Using Temporal Session Types to Analyze Time Complexities of Concurrent Programs

Joseph Moonan Walbran

Division of Science and Mathematics
University of Minnesota Morris
Morris, Minnesota, USA

18 November 2021

Problem

Suppose you are the Morris Telegraph Company

You have a network of telegraph stations

- Each station sends, receives, and processes Morse code messages in various ways

How long does it take a message to get through the network?

Problem

Concurrent program —
A program with several parts running at the same time

Hard to tell how long a concurrent program will take to execute

- Many pieces interacting
  - Some can run in parallel
  - Some need to wait until other pieces are ready
- Tricky to figure out the timing of interactions

Need: a good way to work out the timing between pieces of concurrent programs

Solution

Das et al. (2018) give a way to analyze **the timing of interactions between parts of a program**.

- Big idea: adding timing information to datatypes
- Specifically, they introduce *temporal session types*
  - for describing channels of communication
  - “message rate” becomes part of the type system

Layers

- π-calculus
  - A simple, minimalist concurrent programming language
  - From 1992, developed by Milner et al.
- Session types
  - A way to typecheck π-calculus
  - From 1993, developed by Kohei Honda
- Temporal session types
  - Session types extended with timing information
  - From 2018, developed by Das et al.

Outline

- π-calculus
- Session types
- Temporal session types
- Conclusion

π-calculus

Session types

Temporal session types

π-calculus: motivation

Need a way to represent concurrent programs

Want it to be:

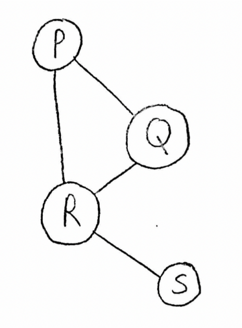
- general
- precise
- small

Several such systems exist

Das et al. use π*-calculus*

- From the early 1990s
- Good at modeling **independent processes that send messages** back and forth
  - servers on the web
  - processes in Unix

π-calculus: what is it?



What is π-calculus?

A very small programming language

Three main constructs:

- Processes
  - Like small programs
- Channels
- Labels
  - The data you send over channels
  - A finite set of symbols
  - E.g., for Morse code: $\{$ DOT, DASH, NEXT_LETTER, \$ $\}$

π-calculus: elementary operations

Defining a process

```
processName(channel1, channel2) =
   operation1;
   operation2;
   operation3;
   operation4
```

Spawning a process

```
a() =
   a();
   a()
```

π-calculus: elementary operations

Sending a label

```
sayHi(outChannel) =
  outChannel.DOT;
  outChannel.DOT;
  outChannel.DOT;
  outChannel.DOT;
  outChannel.NEXT_LETTER;
  outChannel.DOT;
  outChannel.DOT;
  outChannel.$;
  close outChannel
```

H I

•••• ••

The message “Hi” in Morse code

Receiving a label

```
invert(inChannel, outChannel) =
  case inChannel
  | DOT =>
      outChannel.DASH;
      invert(inChannel, outChannel)
  | DASH =>
      outChannel.DOT;
      invert(inChannel, outChannel)
  | NEXT_LETTER =>
      outChannel.NEXT_LETTER;
      invert(inChannel, outChannel)
  | $ =>
      outChannel.$;
      wait inChannel;
      close outChannel
```

π-calculus: what's next?

We have: a simple way to describe concurrent programs

Goal: figure out the times at which messages are sent over a channel

Next step: describe the interactions over a given channel

- This description is called the *session type* of the channel
- Eventually will include timing information
- But for now just says who's sending messages to whom

π-calculus

Session types

Temporal session types

Session types: what are they?

Session types are:

- Datatypes describing channels
- More complicated than `int` or `string`
- But they serve the same purpose:
  - Says what operations does this channel supports
- The typechecker makes sure you're using each channel correctly

Session types: what are they?

Session types describe **the structure of how two processes interact over a channel**

- E.g., “send two labels, then receive one label, then repeat”.

Like a very small network protocol

- A contract for how processes should talk to each other
- The typechecker makes sure you follow that contract

Session types: how are they written?

