Chapter 9

Programming with Actors

Programming with the abstraction of *actors* can be done in several modern high level programming languages, including the SALSA programming language [54], Erlang [3], and Scala [38]. While Erlang and Scala extend a functional programming language core and focus on programming concurrent systems, SALSA extends an object-oriented programming language core and focuses on programming distributed and mobile systems (see Figure 9.1.)

The most significant semantic differences between SALSA and the actor language presented in Chapter 4, are the following:

- **Object-oriented core** Agha, Mason, Smith and Talcott's language (from now on, AMST) uses the λ calculus to model sequential computation within an actor. SALSA instead uses a sequential non-shared-memory subset of Java to model internal state and computation within an actor.

- **Classes as behaviors, methods as messages, tokens as return values** AMST uses a lambda abstraction to model an actor's behavior: receiving a message is modeled as applying the abstraction to the incoming message content. SALSA uses classes (in object-oriented terms) to model actor behaviors: individual actors are objects (instances of behavior classes) that conceptually encapsulate an independent thread of execution, messages are modeled as potential asynchronous method invocations on these instances, and tokens represent the future values returned by messages (see Figure 9.2.)

- **Static behaviors** AMST enables actors to completely change their behavior when becoming *ready* to receive new messages. SALSA's actors always have the same static *behavior*, however, this behavior may depend on the internal state of the actor, which can change between message receptions.

- **Coordination constructs** AMST uses asynchronous message sending as the only primitive form of communication. SALSA provides a number of

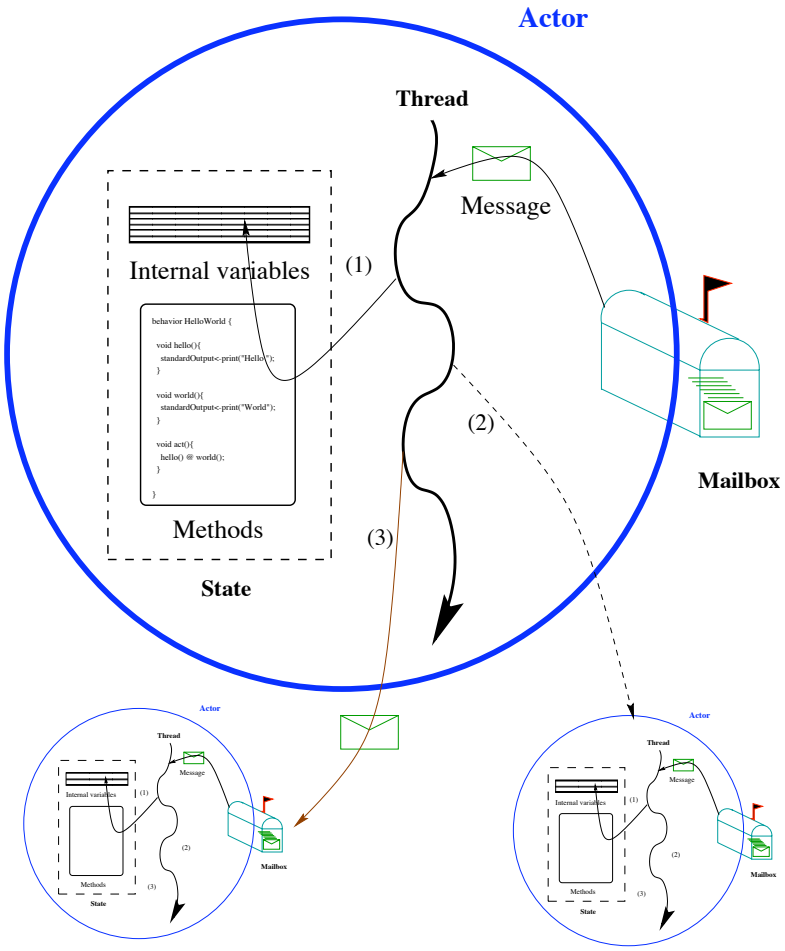

Figure 9.1: In response to a message, an actor can: (1) modify its local state, (2) create new actors, and/or (3) send messages to acquaintances.

Actor-oriented programming	**Object-oriented programming**
Behaviors	Classes (extending `UniversalActor` class)
Actors	Objects (instances of behavior classes)
Messages	Asynchronous method invocations
Tokens	Message return values

Figure 9.2: Modeling actors in an object-oriented programming language

AMST	SALSA	
`send`$(a, \texttt{pr}(m, v))$	`a <- m(v);`	Send named message
`new`(b)	`new B();`	Create new actor
`ready`(e)	`s = e; return;`	Behavior change

Figure 9.3: AMST Actor Syntax and Equivalent SALSA Syntax

higher-level constructs that facilitate coordinating otherwise independent activities between actors.

- **Distribution and mobility** AMST does not explicitly model actor locations and mobility. SALSA has a notion of *universal actor names* that enables transparent actor migration and location-independent communication.

