mid \mathscr{P}$$

$$(Babs)\ \langle st, \mathscr{E}[\boldsymbol{y : T\ e}\ future\ T], M\rangle_{id} \mid\mid \mathscr{P} \overset{set(id)}{\rightsquigarrow} \langle st, \mathscr{E}[receive\ x : T\ (y : T\ e\ x)\ (y : T\ e\ futureT)], M\rangle_{id} \mid\mid \mathscr{P}$$

$$(Bseq)\langle st, \mathscr{E}[\boldsymbol{future\ T\ ;\ e}], M\rangle_{id} \mid\mid \mathscr{P} \overset{set(id)}{\rightsquigarrow} \langle st, \mathscr{E}[e], M\rangle_{id} \mid\mid \mathscr{P}$$

$$(NReceive)\frac{\Gamma\ ,\ x : T \vdash e : T', C' \qquad \Gamma \vdash e' : T', C''}{\Gamma \vdash \textbf{\textit{nreceive}}\ \ x : T\ \ e\ e' : T',\ ?[T] :: C'\ \&\ C''}$$

Figure 7.4 Syntax, dynamic semantic and static semantic for Non-blocking receive.

المنارة للاستشارات
www.manaraa.com

$e ::= \ldots$ Exp
$\mid$ ***wait*** e Wait

$$(Wait)\frac{\langle st', v, M'\rangle_{id'} \in \mathscr{P}}{\langle st, \mathscr{E}[\textbf{\textit{wait}}\ id'], M\rangle_{id} \mid\mid \mathscr{P} \overset{local\ (id)}{\rightsquigarrow} \langle st, \mathscr{E}[v], M\rangle_{id} \mid\mid \mathscr{P}} \qquad (Wait)\frac{\Gamma \vdash e : T,\ C}{\Gamma \vdash \textbf{\textit{wait}}\ e : T,\ C}$$

Figure 7.5 Syntax, dynamic semantics, and static semantics for Wait.

the address of the process whose body should be reduced to a value to unblock the wait state. The notation $\langle st', v, M'\rangle_{id'} \in \mathscr{P}$ denotes such process configuration of process id' exists in the global configuration $\mathscr{P}$. The evaluation of wait expression shown in the Figure 7.5 reduces the body of the process id to the value v. The wait expression performs a local action. The rule *(Wait)* shown in the Figure 7.5 type checks the wait expression by assigning the type value type T and communication type C in the typing context Γ where T represents the type of the value to which the body of the process id' will eventually reduce to.

7.6 Extension in properties

We extend the properties presented in the §6 which is shown in Figure 7.6. Broadcast action of a message to all the process instance conflicts with receive actions of the same message in those processes. Similarly multicast action of a message conflicts with receive action if that message in the group of processes to which message is sent. *Happens-before* relation says that a *broadcast* action and a *multicast* action of a message must happen before it is being received by the other processes to which it is sent.

Lemma 7.6.1. ***(λ_{ir}'s mover properties)*** *Let $\mathscr{T}$ be the execution trace of an arbitrary Message passing program. Then, in trace $\mathscr{T}$ broadcast action broadcast (v, id) of process instance id is a left mover, as defined in Definition 4; a multicast (v, id, id$'$) of process instance id is left mover.*

Proof. Let a be an action with left and right neighbours a_l and a_r in the sub-trace $a_l \hookrightarrow a \hookrightarrow a_r$. We replace a with broadcast and multicast actions of a process instance id to show their mover properties in an arbitrary trace with arbitrary left and right neighbour actions from other process instance. In a subtrace $a_l \hookrightarrow$ *broadcast (v, id)* $\hookrightarrow a_r$, the broadcast action of a message v conflicts

www.manaraa.com

with receive action of the same message v. This is because swapping the broadcast action with its right neighbour allows receiving a message v from process instance id_i which is not even send by id. However, Message passing model's happens-before relation, in Figure 7.6, does not allow this by ensuring that a message v must be sent by process instance id to process instance id_i before it can be received by process instance id_i i.e. *broadcast (v, id)* $\prec$ *receive (v, T, id_i)*. This in turn means the broadcast action can not be right mover. Since broadcasting of the message $\overline{v}$ to process instance id must happen before its being received by process instances id_i, a broadcast action *broadcast (v, id)* can not be right neighbour to the receive action *receive (v, T, id_i)* and thus the broadcast action can be safely swapped with any of its left neighbour, i.e. the broadcast action is a left mover. The same argument applies with *multi (v, id, id_i)* . This in turn means broadcast and multicast action a are a left mover and not a right mover. □

Conflicting actions # and Their happens-before relation for the extensions: $\prec$

$$\frac{\forall \langle st_i, \mathscr{E}[\textbf{\textit{receive}}\ x : T\ e\ e'], v.M_i\rangle_{id_i} \in \mathscr{P}}{receive(v,T,\ id_i)\ \#\ broadcast(v,\ id)}$$

$$\frac{\overline{id} = (id_1, id_2, \ldots) \quad \forall id_i \in \overline{id} \quad \langle st_i, \mathscr{E}[\textbf{\textit{receive}}\ x : T\ e\ e'], v.M_i\rangle_{id_i} \in \mathscr{P}}{receive(v,T,\ id_i)\ \#\ multicast(v,\ id,\ id_i)}$$

$$\frac{\forall \langle st_i, \mathscr{E}[\textbf{\textit{receive}}\ x : T\ e\ e'], v.M_i\rangle_{id_i} \in \mathscr{P}}{broadcast(v,\ id) \prec receive(v,T,\ id_i)}$$

$$\frac{\overline{id} = (id_1, id_2, \ldots) \quad \forall id_i \in id \quad \langle st_i, \mathscr{E}[\textbf{\textit{receive}}\ x : T\ e\ e'], v.M_i\rangle_{id_i} \in \mathscr{P}}{multicast(v,\ id,\ \overline{id}) \ \prec\ receive(v,T,\ id_i)}$$

Figure 7.6 Conflicting actions in Message Passing Model for the extensions where # denotes conflict and their happens-before $\prec$ relation.