Internal choice

$$\oplus\{\ \mathtt{A} : T_1,\ \mathtt{B} : T_2\ \}$$

is a type meaning we can choose to:

- send label `A` and then do an action of type T_1
- send label `B` and then do an action of type T_2

Closing a channel

$$\mathbf{1}$$

is a type saying to close the channel immediately.

Session types: example

```
sayHi(output) =
  output.DOT;
  output.DOT;
  output.DOT;
  output.DOT;
  output.NEXT_LETTER;
  output.DOT;
  output.DOT;
  output.$;
  close output
```

$$
\textit{sendMessage} = \oplus\{
\quad \mathtt{DOT} : \textit{sendMessage},
\quad \mathtt{DASH} : \textit{sendMessage},
\quad \mathtt{NEXT\_LETTER} : \textit{sendMessage},
\quad \$: \mathbf{1}
\}
$$

The channel `output` has type *sendMessage*

Session types: how are they written?

External choice

$\&\{$

$\quad \mathtt{A} : T_1,$

$\quad \mathtt{B} : T_2$

$\}$

means we should be prepared to either:

- receive label $\mathtt{A}$ and then do an action of type T_1
- or receive label $\mathtt{B}$ and then do an action of type T_2

Waiting for a channel to close

$$\perp$$

means “wait for the other person to close this channel”.

Session types: example

```
invert(input, output) =
  case input
  | DOT =>
    output.DASH;
    invert(input, output)
  | DASH =>
    output.DOT;
    invert(input, output)
  | NEXT_LETTER =>
    output.NEXT_LETTER;
    invert(input, output)
  | $ =>
    output.$;
    wait input;
    close output
```

$$\mathit{sendMessage} = \oplus\{ \\ \quad \texttt{DOT} : \mathit{sendMessage}, \\ \quad \texttt{DASH} : \mathit{sendMessage}, \\ \quad \texttt{NEXT\_LETTER} : \mathit{sendMessage}, \\ \quad \$: \mathbf{1} \\ \}$$

$$\mathit{readMessage} = \&\{ \\ \quad \texttt{DOT} : \mathit{readMessage}, \\ \quad \texttt{DASH} : \mathit{readMessage}, \\ \quad \texttt{NEXT\_LETTER} : \mathit{readMessage}, \\ \quad \$: \bot \\ \}$$

The channel `input` has type *readMessage*
The channel `output` has type *sendMessage*

Session types: non-example

```
invert(input, output) =
  case input
  | DOT =>
      output.DASH;
      invert(input, output)
  | NEXT_LETTER =>
      output.NEXT_LETTER;
      invert(input, output)
  | $ =>
      output.$;
      wait input;
      close output
```

$$\mathit{sendMessage} = \oplus\{\ \texttt{DOT} : \mathit{sendMessage},\ \texttt{DASH} : \mathit{sendMessage},\ \texttt{NEXT\_LETTER} : \mathit{sendMessage},\ \$: \mathbf{1}\ \}$$

$$\mathit{readMessage} = \&\{\ \texttt{DOT} : \mathit{readMessage},\ \texttt{DASH} : \mathit{readMessage},\ \texttt{NEXT\_LETTER} : \mathit{readMessage},\ \$: \bot\ \}$$

The channel `input` **does not** have type *readMessage*
The channel `output` has type *sendMessage*

Session types: what's next?

We have: the structure of interactions over a channel

Goal: figure out the times at which those interactions happen

Next step: add timing information to session types

π-calculus

Session types

Temporal session types

Temporal session types: what are they?

New session type: **delay**

$$\circ T$$

means that an action of type T will occur after one second

Temporal session types: example

Each I/O operation takes one second

By convention, delays occur after the operation

```
sayHi(output) =
  output.DOT;               (delay 1)
  output.DOT;               (delay 1)
  output.DOT;               (delay 1)
  output.DOT;               (delay 1)
  output.NEXT_LETTER;       (delay 1)
  output.DOT;               (delay 1)
  output.DOT;               (delay 1)
  output.$;                 (delay 1)
  close output              (delay 1)
```

$$
\mathit{timedSendMessage} = \oplus\{
\quad \texttt{DOT} : \circ\mathit{timedSendMessage},
\quad \texttt{DASH} : \circ\mathit{timedSendMessage},
\quad \texttt{NEXT\_LETTER} : \circ\mathit{timedSendMessage},
\quad \$: \circ\mathbf{1}
\}
$$