In Section 9.1, we cover the syntax of SALSA. Section 9.2 specifies the operational semantics of SALSA. Section 9.3 covers programming idioms and patterns used in SALSA programs, including higher-level coordination constructs. Section 9.4 goes over concurrent, distributed and mobile programming examples in SALSA. Sections 9.5 and 9.6 conclude with a discussion and exercises.

9.1 SALSA Programming Language Syntax

9.1.1 Concurrent Systems Programming

A summary of the key AMST actor language primitives and the equivalent SALSA programming language syntax is given in Figure 9.3. AMST can use simple values as messages, on the other hand, SALSA actors use potential method invocations as messages. Sending the message is asynchronous: the sender does not block but instead continues with its sequential computation. The message is typically buffered, and when the recipient actor eventually receives the message, the corresponding method is actually executed. Creating a new actor takes its behavior and returns the newly created actor's name. In AMST, behaviors are λ calculus abstractions, while in SALSA, the behavior is the name of a previously defined class. Finally, in AMST an actor becomes ready to receive a new message with a new behavior e by executing `ready`(e). In SALSA, internal state is modeled directly as internal variables which can be reassigned new values and persist across message receptions. Returning from a method implicitly makes the actor ready to receive a new message.

SALSA uses the same primitive types as Java: booleans, characters, as well as different number types (byte, short, int, long, float, double.) SALSA also supports Java's primitive operations on these types. For standard I/O streams interaction, SALSA provides special primitive actors called `standardOutput`, `standardInput`, and `standardError`.

Figure 9.4 specifies SALSA's core language syntax. *Types* include primitive types, user-defined types, and a special *token* type to denote eventual message

return values. *Expressions* represent values potentially connected by operators. *Values* include variables, object field accesses, and object method invocations. *Variables* can be literal constants of different types, or allocation expressions that enable new object, as well as new local and remote actor, creations. *Statements* include local variable creation, blocks of statements, assignment statements, asynchronous continuation statements, conditionals, loops, and return statements. *Behaviors* consist of a name (used to create new instances,) and a body, which is a collection of state variables, constructors and message handlers.

Asynchronous statements can be named by tokens. For example, `token t = a <- m();` means that a message `m` is sent to actor `a` asynchronously, and when the message is delivered and the method `m` is actually executed, the returned value will be available as token `t`. An asynchronous statement can also be a *join block* which specifies that any associated continuation should wait until all the messages in the join block have been executed. For example, `join{ a1 <- m1(); a2 <- m2();} @ a <- m();` will send messages `m1` and `m2` concurrently to actors `a1` and `a2` but message `m` to actor `a` will only be sent when both `m1` and `m2` have been received and processed. This `join` statement is asynchronous: execution continues with the next statement immediately after sending the messages inside the join block. *Token-passing continuations* are chained sequences of asynchronous statements causally connected, for example `a1 <- m1() @ a2 <- m2(token);` means that message `m2` will only be sent to actor `a2` when message `m1` has been processed by actor `a1`: indeed, the argument to `m2` is the *token* that proves that `m1` has finished execution: it is set as the returned value of `m1`. Finally, *first-class continuations* allow an actor's message handler to *delegate* returning a value to another actor. For example, `b <- m2() @ currentContinuation;` executed in actor `a`'s `m1` message handler means that the return value of `m1` will be produced by `b`'s `m2` message handler, which should then have a compatible return type with `m1`.

Consider the AMST *ticker* example:

$$ticker = \texttt{rec}(\lambda b.\lambda t.\lambda n.\texttt{seq}(\texttt{send}(t, n+1), \texttt{ready}(b(t))))$$

We can write it in SALSA as follows:

```
behavior Ticker{
  void tick(int n){
    self <- tick(n++);
  }
}
```

In AMST, it is created and started as:

$$\texttt{letrec}\ t = \texttt{new}(ticker(t))\ \texttt{in}\ \texttt{send}(t, 0)$$

Equivalent code in SALSA would look as follows:

```
Ticker ticker = new Ticker();
ticker <- tick(0);
```

