Overview

Safety

Concurrency

Pi calculus

Derived forms
Safety

Most implementatations don’t code up booleans / pairs / numbers etc. as functions!

They use efficient implementations (native binary, or pointers).

It’s now important to avoid bad states such as:

\[
\text{if } (\lambda x. M) \{ M \} \text{ else } \{ N \}
\]

True (\(M\))
False (\(M\))

How can we formally define this problem? How could we solve it?
Safety

Problem definition: add native booleans, plus a bad state.

\[ M ::= \]
\[ x \]
\[ M \ N \]
\[ \lambda x. M \]
\[ \text{True} \]
\[ \text{False} \]
\[ \text{if (} \ L \ \text{)} \{ M \} \text{ else } \{ N \} \]
\[ \text{FAIL} \]

Note these are now native, not derived forms!
Safety

Add reduction rules for good states:

\[
\begin{align*}
&\text{if ( True ) } \{ M \} \text{ else } \{ N \} \rightarrow M \\
&\text{if ( False ) } \{ M \} \text{ else } \{ N \} \rightarrow N
\end{align*}
\]

and bad states:

\[
\begin{align*}
&\text{if ( } \lambda x. M \text{ ) } \{ M \} \text{ else } \{ N \} \rightarrow \text{FAIL} \\
&\text{True ( } M \text{ ) } \rightarrow \text{FAIL} \\
&\text{False ( } M \text{ ) } \rightarrow \text{FAIL}
\end{align*}
\]

Define safety as:

\[
M \text{ is safe whenever } \forall N . M \rightarrow^* N \text{ implies } \text{FAIL} \notin N
\]
Safety

Define a type system with types:

\[ \sigma, \tau ::= \]
\[ \text{bool} \]
\[ \sigma \rightarrow \tau \]

Type system is of the form:

\[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \]

meaning:

If each variable \( x_i \) has type \( \tau_i \)
then program \( M \) has type \( \tau \)

As shorthand, write \( \Gamma \) for \( (x_1 : \tau_1, \ldots, x_n : \tau_n) \).
Safety

What should the types for these be?

Not = \lambda x. \text{if} (x) \{ \text{False} \} \text{else} \{ \text{True} \}
And = \lambda x. \lambda y. \text{if} (x) \{ y \} \text{else} \{ \text{False} \}
Or = \lambda x. \lambda y. \text{Not} (\text{And} (\text{Not} x) (\text{Not} y))
Safety

Type rules for functions:

1. If $x : \tau \in \Gamma$ then $\Gamma \vdash x : \tau$.
2. If $\Gamma, x : \sigma \vdash M : \tau$ then $\Gamma \vdash \lambda x. M : \sigma \rightarrow \tau$.
3. If $\Gamma \vdash M : \sigma \rightarrow \tau$, and $\Gamma \vdash N : \sigma$, then $\Gamma \vdash M \ N : \tau$.

Similar rules for booleans.

No rules for FAIL!
Safety

Check to make sure we get the right types for these:

Not = λx. if (x) { False } else { True }
And = λx. λy. if (x) { y } else { False }
Or = λx. λy. Not (And (Not x) (Not y))
Safety

Types imply safety:

1. If $\Gamma \vdash M : \tau$ then $\text{FAIL} \notin M$
2. If $\Gamma \vdash M : \tau$ and $M \rightarrow N$ then $\Gamma \vdash N : \tau$.

Plug these two together, and we get:

If $\Gamma \vdash M : \tau$ then $M$ is safe.

Hooray, safety!
Concurrency

What is concurrency? What makes concurrent programming different from sequential programming?

What are the core components of a concurrent language?
Concurrency

Possible inter-thread communication mechanisms:

- Read/write to shared memory.
- Locks.
- Monitors (a.k.a. wait/notify).
- Buffered streams.
- Unbuffered streams.
- ...

Which of these does Java support? Which should we include in a foundational calculus?
Pi calculus

History:

Models of concurrency (late 1970s-80s): Communicating Sequential Processes (Hoare), Petri Nets (Petri), Calculus of Communicating Systems (Milner)...

Additional features to model dynamic network topologies (late 1980s-90s): Pi-calculus (Milner), Higher order pi-calculus (Sangiorgi), Ambients (Cardelli and Gordon)...

Pi-calculus is a minimal model, but with ‘enough stuff’ to perform interesting computation (e.g. is more powerful than the lambda-calculus).
Pi calculus

First shot:

\[ P, Q, R ::= 0 \]
\[ \text{out } x \ y; \ P \]
\[ \text{in } x \ (y); \ P \]
\[ P | Q \]

What are these?