المنارة للاستشارات
www.manaraa.com

CHAPTER 8. FORMALIZATION

The syntax, semantics, type system and all proofs in this thesis have been formalized using the Coq proof assistant. Our formalization for the core λ_{ir} has 2263 lines of code. The Coq formalization is inspired by the lambda calculus formalization, but there are several differences that make it more interesting from a design perspective. First, we built support for message passing primitives such as spawning a process, sending and receiving a message, etc. Next, to support the intensional receive expression we incorporated dynamic type checking of messages at the head of the mailbox in the operational semantics of receive expression. Because of the dynamic typing feature, standard theorems like progress, preservation and soundness become more challenging to prove. We solved these challenges by leveraging the type information in the syntax of receive, that leads to an extra hypothesis in theorems which states that there exists a type T which is assigned to the message at the head of the mailbox by our inductive typing relation. This design is different from the traditional proofs for lambda calculus.

Also different from standard Coq encoding of STLC-like calculi, where only one expression takes one step at one time, we have incorporated parallel configurations, which enable several entities which take one step at the same time. Also, multiple configurations can take a step at the same time in our dynamic semantics. We enable this configuration-level concurrency via the definition of global configurations, consisting of a state, body and mailbox of a particular process. Therefore, a step relation is a multiple-place relation enclosed under a global configuration that enables processes to progress independently.

We also record the set of actions observed during the execution of a message passing program. This action record is represented as a trace of the program, which helps us to reason about the happens-before relation between the various actions and enables analysis of concurrency-related issues. Each execution creates a sequential record that could be used in debugging, validation,

المنارة للاستشارات
www.manaraa.com

verification, and other analysis tasks, such as reasoning about interference, order in which messages are received etc. We formalized the happens-before relation between the various actions in Coq and also proved their mover properties.

8.1 Parallel Composition

In addition to the formalization of λ_{ir} in Coq proof assistant and proving the theorems related to it, we also provide a library for parallel composition, shown in Figure 8.1. All the prior work uses list to model parallel composition which lack theorems related to structural congruence. Our library provides both the notion of parameterized parallel composition as well as theorems related to structural congruence. Our library includes the entire set definitions and lemmas that were used in our formalization; these are directly available for others to extend and build upon through Require Export command in Coq. In this library, we propose a general framework for parallel composition, and we define instances of the various parallel compositions of the variety of entities that support the congruence property. The parallel composition library is defined such that future constructions can be parameterized over any composition. In our library $comp(\alpha)$ stands for parallel composition holding data values in some type α.

$$
\begin{array}{rcl}
comp(\alpha) & : & Parallel\ Composiiton \\
comp(\alpha) & : & \alpha \mid : \ comp(\alpha) \\
empty & : & comp(\alpha) \\
append & : & comp(\alpha) ++ comp(\alpha) \rightarrow comp(\alpha) \\
membership & : & \alpha \in comp(\alpha) \rightarrow bool \\
subset & : & comp(\alpha) \subseteq comp(\alpha) \rightarrow bool \\
number & : & comp(\alpha) \rightarrow nat \\
add & : & comp(\alpha) + \alpha \rightarrow comp(\alpha) \\
same & : & comp(\alpha_1) \subseteq comp(\alpha_2) \wedge comp(\alpha_2) \subseteq comp(\alpha_1) \\
\mathsf{equiv} & : & comp(\alpha) \equiv comp(\alpha)
\end{array}
$$

Figure 8.1 Parallel composition library

المنارة للاستشارات

www.manaraa.com

Figure 8.1 lists our notational conventions. We declare that $comp(\alpha)$ is a type by parameterising it over types. An empty parallel composition exists for any α. The operations of append, membership, subset, number, add, and same are also available for any α. For example, in our λ_{ir} we use the parallel composition composed of the type process configuration where global configuration is a parallel composition of local process configurations. We use the structural congruence operation to indicate that $\equiv$ is a structural congruence relation over the parallel composition. We formalized it by following laws shown in Figure 8.2.

Structural Congruence: $P_1 \equiv P_2$

$$(Reflexivity)\frac{}{P \equiv P} \qquad (Append\ Empty\ Composition)\frac{}{P\ \ ++\ \ empty \equiv P}$$

$$(Symmetry)\frac{P_1 \equiv P_2}{P_2 \equiv P_1} \qquad (Transitivity)\frac{P_1 \equiv P_2 \quad P_2 \equiv P_3}{P_1 \equiv P_3}$$

$$(Commutative)\frac{}{P_1\ \ ++\ \ P_2 \equiv P_2\ \ ++\ \ P_1}$$

$$(Associativity)\frac{}{P_1\ \ ++\ \ (P_2\ \ ++\ \ P_3) \equiv (P_1\ \ ++\ \ P_2)\ \ ++\ \ P_3}$$

$$(Parallel)\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1\ \ ++\ \ P_2 \equiv P_1'\ \ ++\ \ P_2'}$$

Figure 8.2 Structural Congruence Rules.

Our parameterized parallel composition provides structural congruence with the reduction semantics. Two parallel compositions are structurally congruent if they are identical up to the structure. The dynamic nature of many calculi using parallel composition is represented using reduction rules or computation steps. We represent the reduction semantics, which means the computation step of the parallel composition, using three basic rules. The three basic rules of the computation are empty parallel composition can take computational step to an empty composition, if the parallel composition is formed by appending two parallel composition P and Q then if P takes step to P' then P ++ Q takes step to P' ++ Q and if Q takes step to Q' then P ++ Q takes step

المنارة للاستشارات

www.manaraa.com

- If P $\Rightarrow P'$ then $\exists\, P''$: $P' \equiv P'' \wedge$ P $\Rightarrow P''$.
- If P $\Rightarrow$ Q then $\exists\, Q'$: P $\Rightarrow Q'$ implies Q $\equiv Q'$.
- If P $\equiv P' \wedge P' \Rightarrow Q' \wedge$ Q $\equiv Q'$ implies P $\Rightarrow$ Q.
- If P $\Rightarrow$ Q then $\exists\, P'$: P $\equiv P' \wedge P' \Rightarrow$ Q.
- If P $\Rightarrow$ Q then $\exists\, P'\ Q'$: $P' \Rightarrow Q' \wedge$ Q $\equiv Q' \wedge$ P $\equiv P'$.

