CS 6112 (Fall 2011)
Foundations of Concurrency
6 October 2011
Scribe: Jean-Baptiste Jeannin

Cornell University Department of Computer Science

Today we will see two different kinds of type systems for the π-calculus. After quickly reviewing the simply typed languages, we will move on to the π-calculus. In the π-calculus, we will add in a channel action, which has two actions, read and write, as well as sub-typing relations. In a second part of the lecture, we will see session types, and use process calculi types.

The reading for today was *Secure Implementations for Typed Session Abstractions*, by R. Corin, P.M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer, published in CSF'07, and available at available at `http://dx.doi.org/10.1109/CSF.2007.29`. Throughout this lecture we will follow a deck of slides on the typed π-calculus by Francesco Zappa Nardelli, available at `http://moscova.inria.fr/~zappa/teaching/mpri/2006/pitypes.pdf`.

In sequential languages, types are mainly used to detect simple programming errors at compilation time. They allow to detect that some expressions make no sense, and thus reject them. They are also useful to optimize compilers or reason about programs.

1 Simply-typed π-calculus

In the π-calculus so far, the only values are names. We now extend the π-calculus with base values of types `int` and `bool`, and with tuples. Now we can write terms which make no sense, like trying to add with a boolean:

$$\bar{x}\langle \mathsf{true}\rangle.P \mid\mid x(y).\bar{z}(y+1)$$

or even worse, trying to receive a channel, but instead receiving a bool:

$$\bar{x}\langle \mathsf{true}\rangle.P \mid\mid x(y).\bar{y}(4)$$

These terms raise runtime errors; here a runtime exception is raised. Typing is about preventing runtime errors. To do so and as usual, we perform some simple static analysis before we run the program. Names are values which denote channels. But in the term $\bar{x}\langle 3\rangle.P'$, what should the type of x be? The idea is to state that x should be a channel that can transport values of type `int`. We introduce a parameterized type constructor ch and write:

$$x : \mathsf{ch}(\mathtt{int})$$

1.1 Syntax, semantics and type system

The syntax of types T, messages M and processes P is given by:

$$T \ ::= \ \mathsf{ch}(T) \mid T \times T \mid \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool}$$

$$M \ ::= \ x \mid (M, M) \mid () \mid 1, 2, \ldots \mid \mathtt{true} \mid \mathtt{false}$$

$$P \ ::= \ \mathbf{0} \mid x(y:T).P \mid \bar{x}\langle M\rangle.P \mid P||P \mid (\nu x:T)P \mid \mathsf{match}\ z\ \mathsf{with}\ (x:T_1, y:T_2)\ \mathsf{in}\ P \mid !P$$

It is a small variation on the π-calculus we have seen so far: we have annotated with a type any bound variable on a receive, and any new variable

The type environments are $\Gamma \ ::= \ \emptyset \mid \Gamma, x:T$, and typing judgments are:

- $\Gamma \vdash M : T$, meaning that value M has type T under the environment Γ;
- $\Gamma \vdash P$, meaning that process P respects the type assignment Γ.

The typing rules are pretty simple and are in the slides. The interesting ones are:

$$\frac{\Gamma, x : T \vdash P}{\Gamma \vdash (\nu x : T)P} \qquad \frac{\Gamma \vdash x : \mathsf{ch}(T) \quad \Gamma, y : T \vdash P}{\Gamma \vdash x(y : T).P} \qquad \frac{\Gamma \vdash x : \mathsf{ch}(T) \quad \Gamma \vdash M : T \quad \Gamma : P}{\Gamma \vdash \bar{x}\langle M\rangle.P}$$

The simply-typed π-calculus makes every single process terminate. We need to have a more complicated type system for Turing completeness.

1.2 Soundness

Type soundness of the type system can be proved along the lines of Wright and Felleisen's syntactic approach to type soundness: proving progress and preservation guarantees that the system will not encounter any type error.

We extend the syntax with a special expression `wrong`, that is a normal form but not a value, and show that `wrong` can never occur in a well-typed system. Adding the `wrong` statement can be a huge pain, since you need to add it to each one of your operational semantics terms. If you have run down to a value (e.g., **0**), you do not have to show that it can take a step (progress lemma).

1.3 Sub-typing

We separate the channel type into two different capabilities:

- $i(T)$ input (read) capability
- $o(T)$ output (write) capability

A channel without annotation can do both inputs and outputs. This forms a basis for sub-typing on channels.

As usual with sub-typing, the directionality is bizarre, for example:

$$x : o(o(T)) \vdash (\nu y : \mathsf{ch}(T))\bar{x}\langle y\rangle.!y(z : T)$$

is well-typed, because $\mathsf{ch}(T) <: o(T)$. If channel x is a type that sends output channels that sends output channels that send T, then a new y of $\mathsf{ch}(T)$ can be sent on x, and then values of type T can be sent on it.