Type	::=		**Type**
		`token`	Token
	|	`boolean` | `char` | `int` | ...	Primitive
	|	*Id*	User-defined
Operator	::=	`+` | `-` | `>` | ...	Operator
Expression	::=	*Val* (*Operator Val*)*	**Expression**
Variable	::=		**Variable**
		boolean | *char* | *int* | ...	Literal
	|	*Id* (`.` *Id*)* | `this` | `super` | `self`	Path
	|	`new` *Id Arguments*	**Actor creation**
		[`at (` *Expression* [`,` *Expression*]`)`]	Optionally remote
	|	`(` *Expression* `)`	Expression
Arguments	::=	`(` [*Expression* (`,` *Expression*)*]`)`	Arguments
Val	::=	*Variable* (`.` *Id* | *Arguments*)*	**Value**
AsyncSend	::=		**Asynchronous send**
		[[`token`] *Id* `=`]	Optional named token
		([*Val* `<-`] *Id Arguments*	Message send
		| `join` *Block*)	Join block
Continuation	::=		**Continuation**
		(*AsyncSend* `@`)*	Token-passing
		(*AsyncSend*	Asynchronous send
		| `currentContinuation`);	First-class
Statement	::=		**Statement**
		State	Local variable
	|	*Block*	Block
	|	*Val* `=` *Expression*;	Assignment
	|	*Continuation*	Continuation
	|	`if (` *Expression* `)` *Statement*	Conditional
		[`else` *Statement*]	
	|	`while (` *Expression* `)` *Statement*	Loop
	|	`return` [*Expression*]`;`	Return
Block	::=	{ (*Statement*)* }	Block
Parameters	::=	`(` *Type Id* [(`,` *Type Id*)*] `)`	Formal parameters
Message	::=	(*Type* | `void`) *Id Parameters Block*	**Message handler**
Constructor	::=	*Id Parameters Block*	**Constructor**
State	::=	*Type Id* [`=` *Expression*]`;`	**State variable**
Body	::=	(*State* | *Constructor* | *Message*)*	Behavior body
Program	::=	`behavior` *Id* { *Body* }	**Behavior**

Figure 9.4: SALSA Programming Language Syntax

Notice that we do not need to explicitly keep an actor's own name, since the keyword `self` represents it. In fact, when a message has no explicit recipient, it is sent to `self`. So, the following code is equivalent:

```
behavior Ticker{
  void tick(int n){
    tick(n++);
  }
}
```

We could also represent the internal ticker state explicitly and write it in SALSA as follows:

```
behavior Ticker{
  int n;
  Ticker(int n){
    this.n = n;
  }
  void tick(){
    n++;
    tick();
  }
}
```

This ticker would be created and started as follows:

```
Ticker ticker = new Ticker(0);
ticker <- tick();
```

In this case, we have a state variable, `n`; a constructor `Ticker`, which following Java's convention uses the same name as the behavior; and a message handler `tick` that changes the actor's internal state and sends itself a message to continue ticking. Also notice that we can use the `this` keyword to refer to the actor's state variables as opposed to the constructor/method formal arguments that may have the same name.

9.1.2 Distributed and Mobile Systems Programming

SALSA concurrent programs can be directly used for distributed and mobile systems programming. SALSA's run-time system consists of a *name service* and *theaters*, which are Java virtual machines extended with actor creation, migration, and communication services. SALSA's actors can be assigned *Universal Actor Names* (UAN,) strings which are unique identifiers that enable location-independent communication with these actors, mediated by the naming service. For example, a *calendar* actor may be written as follows:

```
behavior Calendar implements ActorService{
  Appointment[] getAppointments(){
    ...
```

```
  }
  ...
}
```

A calendar instance can be created as follows:

```
Calendar myCalendar = new Calendar() at (uan, host);
```

`uan` represents the unique name of the calendar actor, which has Uniform Resource Identifier (URI) syntax, for example: `uan://wcl.cs.rpi.edu/~cvarela/calendar`, and the `host` represents the initial location of this actor, which is typically a string with a Domain Name System (DNS) domain name optionally followed by a listening port, for example `jupiter.wcl.cs.rpi.edu:4040`.

Any SALSA program can now obtain references to this universal actor and send messages to it, in the following manner:

```
Calendar calendar = reference Calendar(uan);
token appointments = calendar <- getAppointments();
```

Notice that the calendar actor can be migrated at any point to another theater by sending it a `migrate` message:

```
calendar <- migrate(newHost);
```

Example An applet server can be encoded in SALSA as follows:

```
behavior Applet {...}

Applet getApplet(String uan, String host){
  return new Applet() at (uan, host);
}
```

For each request, the applet server first creates a new applet actor with the given universal actor name at the given remote host, and then returns its reference to the applet client.

9.2 SALSA Programming Language Operational Semantics

To define the operational semantics of the SALSA programming language, we follow an approach similar to the one presented in Section 4.2 for the AMST language. Since AMST has a functional programming core based on the λ calculus and SALSA has an object-oriented core, we need to define an appropriate semantic model for the object-oriented core of SALSA, which is based on Java's. We will abstract over the sequential object-oriented core and focus on the concurrent actor semantics.

The operational semantics of SALSA is thus defined as a set of labelled transition rules from *actor configurations* to actor configurations specifying valid computations. Concurrent, distributed and mobile systems' evolution over time can be followed by applying the rules specified in the operational semantics in a manner consistent with fairness.

9.2.1 Actor Configurations

Actor configurations model concurrent system components as viewed by an idealized observer, frozen in time. An actor configuration is composed of:

- a set of individually named actors, and
- messages "en-route".

An *actor configuration*, κ, denoted $\alpha \parallel \mu$ contains an actor map, α, which is a function mapping actor names to actor expressions, and a multi-set of messages, μ. A message to actor named a to execute method m with argument v is denoted as $\langle a \Leftarrow m(v) \rangle$.

