Verifying and timing concurrent instruments

Dominic Orchard
thanks to Sam Aaron for some of these slides & inspiration
A talk about…

• Verification
• Analysis

• Programming
• Music
• Outreach
• Education
Hello Sam. Do you feel it? I do. Creativity is rushing through your veins today!
has stopped whirring past on the
screen as you type it. This is a security
feature now be

Starting the
The stranger
and is full of strange arcane commands - much like
the contents of a magic spell. We therefore move to a more familiar graphical
environment with windows and menu bars that may perhaps feel a bit more comfortable. To
CODE LIKE A ROCKSTAR

LEARN TO CODE AND MAKE MUSIC WITH ((( Sonic π )))

VISIT RASPBERRYPI.ORG TO FIND OUT MORE!

DO IT! EDUCATEUR.COM / ROB-S-WILLIAMS.COM
SONIC PI
LIVE & CODING

MUSIC
TECHNOLOGY
ART

AT CAMBRIDGE JUNCTION
MON 28 JUL - FRI 01 AUG
JUNCTION.CO.UK
Sonic Pi v2.0

COMPETITION FOR SCHOOLS

Are you the next Daft Punk?

Make sure there’s more than a screwdriver in your sonic toolbox.

Sonic Pi is a way to get creative with music and computing. We’re hunting down the UK’s best young musical coding talent from outer space: is it you? Create a 2-minute or 200-line piece of music on the theme Space Wonders with Sonic Pi v2.0 on a Raspberry Pi, and you could be in with a chance of winning one of hundreds of Raspberry Pi kits for yourself and your school, along with workshops with musical artists Juneau Projects, and with live coder-musician Sam Aaron!

Check out raspberry.org/competitions/sonic-pi to find out how to enter.
Demo #1

play, sleep, loops, iteration
<table>
<thead>
<tr>
<th>Statement</th>
<th>Duration</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>play :C</td>
<td>$\Delta_a$</td>
<td>$\Delta_a$</td>
</tr>
<tr>
<td>play :E</td>
<td>$\Delta_b$</td>
<td>$\Delta_a + \Delta_b$</td>
</tr>
<tr>
<td>play :G</td>
<td>$\Delta_c$</td>
<td>$\Delta_a + \Delta_b + \Delta_c$</td>
</tr>
<tr>
<td>sleep 1</td>
<td></td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1$</td>
</tr>
<tr>
<td>play :F</td>
<td>$\Delta_d$</td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d$</td>
</tr>
<tr>
<td>play :A</td>
<td>$\Delta_e$</td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d + \Delta_e$</td>
</tr>
<tr>
<td>play :C</td>
<td>$\Delta_f$</td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d + \Delta_e + \Delta_f$</td>
</tr>
<tr>
<td>sleep 0.5</td>
<td></td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d + \Delta_e + \Delta_f + 0.5$</td>
</tr>
<tr>
<td>play :G</td>
<td>$\Delta_g$</td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d + \Delta_e + \Delta_f + 0.5 + \Delta_f$</td>
</tr>
<tr>
<td>play :B</td>
<td>$\Delta_h$</td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d + \Delta_e + \Delta_f + 0.5 + \Delta_f + \Delta_g$</td>
</tr>
<tr>
<td>play :D</td>
<td>$\Delta_i$</td>
<td>$\Delta_a + \Delta_b + \Delta_c + 1 + \Delta_d + \Delta_e + \Delta_f + 0.5 + \Delta_f + \Delta_g + \Delta_h$</td>
</tr>
</tbody>
</table>
Statement | Duration | Real Time | Virtual Time
---|---|---|---
Δₐ | play :C | 0 + Δₐ | 0
Δₐ | play :E | 0 + Δₐ + Δₐ | 0
Δₐ | play :G | 0 + Δₐ + Δₐ + Δₐ | 0
sleep 1 | | 1 | 1
Δₐ | play :F | 1 + Δₐ | 1
Δₐ | play :A | 1 + Δₐ + Δₐ | 1
Δₐ | play :C | 1 + Δₐ + Δₐ + Δₐ | 1
sleep 0.5 | | 1.5 | 1.5
Δₐ | play :G | 1.5 + Δₐ | 1.5
Δₐ | play :B | 1.5 + Δₐ + Δₐ | 1.5
Δₐ | play :D | 1.5 + Δₐ + Δₐ + Δₐ | 1.5
A formal semantics for sleep

- Abstract interpretation “time system”
- Denotational semantics (via monads)
- Prove “time safety” = prove semantics sound wrt. time system