Figure 8.3 Theorems related to Congruence

to P ++ Q'. The parallel composition should be represented as appending two compositions to extract the proofs of theorems from the library.

We also proved various theorems associated with the congruence which can be found in the library. All the theorems are listed in the Figure 8.3. The design of the theorems is inspired by structural congruence property [8] for Π calculus and it is unique in the sense that we prove the theorems for parameterized parallel compositions regardless of just the parallel composition of processes. The notation $\Rightarrow$ represents the computation step any parallel composition may perform. The reduction relation is closed under the basic set of reduction rules such as if P $\Rightarrow P'$ then P ++ Q $\Rightarrow$ P' ++ Q, if Q $\Rightarrow Q'$ then P ++ Q $\Rightarrow$ P ++ Q' etc. These reduction rules can be extended to add more rules related to parallel composition like if P_1 ++ P_2 — P_1' ++ P_2' then P_1 ++ P_2 — P_3 ++ $P_4 \Rightarrow P_1'$ ++ P_2' — P_3 ++ P_4. In the above representation all the variables P, Q, P_1, P_2 etc represents a parallel composition. All the theorems listed in Figure 8.3 states that the parallel composition that are structurally congruent have the same reductions. All the theorems proved in the library can be reused if the composition is represented according to the signature presented in the library. The design of our reduction semantics and parallel composition make these theorems provable for parameterized parallel composition.

8.2 Formalization of the extensions

We have formalized the five extensions presented in our thesis in the section §7 in the Coq proof assistant. Each extension is provided in a separate Coq file. Each contains pieces of syntax,

www.manaraa.com

semantics, type systems, and the metatheoretical proofs needed by the addition of that extension. All the proofs of our core calculus were reused in formalizing the extensions without breaking the existing proofs. The table below presented in Figure 8.4 shows the number of lines of codes for the core λ_{ir} and then the number of lines of codes needed to add to each of the segments to insert each feature. All the Coq formalisms are indented using Coq recommended style [32].

Feature	SYN	SEM	TYPE	SOUNDNESS	GD	HB	Total
λ_{ir}	50	813	241	899	30	230	2263
+	+	+	+	+	+	+	+
Broadcast	1	68	4	44	0	1	118
Multicast	1	162	15	127	0	1	306
Guarded Receive	1	85	7	86	0	1	180
Non-Blocking Receive	2	102	4	58	0	1	167
Synchronization primitive "wait"	1	43	2	128	0	1	175

Figure 8.4 Lines of code in Coq. (SYN : Syntax, SEM : Semantics, TYPE : Typing rules, GD : Guaranteed delivery, HB : Happens-before relation, Total : Total number of lines of code, + : Number of lines of code added in each of the module for each extension).

Formalizing the new features helped us validate the extensibility of our calculus in the sense that all the proofs for the theorems were reused by simply extending the proofs to add the new features. Formalization of the feature *"broadcast"* required addition of a new auxiliary function that updates the mailbox of all the configurations present in the program. Formalization of *"guarded receive"* was similar to the *"receive"* operation except that we need to take into account the extra expression that resembles the guard both in the dynamic and static semantics. In the dynamic semantics, we need to make sure that the guard expression can be destructed to be only of three cases true, false or any expression e.

Formalization of *"non-blocking receive"* includes extra dynamic semantic rules for unblocking the state reached due to trying to access the future value. These rules required additional lines of codes in the existing proofs for taking care of the subgoal in the new proof that emerged due to the expressions that might get substituted by future values when the mailbox is empty. Also we

المنارة للاستشارات
www.manaraa.com

need to destruct the mailbox to deal with both the cases when the mailbox is empty and when the mailbox is not empty.

Formalization of *"multicast"* added the largest number of lines of code to the core λ_{ir}. In multicast we send a message to a set of processes. To avoid adding a list of types to our core calculus, we represented the set of addresses differently. We used standard lambda expression *"pair"* to represent the set of process addresses to which a message is sent. In the static and dynamic semantics we incorporated this information by modelling the set of addresses as a pair of addresses written using (pair id_1 (pair id_2 ...))) to represent (id_1, id_2, ...). Formalization of *"multicast"* also leads to the addition of an auxiliary function that updates only the mailbox of the process addresses present in the pair of addresses to which message is sent. Formalizing the synchronization primitive *"wait"* required a change in the formulation of preservation theorem to include an additional hypothesis for reasoning about the type of the body of other configuration present in the program. This is the only feature that required a change in the signature of preservation theorem.

All the features required less than 200 lines of codes except the multicast. The Coq encoding of the core λ_{ir} along with all the feature is presented in the supplementary material.

المنارة للاستشارات
www.manaraa.com

CHAPTER 9. RELATED WORK

There have been several proposals to explore the benefits of type systems that mix static and dynamic information, but these works have not focused on message passing systems that was the focus of λ_{ir}. Our design is inspired by Harper and Morrisett [14], but goes beyond its mechanical adaptation to address several unique challenges in the message passing setting.

There have also been several proposals for reasoning about the messages, its effect and the communication flow of message passing concurrency. For example, there are various type systems proposed for actor model incorporating session types, behavioral types, parameterized type and static types. Session type system proposed for Featherweight Erlang captures the flow of communication within a session but still uses dynamic pattern matching for retrieving messages from the mailbox [24]. Their approach doesn't retains any information about the type of messages at compile time but λ_{ir} does. A static type system for actors proposed by Fowler [10] uses two typing judgments: the standard judgment on values $\Gamma \vdash V : A$, and a judgment $\Gamma \mid B \vdash M : A$, which states that a term M has type A under Γ, and can receive values of type B. Their approach consider the case that an expression is only allowed to receive message of a single type B. But by using our design, a process can receive messages of multiple types from the mailbox.