9.2.2 Reduction Contexts and Java Reduction Rules

An actor expression, e, is either a value v, or otherwise it can be uniquely decomposed into a *reduction context*, R, filled with a *redex*, r, denoted as $e = \mathsf{R} \blacktriangleright \mathsf{r} \blacktriangleleft$. The redex r denotes the next sub-expression to evaluate in a standard left-first, call-by-value evaluation strategy. The reduction context R denotes the surrounding expression with a *hole*. Figure 9.5 shows the syntax of redexes and reduction contexts for the core SALSA language. For example, the actor expression `new B() <- m(v)` can be decomposed into reduction context $\square$ `<- m(v)` filled with redex `new B()`. We denote this decomposition as:

`new B() <- m(v)` $=$ $\square$ `<- m(v)` $\blacktriangleright$ `new B()` $\blacktriangleleft$

Redexes are of two kinds: *purely object-oriented Java redexes* and *SALSA actor redexes*. Figure 9.6 depicts standard reduction rules for purely object-oriented redexes. These include standard Java evaluation strategies including branching, loops, and primitive operations.

9.2.3 Operational Semantics of Concurrent Execution

The transition rules depicted in Figure 9.7 are of the form $\kappa_1 \xrightarrow{l} \kappa_2$ over actor configurations, where κ_1 is the initial configuration, κ_2 is the final configuration, and l is the transition label.[1]

There are four rules, all of which apply to an actor a, which we call *in focus*: the first one, labelled **oo** specifies sequential object-oriented progress within the actor. The other three rules specify creation of and communication with

[1] We use $\alpha, [e]_a$ to denote the extended map α' which is the same as α except that it maps a to e, *i.e.*, $\alpha'[a] = e \wedge \forall a' \neq a, \alpha'[a'] = \alpha[a']$. We use $\uplus$ to denote multi-set union.

$\mathcal{E}_r$	::=		***Redexes***
	|	$\mathcal{F}(\mathcal{V},\ldots,\mathcal{V})$	*Primitive function application*
	|	$\mathcal{V}.\mathcal{V}(\mathcal{V})$	*Method invocation*
	|	`if (` $\mathcal{V}$ `)` $\mathcal{V}$ `else` $\mathcal{V}$	*Conditional execution*
	|	`while (` $\mathcal{V}$ `)` $\mathcal{V}$	*Loop*
	|	$\mathcal{V}$ `<-` $\mathcal{V}(\mathcal{V})$	*Message send*
	|	`new` $\mathcal{V}(\mathcal{V})$	*Actor creation*
	|	`return` $\mathcal{V}$	*Ready*
$\mathcal{R}$	::=		***Reduction Contexts***
		$\Box$	*Hole*
	|	$\mathcal{F}(\mathcal{V},\ldots,\mathcal{V},\mathcal{R},\mathcal{E}\ldots,\mathcal{E})$	*Primitive function application*
	|	$\mathcal{V}.\mathcal{V}(\mathcal{R})$	*Method invocation*
	|	$\mathcal{V}.\mathcal{R}(\mathcal{E})$	*Method invocation*
	|	$\mathcal{R}.\mathcal{E}(\mathcal{E})$	*Method invocation*
	|	`if (` $\mathcal{R}$ `)` $\mathcal{E}$ `else` $\mathcal{E}$	*Conditional execution*
	|	`while (` $\mathcal{R}$ `)` $\mathcal{E}$	*Loop*
	|	$\mathcal{V}$ `<-` $\mathcal{V}(\mathcal{R})$	*Message send*
	|	$\mathcal{V}$ `<-` $\mathcal{R}(\mathcal{E})$	*Message send*
	|	$\mathcal{R}$ `<-` $\mathcal{E}(\mathcal{E})$	*Message send*
	|	`new` $\mathcal{V}(\mathcal{R})$	*Actor creation*
	|	`new` $\mathcal{R}(\mathcal{E})$	*Actor creation*
	|	`return` $\mathcal{R}$	*Ready*

Figure 9.5: Redexes and Reduction Contexts

$f(v_1,\ldots,v_n)$	$\rightarrow_j$	v	if $f \in \mathcal{F}, v = [\![f]\!](v_1,\ldots,v_n)$
`this.`$m(v_1,\ldots,v_n)$	$\rightarrow_j$	$b\{v_1,\ldots,v_n/a_1,\ldots,a_n\}$	if $m(a_1,\ldots,a_n)\{b\} \in \mathcal{M}$
`if (true) v else _`	$\rightarrow_j$	`v`	
`if (false) _ else v`	$\rightarrow_j$	`v`	
`while (true) v`	$\rightarrow_j$	`v; while (b) e;`	if `while (b) e` $\rightarrow_j$ `while (true) v`
`while (false) _`	$\rightarrow_j$	`;`	

Figure 9.6: Java Core Language Reduction Rules

$$\frac{e \rightarrow_j e'}{\alpha, [e]_a \parallel \mu \xrightarrow{[\mathbf{oo}:a]} \alpha, [e']_a \parallel \mu}$$

