Independence and Concurrent Separation Logic

Jonathan Hayman
joint work with Glynn Winskel

Computer Laboratory
University of Cambridge

13th August 2006
This talk:

- Give a net semantics to a programming language
- Apply it to study the semantics of a program logic: Concurrent separation logic
Independence models help alleviate problems with giving a standard, ‘interleaved’ semantics to parallel processes:

- Atomicity: \( \alpha \parallel \beta \) or \( (\alpha_1; \alpha_2) \parallel \beta \)
- Genuine concurrent execution
- Unnecessary interleavings

**Concurrent separation logic:**

- How is spatial separation related to independence?
- The logic discriminates between parallel composition and interleaved expansion:

\[
\begin{align*}
\not\vdash & \{ \ell \mapsto 0 \} \quad \ell := 1 \parallel \ell := 2 \\
\vdash & \{ \ell \mapsto 0 \} \quad (\ell := 1; \ell := 2) + (\ell := 2; \ell := 1) \quad \{ \ell \mapsto 1 \lor \ell \mapsto 2 \}
\end{align*}
\]
A Petri net consists of:

- **Conditions** (places / circles)
- **Events** (transitions / rectangles)
- An initial *marking* (subset of conditions)

Events have *preconditions* and *postconditions*, determining whether they are able to fire in a marking.

Two events are said to be *independent* if they operate on disjoint sets of conditions.
A Petri net consists of:
- **Conditions** (places / circles)
- **Events** (transitions / rectangles)
- An initial *marking* (subset of conditions)

Events have *preconditions* and *postconditions*, determining whether they are able to fire in a marking.

Two events are said to be *independent* if they operate on disjoint sets of conditions.
Petri nets

- A Petri net consists of:
  - *Conditions* (places / circles)
  - *Events* (transitions / rectangles)
  - An initial *marking* (subset of conditions)

- Events have *preconditions* and *postconditions*, determining whether they are able to fire in a marking

- Two events are said to be *independent* if they operate on disjoint sets of conditions
A Petri net consists of:
- **Conditions** (places / circles)
- **Events** (transitions / rectangles)
- An initial *marking* (subset of conditions)

Events have *preconditions* and *postconditions*, determining whether they are able to fire in a marking.

Two events are said to be *independent* if they operate on disjoint sets of conditions.
• Concurrent separation logic uses separating conjunction to develop Owicki-Gries proof rules to deal with pointer-manipulating programs

• Want a net semantics for terms of the following programming language

\[
t := \alpha \mid t; t \mid t \parallel t \mid \text{while } b \text{ do } t \text{ od} \\
\mid \text{alloc}(\ell) \mid \text{dealloc}(\ell) \mid \text{with } r \text{ do } t \text{ od} \\
\mid \ldots
\]

• \(\alpha\) ranges over commands that affect the memory in which the programs execute (e.g. assignments)
Net semantics

- Within nets formed, shall distinguish two forms of condition: control conditions and state conditions.
- The semantics $N[t]$ of a process consists of:
  - Events
  - Initial marking of control conditions
  - Terminal marking of control conditions
- State conditions are:
  - $(\ell, v)$ for each $\ell \in \text{Loc}$ and $v \in \text{Val}$. Marking $H$
  - $\text{curr}(\ell)$ for each $\ell \in \text{Loc}$. Marking $L$
  - $r$ for each $r \in \text{Res}$. Marking $R$
- The marking of state conditions must be consistent:
  - $H$ is a partial function defined at the locations current in $L$
Action

State conditions

Control conditions

$Ic(\alpha)$ $Tc(\alpha)$

$(\ell, 1)$ $(\ell, 2)$ $(\ell, 3)$ $(\ell, 4)$
Action

\[ I_c(\alpha) \quad T_c(\alpha) \]

State conditions

Control conditions
Action

$(\ell, 1)$  $(\ell, 2)$  $(\ell, 3)$  $(\ell, 4)$

**State conditions**

$\text{Ic}(\alpha)$  $\text{Tc}(\alpha)$

**Control conditions**
alloc(\ell):

- Make an arbitrary non-current location current, assign an (arbitrary) initial value to it, and make \ell point to the new location.
- If \ell initially holds v and run alloc(\ell):

\[
\begin{array}{c|c}
\ell & v \\
\end{array}
\]
alloc(ℓ):

- Make an arbitrary non-current location current, assign an (arbitrary) initial value to it, and make ℓ point to the new location.
- If ℓ initially holds v and run alloc(ℓ):

```
  ℓ  k  v'
```


Allocation

\[ \text{Ic}(\text{alloc}(\ell)) \quad \text{Tc}(\text{alloc}(\ell)) \]
Allocation

\[ T_c(\text{alloc}(\ell)) \]

\[ I_c(\text{alloc}(\ell)) \]

\[ \text{curr}(k) \]

\[ \text{curr}(m) \]
$Tc(\text{alloc}(\ell)) \cap Ic(\text{alloc}(\ell))$
Allocation

\[ T_c(\text{alloc}(\ell)) \]

\[ I_c(\text{alloc}(\ell)) \]

\[ \text{curr}(k) \]

\[ \text{curr}(m) \]