Behavioral types [7] encode the intended communication protocols and guarantees that runtime computation implements these protocols but they don't have any mechanism like our intensional receive to type check the message at compile time and reason about its effect. A functional actor calculus which include primitive for sending a message, creating a new actor, and changing an actor's behavior was proposed by Agha [2]. In this model, actors are typically untyped. He *et al.* [15] have proposed a type system where each actor is parameterized by the type it handles. Though type is specified and actor is only allowed to receive message of that particular type but they are not allowed to handle message of multiple type which is allowed by our intensional design of receive

www.manaraa.com

expression. Oortwij [25] analyze the behavior of Message Passing Interface (MPI) but have no support to reason about the effect of message at run time and also no mechanism to typecheck the message at compile time. Yasutake and Watanabe proposed a calculus for actor model formalized in Coq proof assistant [34] but have not focused on providing static semantics. Colaco [6] has formalized an actor model called CAP where actors are dynamically created, which leads to orphan messages that may or may not be handled by the target actor in some execution path; the model statically detects these orphan messages. They are mainly concerned about these properties but have no mechanism like our intensional receive which intensionally inspects the messages.

Rajan proposed a programming model called Capsule-oriented programming [28] to deal with challenges of concurrency which favors modularity over explicit concurrency, encourages concurrency correctness by construction and exploits the modular structure of programs to expose implicit concurrency. It specifies a concurrent program as a collection of capsules and ordinary object-oriented classes where static analysis of capsules and its interaction is facilitated. Bagherzadeh and Rajan proposed a concurrent core calculus [3] and also presents its semantics which guarantees sparse interference and cognizant interference by controlling sharing among concurrent tasks, accessibility of states of tasks, and dynamic binding. Their calculus includes higher level concurrency constructs like asynchronous method invocation, future values, etc. as compared to λ_{ir} which has the design like traditional actor model. Also, their calculus is not mechanized while our calculus is fully mechanized. Rajan and Leavens [29] proposed a language called Ptolemy that has quantified and typed events. In their work, they have event types that completely decouples subject, and observer modules and also event type declares the types of information communicated between the announcement of events and handler methods. They provide modularization by separating event type declarations from the modules that announce events. Similarly, in our calculus, we decouple the sender and receiver and messages received by a process can be type checked irrespective of sender's information. Bagherzadeh, Rajan, Leavens and Mooney's proposed calculus of Ptolemy [4] with explicit event announcement and also propose a technique called translucid contracts that allows programmers to write modular specifications for code and also allow them to reason about

المنارة للاستشارات
www.manaraa.com

the code's control effects which support modular verification of interaction patterns used in the aspect-oriented code which is not only Ptolemy specific but can be used to reason about other AO interfaces and OO languages and also allow handlers to statically know about types of event they handle.

All of these calculi does not support static type checking of messages received by the processes and reasoning about them by intensionally inspecting their type at run time. Our intensional receive utilizes runtime information to reason about the effect of the message received from the mailbox at the same time retains static type safety. Also our core calculus aims to fill this gap by providing a foundation for message passing concurrency that supports type-effect system for static type checking for messages received by any process and is also fully mechanized.

المنارة للاستشارات
www.manaraa.com

CHAPTER 10. CONCLUSION

We have introduced intensional receive, a novel formulation of the receive expression in message passing concurrency, and λ_{ir}, a mechanized core calculus with intensional receive. This new design is aimed at improving reasoning about the type of message received by any process and its effect. Intensional design of receive expression integrates static and dynamic type checking and allows the effect of the message received to be intensionally inspected through a notion of dynamic typing. A distinct advantage is that this design enables reasoning about the effect of the message received from the head of the mailbox while retaining static type safety. We have demonstrated several applications of intensional design of receive expression in various programming patterns like multiplexing, safe pipelining, encoding state machines and supporting the chain of responsibility pattern. In each case, intensional receive helped provide better safety. We have also formalized λ_{ir} using the Coq proof assistant and prove its soundness.

In the future, we are planning to utilize λ_{ir} as a jumping-off point for the analysis of real-life applications. We believe that λ_{ir} forms a basis that be used to formalize properties about emerging message-passing architectures, including embedded operating systems that rely on message-passing to transfer all information between applications. Examples include ROS (Robot Operating System) [27] and NASA's cFE/cFS (core Flight Executive/core Flight System) software[1], which serves as a basis for many platforms such as the AOS (Autonomy Operating System) [22]. We plan on investigating this application in particular for AOS; proofs about the behaviors of its message-passing architecture will play an integral part in constructing the safety cases required by the FAA to allow AOS to fly on the target Unmanned Aerial Systems (UAS) platforms in U.S. airspace. The extensible foundation provided by λ_{ir} could also be extended to add context sensitivity to the typing environment so that we can reason in a different way about communications between the processes. The context insensitive environment merges all the information about communications happening

[1]NASA Core Flight System https://cfs.gsfc.nasa.gov/

المنارة للاستشارات
www.manaraa.com

with a particular process while in a context sensitive environment separate communication information with the individual process can be taken into account. We believe this variation will prove useful for consistency checking of the set of communications happening within a process.