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{new B(v)} \blacktriangleleft]_a \parallel \mu \xrightarrow{[\mathbf{new}:a,a']} \alpha, [\mathsf{R} \blacktriangleright a' \blacktriangleleft]_a, [\texttt{B(v)}]_{a'} \parallel \mu \quad a' \textit{ fresh}$$

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{a' <- m(v)} \blacktriangleleft]_a \parallel \mu \xrightarrow{[\mathbf{snd}:a]} \alpha, [\mathsf{R} \blacktriangleright \texttt{null} \blacktriangleleft]_a \parallel \mu \uplus \{\langle a' \Leftarrow m(v)\rangle\}$$

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{return;} \blacktriangleleft]_a \parallel \{\langle a \Leftarrow m(v)\rangle\} \uplus \mu \xrightarrow{[\mathbf{rcv}:a,m,v]} \alpha, [\texttt{this.m(v)}]_a \parallel \mu$$

Figure 9.7: SALSA Core Language Operational Semantics

other actors, and apply respectively to actor redexes: `new B(v)`, `a <- m(v)`, and `return`.

The **oo** rule subsumes sequential object-oriented computation using the Java reduction rules presented in Section 9.2.2.

The rule labelled **new** specifies actor creation, which applies when the focus actor a's redex is `new B(args)`: actor a creates a new actor a'. The behavior of a' is set to the value `B(args)`, an actor constructor. The actor a's redex is replaced by the new actor's name a'. a' must be fresh, that is, $a' \notin dom(\alpha) \cup \{a\}$.

The rule labelled **snd** specifies asynchronous message sending, which applies when the focus actor a's redex is `a' <- m(v)`: actor a sends a message containing value $m(v)$ to its acquaintance a'. Actor a continues execution and the network μ is extended with a new message $\langle a' \Leftarrow m(v)\rangle$.

The rule labelled **rcv** specifies message reception, which applies when the focus actor a's redex is `return;`, and there is a message in μ directed to a, e.g., $\langle a \Leftarrow m(v)\rangle$. The actor a's new state becomes `this.m(v)`, that is, its method m is executed with argument v. Notice that the reduction context R is discarded.

9.2.4 Operational Semantics of Distribution and Mobility

To model distributed and mobile actors, we extend the actor configurations' actor map α from Section 9.2.1, to include a location for each actor. That is, $\alpha(a) = \langle e, l\rangle$ where e is actor a's current state and l is actor a's current location.

The transition rules depicted in Figure 9.8 are of the form $\kappa_1 \xrightarrow{l} \kappa_2$ over location-extended actor configurations, where κ_1 is the initial configuration, κ_2 is the final configuration, and l is the transition label.[2]

There are five rules, all of which apply to an actor a at location l, which we call *in focus*: the first two, labelled **lc** and **rc** respectively, specify local and remote actor creation. The **mig** rule specifies actor migration from location l to

[2]We use $\alpha, [e]_{a@l}$ to denote the extended map α' which is the same as α except that it maps a to $\langle e, l\rangle$, *i.e.*, $\alpha'[a] = \langle e, l\rangle \wedge \forall a' \neq a, \alpha'[a'] = \alpha[a']$.

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{new B(v)} \blacktriangleleft]_{a@l} \parallel \mu \xrightarrow{[\mathbf{lc}:a,a']} \alpha, [\mathsf{R} \blacktriangleright a' \blacktriangleleft]_{a@l}, [\mathtt{B(v)}]_{a'@l} \parallel \mu \quad a' \textit{ fresh}$$

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{new B(v) at (n,l')} \blacktriangleleft]_{a@l} \parallel \mu \xrightarrow{[\mathbf{rc}:a,n]} \alpha, [\mathsf{R} \blacktriangleright n \blacktriangleleft]_{a@l}, [\mathtt{B(v)}]_{n@l'} \parallel \mu \quad n \textit{ fresh}$$

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{this.migrate(l')} \blacktriangleleft]_{a@l} \parallel \mu \xrightarrow{[\mathbf{mig}:a,l']} \alpha, [\mathsf{R} \blacktriangleright \texttt{null} \blacktriangleleft]_{a@l'} \parallel \mu$$

$$\alpha, [\mathsf{R} \blacktriangleright \texttt{reference B(n)} \blacktriangleleft]_{a@l} \parallel \mu \xrightarrow{[\mathbf{ref}:n]} \alpha, [\mathsf{R} \blacktriangleright \texttt{n} \blacktriangleleft]_{a@l} \parallel \mu \quad n \in \textit{dom}(\alpha)$$

$$\frac{\alpha, [e]_a \parallel \mu \xrightarrow{l} \alpha, [e']_a \parallel \mu'}{\alpha, [e]_{a@l} \parallel \mu \xrightarrow{l} \alpha, [e']_{a@l} \parallel \mu'}$$

Figure 9.8: SALSA Distributed and Mobile Language Operational Semantics

location l'. There is no special syntax in the language for migration: migration happens in response to a `migrate` message. The **ref** rule obtains a reference to an actor given its type and name. While in the semantics, we use variables for actor names for simplicity, in the SALSA implementation, we actually convert `String` representations of actor names into internal actor references. The last rule subsumes concurrent, but not distributed or mobile computation: any valid concurrent computation is also valid in the distributed and mobile context. In particular, sending and receiving messages is based on actor names, not on actor locations. This enables location-independent communication.