\( \ell, 0 \) \hspace{1cm} \( \ell, 1 \) \hspace{1cm} \( \ell, k \) \hspace{1cm} \( \ell, m \) \hspace{1cm} \( k, 1 \) \hspace{1cm} \( m, 2 \)
Structural connectives

- Operations on control part of net allow us to form the usual operations such as:
- Parallel composition:
Operations on control part of net allow us to form the usual operations such as:

- **Sequential composition**

![Diagram](image-url)
Operational semantics

**Theorem**

The net semantics corresponds to the operational semantics.
- Key judgement $\Gamma \vdash \{\varphi\} t \{\psi\}$
- $\varphi$ is an assertion about the heap (assignment of values to locations):
  $H \models \ell \mapsto v$ iff $H = \{(\ell, v)\}$
  $H \models \text{emp}$ iff $H = \emptyset$
  $H \models \varphi_1 \ast \varphi_2$ iff there exist $H_1, H_2$ s.t. $H_1$ disjoint $H_2$
  and $H = H_1 \cup H_2$ and $H_1 \models \varphi_1$ and $H_2 \models \varphi_2$
  with usual $\neg$, $\land$, $\lor$, etc.
- Meaning (approximate):
  *Whenever $t$ runs in a heap that has a part satisfying $\varphi$, the resulting heap has a part satisfying $\psi$. Moreover, $t$ never accesses locations outside the identified part of the heap.*

Example: $\Gamma \vdash \{\ell \mapsto 0\} \ell := 1 \{\ell \mapsto 1\}$

Non-example: $\Gamma \not\vdash \{k \mapsto 0\} \ell := 1 \{k \mapsto 1\}$
Concurrent separation logic

- Permits a simple rule for parallel composition:

\[
\Gamma \vdash \{\varphi_1\} t_1 \{\psi_1\} \quad \Gamma \vdash \{\varphi_2\} t_2 \{\psi_2\} \\
\Gamma \vdash \{\varphi_1 \ast \varphi_2\} t_1 \| t_2 \{\psi_1 \ast \psi_2\}
\]

- Example: \(\Gamma \vdash \{\ell \mapsto 0 \ast k \mapsto 0\} \ell := 1 \| k := 2 \{\ell \mapsto 1 \ast k \mapsto 2\}\)

  Non-example: \(\Gamma \not\vdash \{\ell \mapsto 0 \ast k \mapsto 0\} \ell := 1 \| \ell := 2 \{\top\}\)

- Critical regions: (\(\chi\) precise)

\[
\Gamma \vdash \{\varphi \ast \chi\} t \{\psi \ast \chi\} \\
\Gamma, r:\chi \vdash \{\varphi\} \text{ with } r \text{ do } t \text{ od } \{\psi\}
\]

- Allows “transfer” of heap:
  Let the invariant \(\chi \equiv k \mapsto 0 \lor k \mapsto 1 \ast \ell \mapsto 0\).

  \[
  r:\chi \vdash \{\text{emp}\} \text{ with } r \text{ do if } k = 1 \text{ then } k := 0 \text{ od } \{\ell \mapsto 0\}
  \]

- Initially proved sound by Brookes (with ‘local’ semantics)
Want a definition of validity $\Gamma \models \{ \varphi \} t \{ \psi \}$ that allows us to prove the rule for parallel composition sound:

Whenever $t$ runs to completion in an environment where other well-behaved processes act from a heap where it owns locations satisfying $\varphi$, the resulting heap that the process owns satisfies $\psi$. Moreover, $t$ is well-behaved.

Place in parallel with an interference net containing events to simulate the permitted forms of interference.

Intuition presented by O’Hearn is ownership, which constrains interference.
Interference and synchronization

- Add events to the net to represent the execution of other processes.
- Possible behaviour of other processes depends on ‘ownership’ of the process, so add conditions $\omega_{\text{proc}}(l)$, $\omega_{\text{oth}}(l)$ and $\omega_{\text{inv}}(l)$ to the net.

Forms of interference
- Action on ‘other’ owned locations
- Enter and leave critical regions
- Allocation and deallocation

‘Synchronize’ the original events of the process so that they correspond to interference. ‘Violation’ if there is no synchronized event with concession but the unsynchronized event could occur. Interference net with synchronized events simulates the original net.
Interference and synchronization

Ownership conditions

\( \omega_{\text{oth}}(\ell) \)

\( \omega_{\text{inv}}(\ell) \)

\( \omega_{\text{proc}}(\ell) \)

\( (\ell, 0) \)

\( (\ell, 1) \)

\( r \)

\( \text{Ic}(\text{with } r \text{ do } t \text{ od}) \)
Interference and synchronization

Ownership conditions

\( \omega_{oth}(\ell) \)

\( \omega_{inv}(\ell) \)

\( \omega_{proc}(\ell) \)

\( \omega_{proc}(\ell, 0) \)