المنارة للاستشارات

www.manaraa.com

BIBLIOGRAPHY

[1] Agha, G., Hewitt, C.: Actors: A conceptual foundation for concurrent object-oriented programming. In: Shriver, B., Wegner, P. (eds.) Research Directions in Object-oriented Programming, pp. 49–74. MIT Press, Cambridge, MA, USA (1987), http://dl.acm.org/citation.cfm?id=36160.36162

[2] Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (Jan 1997), http://dx.doi.org/10.1017/S095679689700261X

[3] Bagherzadeh, M., Rajan, H.: Panini: A concurrent programming model for solving pervasive and oblivious interference. In: Modularity'15: 14th International Conference on Modularity (March 2015)

[4] Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.: Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In: AOSD '11: 10th International Conference on Aspect-Oriented Software Development (March 2011)

[5] Bonér, J., Klang, V., Kuhn, R.: Akka library. http://akka. io (July 2009)

[6] Colaco, J.L., Pantel, M., Dagnat, F., Sallé, P.: Static safety analysis for non-uniform service availability in actors. In: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). pp. 463–. Kluwer, B.V., Deventer, The Netherlands, The Netherlands (1999), http://dl.acm.org/citation.cfm?id=646816.708629

[7] Crafa, S.: Behavioural types for actor systems. CoRR abs/1206.1687 (2012), http://arxiv.org/abs/1206.1687

المنارة للاستشارات
www.manaraa.com

[8] Engelfriet, J., Gelsema, T.: An exercise in structural congruence. Inf. Process. Lett. 101(1), 1–5 (2007), https://doi.org/10.1016/j.ipl.2006.08.001

[9] Flanagan, C.: Hybrid type checking. In: POPL '06. pp. 245–256. POPL '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1111037.1111059

[10] Fowler, S., Lindley, S., Wadler, P.: Mixing metaphors: Actors as channels and channels as actors. ECOOP'17: The European Conference on Object-Oriented Programming 7253 (2017)

[11] Griesemer, R., Pike, R., Thompson, K.: The go programming language. The Go Programming Language (2010)

[12] Hackett, B., Guo, S.y.: Fast and precise hybrid type inference for JavaScript. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 239–250. PLDI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2254064.2254094

[13] Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3), 202–220 (Feb 2009), http://dx.doi.org/10.1016/j.tcs.2008.09.019

[14] Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 130–141. POPL '95, ACM, New York, NY, USA (1995), http://doi.acm.org/10.1145/199448.199475

[15] He, J., Wadler, P., Trinder, P.: Typecasting actors: From akka to takka. In: Proceedings of the Fifth Annual Scala Workshop. pp. 23–33. SCALA '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2637647.2637651

[16] Hewitt, C.: Viewing control structures as patterns of passing messages. Artificial Intelligence. 8(3), 323–364 (Jun 1977), http://hdl.handle.net/1721.1/6272

المنارة للاستشارات
www.manaraa.com

[17] Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (Aug 1978), http://doi.acm.org/10.1145/359576.359585

[18] Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. SIGPLAN Not. 36(3), 128–141 (Jan 2001), http://doi.acm.org/10.1145/373243.360215

[19] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (Jul 1978), http://doi.acm.org/10.1145/359545.359563

[20] Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (Dec 1975), http://doi.acm.org/10.1145/361227.361234

[21] Long, Y., Liu, Y.D., Rajan, H.: Intensional effect polymorphism. In: Proceedings of the 29th European Conference on Object-oriented Programming. ECOOP'15 (July 2015)

[22] Lowry, M., Bajwa, A., Quach, P., Karsai, G., Rozier, K., Rayadurgam, S.: Autonomy operating system for uavs. Online: https://nari.arc.nasa.gov/sites/default/files/attachments/15%29%20Mike%20Lowry%20SAEApril19-2017.Final_.pdf (April 2017)

[23] Milner, R.: The polyadic-calculus: a tutorial. Logic and algebra of specification 94, 203–246 (1991)

[24] Mostrous, D., Vasconcelos, V.T.: Session typing for a featherweight erlang. In: Proceedings of the 13th International Conference on Coordination Models and Languages. pp. 95–109. COORDINATION'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2022052.2022059

[25] Oortwijn, W., Blom, S., Huisman, M.: Future-based static analysis of message passing programs. In: Proceedings of the Ninth workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software. pp. 65–72. PLACES 2016, Eindhoven, The Netherlands (April 2016), https://doi.org/10.4204/EPTCS.211.7

www.manaraa.com

[26] Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner. pp. 455–494. The MIT Press, Cambridge, MA (2000)

[27] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe (2009)

[28] Rajan, H.: Capsule-oriented programming. In: ICSE'15: The 37th International Conference on Software Engineering: NIER Track (May 2015)

[29] Rajan, H., Leavens, G.T.: Ptolemy: A Language with Quantified, Typed Events, pp. 155–179. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-70592-5_8

[30] Reppy, J.H.: Cml: A higher concurrent language. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. pp. 293–305. PLDI '91, ACM, New York, NY, USA (1991), http://doi.acm.org/10.1145/113445.113470

[31] Siek, J., Taha, W.: Gradual typing for objects. In: Proceedings of the 21st European Conference on Object-Oriented Programming. pp. 2–27. ECOOP '07, Springer-Verlag, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-73589-2_2

[32] The Coq Team: Coqstyle. https://coq.inria.fr/cocorico/CoqStyle (August 2013)

[33] Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang (2Nd Ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK (1996)

[34] Yasutake, S., Watanabe, T.: Actario: A framework for reasoning about actor systems. In: AGERE! 2015: the ACM SIGPLAN Workshop on Programming based on Actors, Agents, and Decentralized Control. pp. 1–10. ACM, New York, NY, USA (Oct 2015), http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

المنارة للاستشارات

www.manaraa.com

[35] Yi, J.: Cooperability: A New Property for Multithreading. Ph.D. thesis, University of California at Santa Cruz, Santa Cruz, CA, USA (2011), aAI3497948

للاستشارات المنارة

www.manaraa.com

APPENDIX. ADDITIONAL MATERIAL

Appendix is organized as follows:

§A.1 presents the operational semantics of basic lambda expressions;

§A.2 describes the static semantics of basic lambda expresisons.

§A.3 presents detailed definitions and proofs related to *happens-before*.

§A.4 presents inference algorithm.