9.2.5 Fairness

The fairness requirement on valid SALSA program computation sequences and paths can follow the exact form as the one presented in the context of the AMST language in Section 4.2.4.

SALSA programming language implementations must satisfy *fairness* to properly follow the language semantics. This can be accomplished generally in one of two ways: If an actor is implemented as a thread, the underlying thread scheduling system must be fair. If multiple actors share a thread, both the actor scheduling system and the underlying thread scheduling system must then ensure fairness.

9.3 SALSA Programming Patterns

Activities in actor systems happen in response to messages. These activities are typically independent from each other and can happen concurrently and in any order. Causality conditions constrain these otherwise independent activities to occur in a partial order. Actor systems have only two causality conditions:

- **Actor creation** If activity p in actor a preceeds the creation of actor b, then activity p must preceed every activity q in actor b.
- **Message sending** If activity q in actor b happens in response to a message m from actor a, then all activities that preceed the message sending from a also must preceed q.

Asynchronous messaging and state encapsulation are important properties for systems modularity: they enable easier distribution and dynamic reconfiguration of software sub-components, which in turn promote scalability and fault tolerance. However, programming only with pure asynchronous messaging is very low level and prone to error.

In the SALSA programming language, there are higher level coordination constructs that enable building more complex interaction patterns, without sacrificing modularity. We present these language abstractions in the following sections.

9.3.1 Token-passing Continuations and Named Tokens

Continuations have been used in functional programming languages as a way to tell a function how to proceed after the function is computed. For example, there may be a *success* continuation specifying how to continue the computation if no errors are encountered and a *failure* continuation specifying how to handle anomalies, such as division by zero.

Token-passing continuations in SALSA also enable to establish causality conditions among otherwise independent activities. Since actor messages in SALSA are modeled as potential method invocations, it is natural to want to use the result of invoking a method, even if this invocation will happen asynchronously and in the future. We called the result of a future message invocation, a *token*, and we allow the use of the token as an argument to future messages.

Consider, for example, the code:

```
checking <- getBalance() @ savings <- transfer(checking, token);
```

In that example, we send a `getBalance` message to a `checking` account actor, and we specify the continuation as sending the message `transfer` to the `savings` account actor with two parameters: first, the `checking` account actor, and second, the `token` that represents the return value of the `getBalance` message to the checking account actor.

This example, could also have been written as:

```
token balance = checking <- getBalance();
savings <- transfer(checking, balance);
```

Named tokens enable arbitrary data-flow computations to be specified. For example, consider the following code:

```
token t1 = a1 <- m1();
token t2 = a2 <- m2();
token t3 = a3 <- m3( t1 );
token t4 = a4 <- m4( t2 );
a <- m(t1,t2,t3,t4);
```

Sending `m` to `a` will be delayed until messages `m1,...,m4` have been processed. `m1` can proceed concurrently with `m2`. `m3` can also proceed concurrently with `m4`, but `m3` can only happen after `m1` has been processed and likewise, `m4` can only happen after `m2` has been processed.

9.3.2 Join Continuations

Join blocks in SALSA enable to program *rendezvouz*-like interaction patterns. Several independent activities are barrier-synchronized so that only after all of them have finished execution does a given join continuation become enabled.

For example, consider the following code:

```
Searcher[] actors = { searcher0, searcher1, searcher2 };
join {
  for (int i=0; i < actors.length; i++){
    actors[i] <- find( phrase );
  }
} @ customer <- output( token );
```

This code sends the `find( phrase )` message to each actor in the `actors` array and after all the searcher actors have processed their respective messages, the `output` message is sent to the `customer` actor. The set of individual results (or `tokens`) is combined as a single array of tokens and sent as an argument to the `output` message.

9.3.3 Delegation through First-class Continuations

First-class continuations enable actors to delegate computation to a third party independently of the current processing context.

For example:

```
behavior A{
  int m(...){
    b <- n(...) @ currentContinuation;
  }
  ...
}
```

An instance `a` of this behavior `A`, when processing message `m` asks (delegates) actor `b` to respond to this message `m` on its behalf by processing its message `n`. In this example, actor `b`'s message handler for `n` should return a value of type `int`.