Note that Pierce uses ‘overbar’ for ‘out’, which is not very HTML friendly!
Pi calculus

Example programs:

1. out stdout hello; out stdout world; 0
2. in stdin (name); out stdout hello; out stdout name; 0
3. (out c fred; 0) | (in c (name); out d name; 0)
4. (out c fred; out c wilma; 0) | (in c (x); out d x; 0) | (in c (y); out e y; 0)
5. (out c fred; in d x; 0) | (in c (y); out d wilma; 0)
6. (in d x; out c fred; 0) | (in c (y); out d wilma; 0)
7. (out c fred; in d (x); 0) | (out d wilma; in c (y); 0)

What do these programs do?
Pi calculus

Dynamic semantics is defined in two steps...

Structural congruence $P \equiv Q$ is generated by:

1. If $P =_{\alpha} Q$ then $P \equiv Q$.
2. $P | Q \equiv Q | P$.
3. $(P | Q) | R \equiv P | (Q | R)$.

Dynamic semantics $P \rightarrow Q$ is generated by:

1. $(\text{out } x y; P) | (\text{in } x (z); Q) \rightarrow P | Q[y/z]$
2. If $P \rightarrow Q$ then $P | R \rightarrow Q | R$.
3. If $P \equiv \rightarrow \equiv Q$ then $P \rightarrow Q$. 
Pi calculus

Examples:

1. (out c fred; 0) | (in c (name); out d name; 0)
2. (out c fred; out c wilma; 0) | (in c (x); out d x; 0) | (in c (y); out e y; 0)
3. (out c fred; in d x; 0) | (in c (y); out d wilma; 0)
4. (in d x; out c fred; 0) | (in c (y); out d wilma; 0)
5. (out c fred; in d (x); 0) | (out d wilma; in c (y); 0)
Pi calculus

Missing feature: recursion/looping/infinite behavior.

Minimal solution replication: $!P \text{ acts like } P \mid P \mid P \mid ...$

Examples:

1. $!\text{in } x (z); \text{ out } y z; 0$
2. $\text{out acquire lock; 0 } \mid !\text{in release (lock); out acquire lock; 0}$

Replicated input $!\text{in accept (socket); P acts a lot like a multithreaded server (Java ServerSocket).}$

Dynamic semantics just given by:

$$!P \equiv P \mid !P$$
Pi calculus

Last missing feature: create new channels.

Minimal solution \textit{channel generation}: \texttt{new (x); P} generates a fresh channel for use in \texttt{P}.

Example:

1. \texttt{new (c); out x c; in c (y_1); .. in c (y_n); P}
2. \texttt{in x (c); out c z_1; .. out c z_n; Q}

Put these in parallel, and what happens?

New channel generation acts a lot like \texttt{new object generation / new key generation / new nonce generation / ...}

Dynamic semantics just given by:

\[
\texttt{(new (x); P) | Q \equiv new (x); (P | Q)} \quad \text{(as long as} \ x \not\in Q)\]

If \texttt{P \rightarrow Q} then \texttt{new (x); P \rightarrow new (x); Q}.
Derived forms

Multiple messages:

\[
\text{in } x (y_1, \ldots, y_n); P = \text{new } c; \text{ out } x \ c; \text{ in } c \ (y_1); \ldots \text{ in } c \ (y_n); P
\]

\[
\text{out } x (z_1, \ldots, z_n); Q = \text{in } x \ (c); \text{ out } c \ z_1; \ldots \text{ out } c \ z_n; Q
\]

Let’s double check:

\[
( \text{in } x (y_1, \ldots, y_n); P | \text{out } x (z_1, \ldots, z_n); Q ) \rightarrow^* \text{P}[z_1/y_1, \ldots, z_n/y_n] | Q
\]
Derived forms

Oops, it’s not quite true, we have to do a bit of garbage collection:

new (c); $P =_{gc} P$ \hspace{1cm} (when $c \notin P$)

new (c); in c (x); $P =_{gc} 0$

new (c); !in c (x); $P =_{gc} 0$

new (c); out c x; $P =_{gc} 0$

new (c); !out c x; $P =_{gc} 0$

$P | 0 =_{gc} P$

Let’s double check:

$\left( \text{in } x (y_1, \ldots, y_n); P | \text{out } x (z_1, \ldots, z_n); Q \right)$

$\rightarrow^* =_{gc} P[z_1/y_1, \ldots, z_n/y_n] | Q$
Revisit garbage collection later...
Derived forms

Booleans:

True(b)
   = !in b (x, y); out x (); 0

False(b)
   = !in b (x, y); out y (); 0

if (b) { P } else { Q }
   = new (t); new (f); ( out b (t, f); 0 | in t (); P | in f (); Q )

Sanity check:

True(b) | if (b) { P } else { Q }
   →* =_gc True(b) | P
Derived forms

Can also code integers, linked lists, ...

and the lambda-calculus...

and concurrency controls like mutexes, mvars, ivars, buffers, etc.
Derived forms

Correctness of garbage collection:

\[
\begin{align*}
\text{If } P &=_{gc} Q \text{ and } P \rightarrow P' \\
\text{then } P' &=_{gc} Q' \text{ and } Q \rightarrow Q'
\end{align*}
\]

Phew!
Next week

Homework sheet 3.

Calculi for cryptographic protocols: spi-calculus.