A.1 Dynamic Semantics of STLC expressions

Local evaluation rule $\overset{a}{\rightsquigarrow}$**:** $\langle st, \mathscr{E}[e], M\rangle_{id} \;||\; \mathscr{P} \overset{a}{\rightsquigarrow} \langle st', \mathscr{E}[e'], M'\rangle_{id} \;||\; \mathscr{P}$

$$(First)\frac{}{\langle st, \mathscr{E}[\textbf{\textit{fst}}(pair\ v\ v')], M\rangle_{id}||\mathscr{P} \overset{local(id)}{\rightsquigarrow} \langle st, \mathscr{E}[v], M\rangle_{id}||\mathscr{P}}$$

$$(Second)\frac{}{\langle st, \mathscr{E}[\textbf{\textit{snd}}(pair\ v\ v')], M\rangle_{id}||\mathscr{P} \overset{local(id)}{\rightsquigarrow} \langle st, \mathscr{E}[v'], M\rangle_{id}||\mathscr{P}}$$

$$(If)\frac{}{\langle st, \mathscr{E}[\textbf{\textit{if}}\ true\ \textbf{\textit{then}}\ e1\ \textbf{\textit{else}}\ e2], M\rangle_{id}||\mathscr{P} \overset{local(id)}{\rightsquigarrow} \langle st, \mathscr{E}[e1], M\rangle_{id}||\mathscr{P}}$$

$$(If)\frac{}{\langle st, \mathscr{E}[\textbf{\textit{if}}\ false\ \textbf{\textit{then}}\ e1\ \textbf{\textit{else}}\ e2], M\rangle_{id}||\mathscr{P} \overset{local(id)}{\rightsquigarrow} \langle st, \mathscr{E}[e2], M\rangle_{id}||\mathscr{P}}$$

$$(Sequential)\frac{}{\langle st, \mathscr{E}[v; e], M\rangle_{id}||\mathscr{P} \overset{local(id)}{\rightsquigarrow} \langle st, \mathscr{E}[e], M\rangle_{id}||\mathscr{P}}$$

Figure A.1 Operational Semantics

المنارة للاستشارات
www.manaraa.com

Type Checking: $\Gamma \vdash e : T, C$

$$(Variable)\frac{(x,\ T) \in \Gamma}{\Gamma \vdash x :\ T,\ 0} \qquad (Nat)\frac{}{\Gamma \vdash n :\ nat,\ 0} \qquad (Pair)\frac{\Gamma \vdash e :\ T,\ C \qquad \Gamma \vdash e' :\ T',\ C'}{\Gamma \vdash \textbf{pair}\ e\ e' :\ T * T',\ C :: C'}$$

$$(First)\frac{\Gamma \vdash e :\ T * T',\ C :: C'}{\Gamma \vdash \textbf{fst}\ pair\ e\ e' :\ T,\ C} \qquad (Second)\frac{\Gamma \vdash e :\ T * T',\ C :: C'}{\Gamma \vdash \textbf{snd}\ pair\ e\ e' :\ T',\ C'}$$

$$(If)\frac{\Gamma \vdash e :\ bool,\ C \qquad \Gamma \vdash e' :\ T,\ C' \qquad \Gamma \vdash e'' :\ T,\ C''}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' :\ T,\ C :: C'\ \&\ C''} \qquad (Unit)\frac{}{\Gamma \vdash unit :\ unit,\ 0}$$

$$(True)\frac{}{\Gamma \vdash true :\ bool,\ 0} \qquad (False)\frac{}{\Gamma \vdash false :\ bool,\ 0} \qquad (Seq)\frac{\Gamma \vdash e :\ T,\ C \qquad \Gamma \vdash e' :\ T',\ C'}{\Gamma \vdash e\ ;\ e' :\ T',\ C :: C'}$$

Figure A.2 Static Semantics

A.2 Static Semantics of STLC expressions

A.3 Happens-Before Relations

Definitions A.3.1, A.3.2, A.3.3, A.3.4 are adapted from the previous work [35].

Definition A.3.1. ***(Trace)*** *An execution trace of a message passing program is a total order of actions a, as defined in Figure 4.1, performed by individual process instance in the program configuration $\mathscr{P}$ when evaluating the program using the local and global evaluation rules of Figure 4.2.*

Definition A.3.2. ***(Adjacent and neighbor actions)*** *Two actions a and b in a trace $\mathscr{T}$ are adjacent if one follows immediately after another. Two adjacent actions a and b are neighbors if they are performed by different process instances, i.e., instance(a) $\neq$ instance(b). The auxiliary function instance returns the process identifier of an action.*

Definition A.3.3. ***(Commuting and conflicting actions)*** *Let a_1 and a_2 be actions of process identifiers id_1 and id_2 in an execution trace $\mathscr{P} \overset{a_1}{\hookrightarrow} \mathscr{P}' \overset{a_2}{\hookrightarrow} \mathscr{P}''$. Then actions a_1 and a_2 commute, written as $a_1\ !\#\ a_2$, if swapping them in the trace results in the same final state in the trace starting with the same start state, i.e., $\mathscr{P} \overset{a_2}{\hookrightarrow} \mathscr{P}''' \overset{a_1}{\hookrightarrow} \mathscr{P}''$. Otherwise, a_1 and a_2 conflict, written as $a_1\ \#\ a_2$.*

www.manaraa.com

Conflicting actions # and their *happens-before* relation: $\prec$

$$\frac{\langle st, \mathscr{E}[\textbf{\textit{spawn}}\ x : T\ e], M\rangle_{id'} \in \mathscr{P}}{spawn(id',\ id'')\ \#\ send(v,\ id,\ id')} \qquad send(v,\ id,\ id') \prec spawn(id',\ id'')$$

Figure A.3 Conflicting actions in the Message Passing Model where # denotes conflict and their *happens-before* $\prec$ relation.

There can be conflicting actions in message passing program stemming from its semantics: a *send* action of a message to a process instance conflicts with another *send* action of a message to the same process instance; a *send* action of a message to a process instance conflicts with the *receive* action of the same message.