Formally, the sub-typing relation is a preorder (it is reflexive and transitive), and capabilities can be forgotten:

$$\mathsf{ch}(t) <: i(T) \qquad \mathsf{ch}(t) <: o(T)$$

i is a covariant type constructor, o is contravariant, and ch is invariant:

$$\frac{T_1 <: T_2}{i(T_1) <: i(T_2)} \qquad \frac{T_2 <: T_1}{o(T_1) <: o(T_2)} \qquad \frac{T_2 <: T_1 \quad T_1 <: T_2}{\mathsf{ch}(T_1) <: \mathsf{ch}(T_2)}$$

i needs to be covariant because if ints are a subtype of reals, it means that we can receive ints and then use them as reals, so an input channel for ints should be a subtype of an input channel for reals. o needs to be contravariant, because it is always OK to use an output channel expecting reals when you have an output channel expecting ints, since the process on the other side can use the ints you give it as reals. The invariant nature of i and contravariant nature of o leads to the invariant nature of ch.

Example 1.
$\nu(a : \mathsf{ch}(\mathsf{ch}(\mathtt{int})), x : \mathsf{ch}(\mathtt{int}))($
$\bar{a}\langle x\rangle \mid\mid$
$a(x : i(\mathtt{int}).P \mid\mid$
$a(x : o(\mathtt{int}).Q)))$

Finally, we need to update the typing rules as in:

$$\frac{\Gamma \vdash x : i(T) \quad \Gamma, y : T \vdash P}{\Gamma \vdash x(y : T).P} \qquad \frac{\Gamma \vdash x : o(T) \quad \Gamma \vdash M : T \quad \Gamma : P}{\Gamma \vdash \bar{x}\langle M\rangle.P} \qquad \frac{\Gamma \vdash M : T_1 \quad T_1 <: T_2}{\Gamma \vdash M : T_2}$$

1.4 Types for reasoning

A type is a contract between a process and its environment. The environment must respect the constraints imposed by the typing discipline. Is is only possible to plug in a process into a context where the contracts will not be violated.

Thus we might only have to consider the well-typed contexts for congruence relations from last time. This makes it easier to show that two processes are the same, it reduces the number of legal contexts, and gives us more process equalities.

We now get the typed versions of all the equivalence relations we have played with in this course so far. Informally, we say that processes P and Q are equivalent in Γ, and we write $P \cong_\Gamma Q$, if and only if $\Gamma \vdash P, Q$, and they are equivalent in all the testing contexts that respect the types in Γ.

Now they are processes we were able to distinguish in the untyped π-calculus but cannot in the typed π-calculus, for example:

$$P = (\nu x)\bar{a}\langle x\rangle.\bar{x}\langle\rangle \qquad Q = (\nu x)\bar{a}\langle x\rangle.\mathbf{0}$$

This is because under $\Gamma = a : \mathsf{ch}(o(\mathtt{unit}))$, the residual $\bar{x}\langle\rangle$ is deadlocked in all situations.

Another example is given by the specification and implementation of the factorial function:

$$\begin{aligned} \mathsf{Spec} &= !f(x, r).\bar{r}\langle \mathsf{fact}(x)\rangle \\ \mathsf{Imp} &= !f(x, r).\mathtt{if}\ x = 0\ \mathtt{then} \bar{r}\langle 1\rangle\ \mathtt{else}\ (\nu r')\bar{f}\langle x - 1, r'\rangle.r'(m).\bar{r}\langle x * m\rangle \end{aligned}$$

In general $\mathsf{Spec} \ncong \mathsf{Imp}$, because f is not private, therefore some other process could intercept it. This is solved with i/o types, which make it possible to ensure that the f channel is only ever used as an output.

Everything we've done about reasoning about systems can be done in a typed domain.

2 Session types

This discussion on session types is based on *Secure Implementations for Typed Session Abstractions*, by R. Corin, P.M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer, published in CSF'07, and available at available at `http://dx.doi.org/10.1109/CSF.2007.29`. Slides are available at `http://msr-inria.inria.fr/projects/sec/sessions/slides.pdf`.

The idea is to use the π-calculus like a syntax to describe protocols. They use the π-calculus sessions to make a compiler such that, no matter what happens in the distributed system, the protocol is not violated. Even if the attacker takes over some subset of the nodes, he will not be able to interfere with the protocol. The worst the attacker could do is keeping any messages from going out.

They propose a general purpose language for expressing distributed systems with session types, where the compiler guarantees certain properties of the system, and where channel actions cannot just be sent naively.

Well-typed programs always play their roles: through compilation and cryptography, there is always a way of detecting when things go wrong. If I am a node in the system, and I get a message from a remote node, then I have some notion of what I can trust from that remote node. It might be that the remote node is being totally bogus, but that can be detected.

The language is an extension of F#, a Microsoft-developed variant of ML. The output of the compiler is either concrete code full of cryptography, of a symbolic model of the code.

Session types are basically a state machine. The language basically encodes a finite state machine in the π-calculus.

It is possible to write a protocol that is inherently vulnerable. The language makes it possible to detect these and get rid of them. For example, if you have an offer and and a reject symbol going to two different places, a compromised node might send both, and confuse the global state. The paper gives a mechanical transformation from unsafe sessions to more chatty but safe sessions. It also has a pretty good notion of what information you have to maintain to keep the system integrity. You only need to keep a log of the last few entries, known as *visibility* in the paper: this is much more compact than maintaining the entire thing.

Theorem 1. *No attacker can compromise the system such that it violates the protocol integrity.*

Next time we will see the join calculus.