The channel `output` has type *timedSendMessage*

One "$\circ$" in *timedSendMessage*, so
message rate = one label per second

Temporal session types: example

What about `invert(input, output)`?

```
invert(input, output) =
  case input
  | DOT =>
      output.DASH;
      invert(input, output)
  | DASH =>
      output.DOT;
      invert(input, output)
  | NEXT_LETTER =>
      output.NEXT_LETTER;
      invert(input, output)
  | $ =>
      output.$;
      wait input;
      close output
```

- Slightly harder
- Issue: timing of `output` will depend on timing of `input`
- But, if we know the timing of `input`, we can find the timing of `output`

Temporal session types: example

```
invert(input, output) =
  case input
  | DOT =>
      output.DASH;
      invert(input, output)
  | DASH =>
      output.DOT;
      invert(input, output)
  | NEXT_LETTER =>
      output.NEXT_LETTER;
      invert(input, output)
  | $ =>
      output.$;
      wait input;
      close output
```

Suppose we know that `input` has this temporal session type:

$$
\begin{aligned}
&\mathit{readMessageSlowly} = \&\{\\
&\quad \texttt{DOT} : \circ^n \mathit{readMessageSlowly},\\
&\quad \texttt{DASH} : \circ^n \mathit{readMessageSlowly},\\
&\quad \texttt{NEXT\_LETTER} : \circ^n \mathit{readMessageSlowly},\\
&\quad \$: \circ^n \bot\\
&\}
\end{aligned}
$$

Here, $\circ^n$ is a delay of n seconds.

Temporal session types: example

```
invert(input, output) =
  case input
  | DOT =>                     (delay 1)
      output.DASH;             (delay 1)
                               (delay k)
      invert(input, output)
  | DASH =>                    (delay 1)
      output.DOT;              (delay 1)
                               (delay k)
      invert(input, output)
  | NEXT_LETTER =>             (delay 1)
      output.NEXT_LETTER;      (delay 1)
                               (delay k)
      invert(input, output)
  | $ =>                       (delay 1)
      output.$;                (delay 1)
                               (delay k)
      wait input;              (delay 1)
      close output             (delay 1)
```

$$
\begin{aligned}
&\textit{readMessageSlowly} = \&\{\\
&\quad \texttt{DOT} : \circ^{n}\,\textit{readMessageSlowly},\\
&\quad \texttt{DASH} : \circ^{n}\,\textit{readMessageSlowly},\\
&\quad \texttt{NEXT\_LETTER} : \circ^{n}\,\textit{readMessageSlowly},\\
&\quad \$: \circ^{n}\,\bot\\
&\}
\end{aligned}
$$

Can sketch out where delays are:

- 1 second after each I/O operation
- k seconds where `invert` is idling
- Note: spawning a new process is instantaneous

Temporal session types: example

```
invert(input, output) =
  case input
  | DOT =>                      (delay 1)
      output.DASH;              (delay 1)
                                (delay k)
      invert(input, output)
  | DASH =>                     (delay 1)
      output.DOT;               (delay 1)
                                (delay k)
      invert(input, output)
  | NEXT_LETTER =>              (delay 1)
      output.NEXT_LETTER;       (delay 1)
                                (delay k)
      invert(input, output)
  | $ =>                        (delay 1)
      output.$;                 (delay 1)
                                (delay k)
      wait input;               (delay 1)
      close output              (delay 1)
```

$$
\begin{aligned}
&\mathit{readMessageSlowly} = \&\{ \\
&\quad \texttt{DOT} : \circ^{n}\mathit{readMessageSlowly}, \\
&\quad \texttt{DASH} : \circ^{n}\mathit{readMessageSlowly}, \\
&\quad \texttt{NEXT\_LETTER} : \circ^{n}\mathit{readMessageSlowly}, \\
&\quad \$: \circ^{n}\bot \\
&\}
\end{aligned}
$$