A *happens-before* relation [19] $\prec$ orders pairs of conflicting actions. For example, in our message passing model, sending of a message v by a process instance id to process instance id' must *happen-before receive* of the same message from the process instance id', i.e., $send(v,\ id,\ id') \prec receive(v,\ T,\ id')$. Figure A.3 shows our message passing model's conflicting actions and their *happens-before* relations.

Definition A.3.4. ***(Right-, left-, both-, and non-mover actions)*** *Let a_1 and a_2 be adjacent actions that are performed by different processes in an arbitrary execution trace $\mathscr{P} \overset{a_1}{\hookrightarrow} \mathscr{P}' \overset{a_2}{\hookrightarrow} \mathscr{P}''$. Then a_1 is a right-mover if swapping a_1 with a_2 in the trace results in the same final state given the trace began with the same start state, i.e., $\mathscr{P} \overset{a_2}{\hookrightarrow} \mathscr{P}''' \overset{a_1}{\hookrightarrow} \mathscr{P}''$. Conversely, a_2 is a left-mover if swapping it with a_1 results in the same final state, starting from the same start state. An action that can be swapped with its both left and right adjacent actions in any trace is a both-mover. Conversely, an action that cannot be swapped with either its left nor right neighbors is a non-mover.*

The operational semantics of the message passing model determine the mover properties of the actions. Lemma A.3.1 specifies mover properties of the message passing model's actions.

Lemma A.3.1. ***(Message passing model's mover properties)*** *Let $\mathscr{T}$ be the execution trace of an arbitrary message passing program. Then, in trace $\mathscr{T}$ send action send $(v,\ id,\ id')$ of process instance id is a left-mover, as defined in Definition A.3.4; a spawn $(id',\ id'')$ of process instance id'*

المنارة للاستشارات
www.manaraa.com

and a receive receive $(v,\ T,\ id')$ action of message v from the process instance id by process instance id' are a non-mover and local (id), get (id), set (id) and self are a both-mover.

Proof. The proof is based on happens-before relations of message passing model actions in Figure 4.1. Let a be an action with left and right neighbors a_l and a_r in the sub-trace $a_l \hookrightarrow a \hookrightarrow a_r$. We replace a with send, receive and spawn actions of a process instance *id* to show their mover properties in an arbitrary trace with arbitrary left and right neighbor actions from other process instance.

In a sub-trace $a_l \hookrightarrow$ send $(v,\ id,\ id') \hookrightarrow a_r$, the *send* action of a message v conflicts with the *receive* action of the same message v. This is because swapping the *send* action with its right neighbor allows receiving a message v from process instance id', which has not been sent by *id*. However, the message passing model's *happens-before* relation does not allow this by ensuring that a message v must be sent by process instance *id* to process instance id' before it can be received by process instance id', i.e., *send* $(v,\ id,\ id') \prec$ *receive* $(v,\ T,\ id')$. This in turn means the *send* action cannot be right-mover. Since sending of the message v to process instance id' must happen before v is received by process instance id', a *send* action *send* $(v,\ id,\ id')$ cannot be a right neighbor to the receive action *receive* $(v,\ T,\ id')$ and thus the *send* action can be safely swapped with any of its left neighbors, i.e., the *send* action is a left-mover.

A *receive* action *receive* $(v,\ T,\ id')$ only conflicts with another *receive* action if process instance id' receive message from two different process instances, since they both modifies the actions taken by process instance *id'* and also the mailbox of the process instance id'. A receive action cannot be swapped with either its left or right neighbor. Because it might be the case that we swap receive with its left neighbor or right neighbor and receive action happens before the send action for a message v. Also receive action conflicts with another receive action because there can be certain order of type of messages to be received by any process. Hence *receive* action *receive* $(v,\ T,\ id')$ cannot be swapped to its left *receive* $(v',\ T',\ id')$ and its right *receive* $(v'',\ T'',\ id')$ because message of type T should be received after message of type T' and before message of type T''. Hence receive action is a non-mover.

المنارة للاستشارات
www.manaraa.com

τ ::=	Value Types
$\mid \alpha$	variable
$\mid \tau \xrightarrow{\zeta} \tau$	Arrow
$\mid \tau \quad \zeta$	Process
$\mid$ *unit*	Unit
ζ ::=	Comm Types
$\mid \beta$	variable
$\mid$ *0*	Null
$\mid \zeta_1$ & ζ_2	Choice
$\mid \zeta_1 :: \ldots :: \zeta_n$	Seq
$\mid$ *!*$[\tau]$	Send
$\mid$ *?*$[\tau]$	Receive

Figure A.4 Variable representing value type and communication type

Similarly *spawn* (id', id'') is also non-mover. The reason behind the above argument is there might be the case that a process decided to spawn a process based on the message it receives. So send action which sends any message v to the process id' on which these actions are dependent can not happen after spawn actions. So they cannot be left-mover. A spawn action conflicts with the receive action of the same process instance id' as a process instance id' must receive the message from the mailbox before it decided to spawn a new process based on the message it receives. Hence spawn action cannot be a right-mover. Hence spawn action is a non-mover.

There are four actions which can be both-movers, they are *local (id)*, *get (id)*, *set (id)* and *self*. □ □

A.4 Type Inference Algorithm inspired by Hindley and Milner Approach

In this section we present type-inference algorithm which is inspired by Hindley-Milner algorithm. It is an deterministic step-by-step procedure for determining types for untyped expressions. Algorithm share the notions of "expressions" and "type". Accepts an environment Γ and an expression e. It produces a value type τ and communication type ζ or fails.