Let us consider an example:

```
behavior Calculator {

  int fib(int n) {
    Fibonacci f = new Fibonacci();
    f <- compute(n) @ currentContinuation;
  }

  int add(int n1, int n2) {return n1+n2;}

  void act(String args[]) {
    fib(15) @
      standardOutput <- println(token);
    fib(5) @ add(token,3) @
      standardOutput <- println(token);
  }
}
```

The `Calculator` behavior delegates the computation of Fibonacci numbers (`fib`) to a newly created `Fibonacci` actor. In response to the `act` message, the calculator actor sends itself two messages. The first one requests the calculation of `fib(15)` with the continuation specified as printing the result to standard output. The second request (`fib(5)`) specifies a different continuation: to add the result to the number `3` by sending itself an `add` message with the token (result of `fib(5)`) and the number `3` as arguments, and then send the result to the `standardOutput` actor for printing.

The `Fibonacci` behavior could be encoded as follows:

```
behavior Fibonacci {

  int compute(int n) {
    if (n == 0) return 0;
    else if (n <= 2) return 1;
    else {
      Fibonacci fib = new Fibonacci();
      Calculator calc = new Calculator();
      token x = fib <- compute(n-1);
      compute(n-2) @
        calc <- add(x,token) @
          currentContinuation;
    }
  }
```

```
}
```

Notice that the `Fibonacci` behavior creates two helper actors to perform the computation: an actor with the `Fibonacci` behavior to compute `fib(n-1)` and an actor with the `Calculator` behavior to compute the addition of `fib(n-1)` (produced by the helper actor) and `fib(n-2)` (produced by itself.) Notice how the `currentContinuation` keyword enables passing the current continuation, which is different at the top level and at each recursive step.

9.4 Common Examples

9.4.1 Reference Cell in SALSA

A reference cell in SALSA can be written as follows:

```
behavior Cell{
  Object content;
  Cell(Object c){
    content = c;
  }
  Object get(){
    return content;
  }
  void set(Object c){
    content = c;
  }
}
```

The SALSA cell client code can be written as follows:

```
Cell c = new Cell("hello");
c <- get() @
  standardOutput <- print(token);
c <- set("world");
```

The return value of the `get` message (or `token`) will be passed as an argument to the `print` message sent to the `standardOutput` actor. Notice that since message sending is asynchronous, this code does not ensure that the `get` message will be processed before the `set` message. To do that, a token-passing continuation may be used for sequencing as follows:

```
Cell c = new Cell("hello");
c <- get() @
  standardOutput <- print(token) @
  c <- set("world");
```

In this case, the `set` message will only occur after the `print` message is processed. However, these two messages should be able to execute concurrently, so another version of the code follows:

```
Cell c = new Cell("hello");
token t = c <- get() @
  c <- set("world");
standardOutput <- print(t);
```

In this new version of the code, the `set` message will wait until the `get` message has been completed. Furthermore, the `print` message will also wait for the completion of `get` due to the causality imposed by token `t`. As desired, there is no causality between messages `set` and `print` anymore.

9.4.2 Mutual Exclusion in SALSA

Two actors can mutually exclude themselves from accesing a shared resource concurrently, *i.e.*, they can ensure that only one of them is using the resource at a given point in time, by using a *semaphore*, as follows:

```
behavior Semaphore {

  UniversalActor holder = null;

  boolean get(UniversalActor holder){
    if (this.holder == null){
      this.holder = holder;
      return true;
    } else return false;
   }

  void release() { this.holder = null; }
}
```

The semaphore contains state that represents who currently holds access to the shared resource, or `null` if the resource is available. The semaphore replies `true` to the customer requesting access if allowed, and `false` otherwise.

A customer that keeps trying to get access to the shared resource may be encoded as follows:

```
behavior Customer {

  Semaphore sem;

  Customer(Semaphore sem){
    this.sem = sem;
  }
```

```
  void go(){
    sem <- get(self) @
      use(token);
  }

  void use(boolean ok){
    if (ok) {
      join {
        ... // critical code section
      } @ sem <- release();
    }
    else { go(); }
  }
}
```

Two customer actors `c1` and `c2` that mutually exclude each other during the critical code sections, can be written as:

```
Semaphore s = new Semaphore();
Customer c1 = new Customer(s);
Customer c2 = new Customer(s);

c1<-go();
c2<-go();
```

Clearly, the example could be enhanced so that an actor is notified when the resource becomes available, instead of busy-waiting as coded above.

9.4.3 Dining Philosophers in SALSA

The famous dining philosopher example can be encoded in SALSA as follows:

```
behavior Philosopher{

  Chopstick left, right;

  Philosopher{Chopstick left, Chopstick right){
    this.left = left; this.right = right;
  }

  boolean pickLeft(){
    left <- get(self) @ currentContinuation;
  }

  boolean pickRight(){
    right <- get(self) @ currentContinuation;
  }
```

```
  void eat(){
    pickLeft() @
    gotLeft(token);
  }

  void gotLeft(boolean leftOk){
    if (leftOk) {
      pickRight() @
      gotRight(token);
    } else
      eat();
  }

  void gotRight(boolean rightOk){
    if (rightOk) {
      join {
        standardOutput <- println ("eating...");
        left <- release();
        right <- release();
      } @ standardOutput <- println ("thinking...") @
      eat();
    } else
      gotLeft(true);
  }

}
```

The philosopher first attempts to get the left chopstick, then the right chopstick, then eats, and releases the chopsticks. If it fails to obtain a chopstick, it keeps trying. The chopstick can be encoded as follows:

```
behavior Chopstick extends Semaphore{}
```

It is possible to encode the chopstick so that it keeps track of a waiting philosopher, if it is not available, and so that it notifies the waiting philosopher when it becomes available. It is also possible to encode the dining philosophers in such a way that they do not deadlock (all picking up the left chopstick and waiting forever for the right one to become available.) These are left as exercises for the reader.