Delay between successive reads must equal n

Solve for k in terms of n:

$$1 + 1 + k = n$$
$$k = n - 2$$

Temporal session types: example

```
invert(input, output) =
  case input
  | DOT =>                    (delay 1)
      output.DASH;            (delay 1)
                              (delay k)
      invert(input, output)
  | DASH =>                   (delay 1)
      output.DOT;             (delay 1)
                              (delay k)
      invert(input, output)
  | NEXT_LETTER =>            (delay 1)
      output.NEXT_LETTER;     (delay 1)
                              (delay k)
      invert(input, output)
  | $ =>                      (delay 1)
      output.$;               (delay 1)
                              (delay k)
      wait input;             (delay 1)
      close output            (delay 1)
```

$$k = n - 2$$

What next?

k can't be a negative amount of time. So,

$$\begin{aligned} k &\geq 0 \\ n - 2 &\geq 0 \\ n &\geq 2. \end{aligned}$$

So, there must be at least 2 seconds between inputs.

(Otherwise, `invert` won't be able to read fast enough.)

Temporal session types: example

```
invert(input, output) =
  case input
  | DOT =>                  (delay 1)
    output.DASH;            (delay 1)
                            (delay k)
    invert(input, output)
  | DASH =>                 (delay 1)
    output.DOT;             (delay 1)
                            (delay k)
    invert(input, output)
  | NEXT_LETTER =>          (delay 1)
    output.NEXT_LETTER;     (delay 1)
                            (delay k)
    invert(input, output)
  | $ =>                    (delay 1)
    output.$;               (delay 1)
                            (delay k)
    wait input;             (delay 1)
    close output            (delay 1)
```

What next?

Find the temporal session type of *output*

- Initial delay: 1 second
- Delay between writes:

$$1 + k + 1 = n \text{ seconds}$$

Temporal session type:
$\circ\mathit{sendMessageSlowly}$

where

$$\mathit{sendMessageSlowly} = \oplus\{ \\ \texttt{DOT} : \circ^n \mathit{sendMessageSlowly}, \\ \texttt{DASH} : \circ^n \mathit{sendMessageSlowly}, \\ \texttt{NEXT\_LETTER} : \circ^n \mathit{sendMessageSlowly}, \\ \$: \circ^n \mathbf{1} \\ \}$$

Temporal session types: example

```
invert(input, output) =
  case input
  | DOT =>                        (delay 1)
      output.DASH;                (delay 1)
                                  (delay k)
      invert(input, output)
  | DASH =>                       (delay 1)
      output.DOT;                 (delay 1)
                                  (delay k)
      invert(input, output)
  | NEXT_LETTER =>                (delay 1)
      output.NEXT_LETTER;         (delay 1)
                                  (delay k)
      invert(input, output)
  | $ =>                          (delay 1)
      output.$;                   (delay 1)
                                  (delay k)
      wait input;                 (delay 1)
      close output                (delay 1)
```

In summary:

Maximum message rate of `input`:
  2 labels/second

Message rate of `output`:
  n labels/second
  (same as `input`)

Latency of `output`:
  1 second

Conclusion

What do we have?

1. A way to find the timing of interactions between parts of a concurrent program
2. A way to mechanically verify that the timing is correct
   - Given:
     - π-calculus source code
     - a temporal session type for each channel

   a typechecker can verify the channels actually have the session types indicated.

What we don't have (yet):

1. Implementations
2. A way to deduce temporal session types from source code

Questions?

[1] L. Caires.
Types and logic, concurrency and non-determinism.
Technical Report MSR-TR-2014-104, Microsoft Research, September 2014.

[2] L. Caires and F. Pfenning.
Session types as intuitionistic linear propositions.
In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 222–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] A. Das, J. Hoffmann, and F. Pfenning.
Parallel complexity analysis with temporal session types.
Proc. ACM Program. Lang., 2(ICFP), July 2018.

[4] D. Garg and F. Pfenning.
Type-directed concurrency.
In M. Abadi and L. de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, pages 6–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] K. Honda.
Types for dyadic interaction.
In E. Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[6] R. Milner, J. Parrow, and D. Walker.
A calculus of mobile processes, I.
Information and Computation, 100(1):1–40, 1992.