Type Checking: $\Gamma \vdash e : \tau, \zeta$

www.manaraa.com

$$(1)\ (\mathrm{Abs})\ \frac{\Gamma\ ,\ x:\ T \vdash e:T',\ C}{\Gamma \vdash x:\ T\ e:\ T \xrightarrow{C} T',\ 0}$$

$\mathscr{T}(\Gamma \vdash \mathrm{x}:\ \mathrm{T\ e}) =$

do $\tau,\ \zeta = \mathscr{T}(\Gamma,\ \mathrm{x}:\ \mathrm{T} \vdash \mathrm{e})$

do $(\beta = \mathrm{fresh} \wedge \beta = 0)$

return $\mathrm{T} \xrightarrow{\zeta} \tau,\ \beta$

$$(2)\ (\mathrm{App})\ \frac{\Gamma \vdash e:\ T \xrightarrow{C''} T',\ C \qquad \Gamma \vdash e'\ :\ T,\ C'}{\Gamma \vdash e\ e'\ :\ T',\ C :: C'\ ::\ C''}$$

$\mathscr{T}(\Gamma \vdash \mathrm{e}\ e') =$

do $\tau,\ \zeta = \mathscr{T}(\Gamma \vdash \mathrm{e})$

do $\tau',\ \zeta' = \mathscr{T}(\Gamma \vdash e')$

do $\beta = \mathrm{fresh}$

do $\alpha = \mathrm{fresh}$

do $\tau = \tau' \xrightarrow{\beta} \alpha$

return $\alpha,\ \zeta :: \zeta' :: \beta$

$$(3)\ (\mathrm{Send})\ \frac{\Gamma \vdash e'\ :T',\ C' \qquad \Gamma \vdash e:\ P\ (T\ C),\ C'' \qquad ?[T'] \in\ C \qquad \ominus (?[T'], C) = C'''}{\Gamma \vdash \textbf{\textit{send}}\ e\ e'\ :\ T',\ C'\ ::\ C''\ ::\ ![T']\ ::\ C'''}$$

$\mathscr{T}(\Gamma \vdash \mathrm{send\ e}\ e') =$

do $\tau,\ \zeta = \mathscr{T}(\Gamma \vdash \mathrm{e})$

do $\tau',\ \zeta' = \mathscr{T}(\Gamma \vdash e')$

do $\beta = \mathrm{fresh}$

do $\alpha = \mathrm{fresh}$

do $\tau = \mathrm{P}\ (\alpha,\ \beta)$

do $?[\tau'] \in \beta$

do $\beta' = \ominus\ (?[\tau'],\ \beta)$

return $\tau',\ \zeta' :: \zeta :: ![\tau'] :: \beta'$

المنارة للاستشارات

www.manaraa.com

$$(4)\ (\text{Receive})\ \frac{\Gamma\ ,\ x:\ T \vdash e:\ T',\ C' \qquad \Gamma \vdash e'\ :\ T',\ C''}{\Gamma \vdash \textbf{\textit{receive}}\ x:\ T\ e\ e'\ :\ T',\ ?[T]\ ::\ C'\ \&\ C''}$$

$\mathscr{T}(\Gamma \vdash \text{receive x : T e } e') =$

do $\tau, \zeta = \mathscr{T}(\Gamma, \text{x : T} \vdash \text{e})$

do $\tau, \zeta' = \mathscr{T}(\Gamma \vdash e')$

return τ, ?[T] :: ζ & ζ'

$$(5)\ (\text{Spawn})\ \frac{\Gamma,\ \textit{self_st}:\ T,\ \textit{self_id}:\ P(T',\ C') \vdash e:\ T',\ C'}{\Gamma \vdash \textbf{\textit{spawn}}\ x:\ T\ e:\ P(T'\ C'),\ 0}$$

$\mathscr{T}(\Gamma \vdash \text{spawn x : T e}) =$

do $\tau, \zeta = \mathscr{T}(\Gamma, \text{self_st : T, self_id : P}(T', C') \vdash \text{e})$

do $(\beta = \text{fresh} \wedge \beta = 0)$

return $\text{P}(\tau\ \zeta), \beta$

$$(6)\ (\text{Self})\ \frac{\Gamma(\textit{self_id}) = P(T\ C), 0}{\Gamma \vdash \textbf{\textit{self}}:\ P(T\ C),\ 0}$$

$\mathscr{T}(\Gamma \vdash \text{self}) =$

do $\alpha = \text{fresh}$

do $\beta = \text{fresh}$

do $\tau = \text{P}\ (\alpha\ \beta)$

do $\zeta = 0$

return τ, ζ

$$(7)\ (\text{Set})\ \frac{\Gamma \vdash e:\ T',\ C' \qquad \Gamma \vdash \textit{self_st}:\ T,\ 0 \qquad T' \preceq T}{\Gamma \vdash \textbf{\textit{set}}\ e:\ T,\ C'}$$

$\mathscr{T}(\Gamma \vdash \text{set e}) =$

do $\tau', \zeta' = \mathscr{T}(\Gamma \vdash \text{e})$

do $\tau, \zeta = \mathscr{T}(\Gamma \vdash \text{self_st})$

do $\zeta = 0$

do $\tau' \prec \tau$

return τ, ζ'

www.manaraa.com

$$(8)\ (\text{Get})\ \frac{\Gamma \vdash \textit{self_st} : T,\ 0}{\Gamma \vdash \textbf{\textit{get}} : T,\ 0}$$

$\mathscr{T}(\Gamma \vdash \text{get}) =$

do $\tau, \zeta = \mathscr{T}(\Gamma \vdash \text{self_st})$

do $\zeta = 0$

return τ, ζ

$$(9)\ (\text{Fix})\ \frac{\Gamma \vdash e : T \xrightarrow{C} T,\ 0}{\Gamma \vdash \textbf{\textit{fix}}\ e : T,\ C}$$

$\mathscr{T}(\Gamma \vdash \text{fix}\ \ \text{e}) =$

do $\tau, \zeta = \mathscr{T}(\Gamma \vdash \text{e})$

do (α = fresh)

do (β = fresh)

do $\tau = \alpha \xrightarrow{\beta} \alpha$

do $\zeta = 0$

return α, β.

www.manaraa.com