\( \omega_{proc}(\ell, 1) \)

\( r \)

\( Ic(\text{with } r \text{ do } t \text{ od}) \)
Interference and synchronization

Ownership conditions

\[ \omega_{\text{oth}}(\ell) \]

\[ \omega_{\text{inv}}(\ell) \]

\[ \omega_{\text{proc}}(\ell) \]

\[ (\ell, 0) \]

\[ (\ell, 1) \]

\[ r \]

\[ \text{Ic(with } r \text{ do } t \text{ od)} \]
Interference and synchronization

Ownership conditions

\[ \omega_{\text{oth}}(\ell) \]

\[ \omega_{\text{inv}}(\ell) \]

\[ \omega_{\text{proc}}(\ell) \]

\[ (\ell, 0) \]

\[ (\ell, 1) \]

\[ r \]

\[ \text{lc}(\text{with } r \text{ do } t \text{ od}) \]
Interference and synchronization

Ownership conditions

\[ \omega_{\text{oth}}(\ell) \]
\[ \omega_{\text{inv}}(\ell) \]
\[ \omega_{\text{proc}}(\ell) \]

\[ \text{Ic(with r do t od)} \]
Interference and synchronization

Ownership conditions

\[ \omega_{\text{oth}}(\ell) \]

\[ \omega_{\text{inv}}(\ell) \]

\[ \omega_{\text{proc}}(\ell) \]

\[ \omega_{\text{proc}}(\ell) \]

\[ (\ell, 0) \]

\[ (\ell, 1) \]

\[ r \]

\[ Ic(\text{with } r \text{ do } t \text{ od}) \]
Interference and synchronization

Ownership conditions

\[ \omega_{\text{oth}}(\ell) \]
\[ \omega_{\text{inv}}(\ell) \]
\[ \omega_{\text{proc}}(\ell) \]

\[ \omega(\ell, 0) \]
\[ (\ell, 1) \]
\[ r \]

\[ \text{Ic(with } r \text{ do } t \text{ od)} \]
Interference and synchronization

- Add events to the net to represent the execution of other processes
- Possible behaviour of other processes depends on ‘ownership’ of the process, so add conditions $\omega_{\text{proc}}(l)$, $\omega_{\text{oth}}(l)$ and $\omega_{\text{inv}}(l)$ to the net
- Forms of interference
  - Action on ‘other’ owned locations
  - Enter and leave critical regions
  - Allocation and deallocation
- ‘Synchronize’ the original events of the process so that they correspond to interference
- ‘Violation’ if there is no synchronized event with concession but the un-synchronized event could occur
- Interference net with synchronized events simulates the original net
Validity and soundness

A marking satisfies $\varphi$ in $\Gamma$ if the owned locations satisfy $\varphi$ and the invariants for available resources are met.

**Definition (Validity)**

$\Gamma \models \{ \varphi \} \ t \ \{ \psi \}$ if for any initial marking that satisfies $\varphi$ in $\Gamma$, in the net with interference events and synchronized events of $t$, no violating marking is reachable and any reachable terminal marking satisfies $\psi$ in $\Gamma$.

**Theorem (Soundness)**

$\Gamma \vdash \{ \varphi \} \ t \ \{ \psi \}$ implies $\Gamma \models \{ \varphi \} \ t \ \{ \psi \}$.
Enabled, control independent events arise only from parallel composition

**Theorem (Separation)**

Suppose that $\emptyset \vdash \{\varphi\} t \{\psi\}$ and that $\sigma$ is a state in which the heap satisfies $\varphi$. If $M$ is a marking reachable from $(Ic(t), \sigma)$ in $N[t]$ and $e_1$ and $e_2$ are control independent events then:

- If $M \xrightarrow{e_1} M_1$ and $M \xrightarrow{e_2} M_2$ then either $e_1$ and $e_2$ are independent or $e_1$ and $e_2$ compete to take a resource or to allocate the same location.

- If $M \xrightarrow{e_1} M_1 \xrightarrow{e_2} M'$ then either $e_1$ and $e_2$ are independent or $e_1$ releases a resource that $e_2$ takes or deallocates a location that $e_2$ allocates.
Independence

- Permits refinement of heap operations
- $\ell_0$ points to $\ell_1$, which is initially current

\[
\begin{align*}
t_1; \\
dealloc(\ell_0) & \quad | \quad alloc(k); \\
& \quad | \quad while ([k] \neq \ell_1) do \\
& \quad | \quad dealloc(k); alloc(k) \\
& \quad | \quad od; \\
& \quad | \quad t_2
\end{align*}
\]

- The loop exit event causally depends on a prior $alloc(k)$ event that allocates the location that was initially stored in $\ell$
- These events causally depend on the event $dealloc(\ell)$ in the other process, so $t_2$ can only run after $t_1$
- $\implies$ interaction through allocation seen in separation theorem
Conclusions

- Independence is a useful tool in describing features of programming languages and their logics.

Future work:
- Extended models to deal with e.g. name binding
- Liveness properties and refinement
- Modal logic using *