9.4.4 Mobile Reference Cell in SALSA

The code for a mobile reference cell in SALSA is almost identical to the code given in Section 9.4.1. The ability to migrate is automatically inherited from the root `UniversalActor` behavior that all SALSA programs extend. One important difference for actors that are meant to provide a *service*, is that they should

implement the `ActorService` interface. This empty interface simply tells the SALSA run-time system not to garbage collect actors with this behavior.

```
behavior Cell implements ActorService{
  Object content;

  Cell(Object initialContent) {
    content = initialContent;
  }

  Object get() {
    standardOutput <- println ("Returning:"+content);
    return content;
  }

  void set(Object newContent) {
    standardOutput <- println ("Setting:"+newContent);
    content = newContent;
  }
}
```

The `standardOutput` actor refers to the standard output stream in the actor's *current* location. Therefore, as the `Cell` actors migrate, the messages get printed in different hosting environments.

The following code illustrates how to create a cell, migrate it to a different hosting environment, and access it there:

```
behavior MovingCellTester {

  void act( String[] args ) {
    if (args.length != 3){
      standardError <- println("Usage:
        salsa MovingCellTester <UAN> <Host1> <Host2>");
      return;
    }

    Cell c = new Cell("Hello") at (args[0], args[1]);

    standardOutput <- print( "Initial Value:" ) @
    c <- get() @ standardOutput <- println( token ) @
    c <- set("World") @
    standardOutput <- print( "New Value:" ) @
    c <- get() @ standardOutput <- println( token ) @
    c <- migrate(args[2]) @
    c <- set("New World") @
    standardOutput <- print( "New Value at New Location:" ) @
    c <- get() @ standardOutput <- println( token );
```

```
   }
}
```

9.5 Bibliographic Notes

The SALSA programming language [54] is the result of studying how to introduce a well-founded model of concurrency, the actor model, to an object-oriented programming audience. SALSA's run-time system is the result of studying how to develop worldwide computing systems using distributed and mobile actors over the Internet. SALSA was created by Carlos Varela and Gul Agha in the 1998-2001 period [49, 51, 54, 50].

The operational semantics of SALSA in Chapter 9 has followed closely the structure of Chapter 4. The key difference is the use of a sequential non-shared-memory subset of Java for modeling individual actor computation, instead of the call-by-value λ calculus. We have abstracted over many details in the full SALSA language, inherited from Java, including syntax and semantics for arrays, interfaces and inheritance, and typing and polymorphism. We have also skipped discussion on message properties and mobile actor garbage collection. We refer the reader to [53] for a more detailed explanation of the language's capabilities and more thorough examples.

The SALSA programming language and framework have enabled research on several distributed computing areas. We refer interested readers to related publications on advanced topics including: decentralized naming services [48], distributed systems visualization [11], adaptive systems through dynamic reconfiguration [12, 29], distributed and mobile garbage collection [56], applications malleability [13, 30, 14], and fault-tolerant distributed computing [19, 5]. SALSA has also been used as a basis for developing computational science applications [55, 10, 15, 16, 8].

9.6 Exercises

1. Write an actor behavior in SALSA that computes the product of numbers in the leaves of a binary tree (see *e.g.*, *treeprod(t)* in Section 4.1.1.) Your program should compute the product concurrently and use a join continuation.

2. Modify the *mutual exclusion* example in SALSA (see Section 9.4.2) so that it does *not* use busy-waiting. Instead of continuously asking, the semaphore customer gets notified when the resource becomes available.

3. Modify the *dining philosophers* example in SALSA (see Section 9.4.3) so that philosophers can not deadlock.

4. How would you implement token-passing continuations in terms of actor creation and message passing?

5. How would you implement join blocks in terms of actor creation and message passing?

6. How would you implement first-class continuations in terms of actor creation and message passing?

7. Write a *distributed queue* abstract data type in SALSA.

8. Create a *dining nomad philosophers* example in SALSA. Philosophers eat in a *dining room* and think in a *thinking room*, assuming different rooms are in different sites.

9. Create a *mobile address book* behavior that keeps track of contacts (name, email, phone.) Start it in a site, query it, update it, migrate it to another site, and query it and update it again.

10. Develop a *farmer/worker* framework in SALSA to distribute computations over the Internet.