“Temporal semantics for a live coding language”
Aaron, Orchard, Blackwell, FARM 2014
Simplified Sonic Pi v2.0 syntax

\[
\begin{align*}
P & ::= P; S \mid \emptyset \\
S & ::= E \mid v = E \\
E & ::= \text{sleep } \mathbb{R}_{\geq 0} \mid A^i \mid v
\end{align*}
\]
## Time system

\[
\begin{align*}
[\_\_\_]_v & : \text{virtual time} \\
[\emptyset]_v & = 0 \\
[P; v = E]_v & = [P]_v + [E]_v \\
[sleep \ t]_v & = t \\
[A^i]_v & = 0 \\
\end{align*}
\]

\[
\begin{align*}
[\_\_\_]_t & : \text{actual time} \\
[\emptyset]_t & \approx 0 \\
[P; \text{sleep } t]_t & \approx ([P]_v + t) \max [P]_t \\
[P; v = A^i]_t & \approx [P]_t + [A^i]_t \\
\end{align*}
\]

e.g. \quad P; \text{sleep 2} \quad \text{where} \quad [P]_t = 1, \quad [P]_v = 0

\[
\therefore \quad [P; \text{sleep 2}]_t = (0 + 2) \max 1 = 2 \\
[P; \text{sleep 2}]_v = 2
\]
Time system

\[\begin{align*}
[\text{sleep } t]_v &= t \\
[A^i]_v &= 0 \\
[P; v = E]_v &= [P]_v + [E]_v \\
\emptyset_v &= 0 \\
\emptyset_t &= 0 \\
[P; \text{sleep } t]_t &= ([P]_v + t) \max [P]_t \\
[P; v = A^i]_t &= [P]_t + [A^i]_t \\
\end{align*}\]

\[\begin{align*}
\text{e.g. } P; \text{sleep 1} & \quad \text{where } [P]_t = 2, \quad [P]_v = 0 \\
\therefore [P; \text{sleep 1}]_t &= (0 + 1) \max 2 = 2 \\
[P; \text{sleep 1}]_v &= 1
\end{align*}\]
**Time system**

\[ \begin{align*}
&\left[\text{\text{-}}\right]_v : \text{virtual time} & &\left[\text{\text{-}}\right]_t : \text{actual time} \\
&\left[\emptyset\right]_v = 0 & &\left[\emptyset\right]_t \approx 0 \\
&P; v = E]_v = [P]_v + [E]_v & &[P; \text{sleep} t]_t \approx ([P]_v + t) \max [P]_t \\
&\left[\text{sleep} t\right]_v = t & &[P; v = A^i]_t \approx [P]_t + [A^i]_t \\
&[A^i]_v = 0 & &
\end{align*} \]

**Lemma 1.** For any program \( P \) then \( [P]_t \geq [P]_v \).
Denotational semantics

- State for *virtual time*
- Read only *actual time* (updated from OS)
  
  \[
  \text{Temporal } a = (\text{start time, current time}) \rightarrow \frac{(\text{old vtime } \rightarrow (a, \text{new vtime}))}{[P]_{\text{top}} : \text{Temporal }()}\]

- Paper describes core monadic semantics with Haskell
  
  \[
  \text{Temporal } a = (\text{Time, Time}) \rightarrow \frac{(\text{VTime } \rightarrow \text{IO } (a, \text{VTime}))}{\quad}
  \]
Time safety

soundness of the denotational semantics

- wrt. virtual time

Lemma 2. \[ \text{runTime} \left[ P \right]_v = \left[ P \right]_v \]

- wrt. actual time (modulo constant sequential overhead)

Lemma 3. \[ \text{runTime} \left[ P \right]_t \approx \left[ P \right]_t \]
Demo #2

“live loops”, synchronisation

Two problems

- Thrashing (zero time sleep)
- Deadlock
Preventing thrasing

- Perform virtual time analysis

- Ensure that:

\[ [P]_v \neq 0 \]

- Current system needs extending to concurrent & higher-order setting
Extended time system

\[
\begin{align*}
[P]_v &= p \\
[E]_v &= e \\
[P; v = E]_v &= p + e \\
[\emptyset]_v &= 0 \\
[v]_v &= 0 \\
[sleep t]_v &= t \\
[if g_1 \text{ then } e_1 \text{ else } e_2]_v &= s \max t \\
[spawn \colon \text{name } P]_v &= 0 \\
[loop P]_v &= \infty \\
[n \times P]_v &= nt \\
\end{align*}
\]
Extended time system

\[
\begin{align*}
\text{[seq]} & \quad [P]_v = p & [E]_v = e & \quad [P; v = E]_v = p + e \\
\text{[sleep]} & \quad \text{[sleep } t\text{]}_v = t \\
\text{[cond]} & \quad [e_1]_v = s & [e_2]_v = t & \quad [\text{if } g_1 \text{ then } e_1 \text{ else } e_2]_v = s \ \text{max} \ t \\
\text{[spawn]} & \quad [P]_v = t & \quad [\text{spawn : name } P]_v = 0 \\
\text{[loop]} & \quad [P]_v = t & \quad [\text{loop } P]_v = \infty \\
\text{iterate} & \quad [P]_v = t & \quad [n \times \text{times } P]_v = nt & \quad n \text{ is constant}
\end{align*}
\]
Higher-order time system

• Need to associate (virtual) times to functions

\[
\begin{align*}
\text{[abs]} & \quad \frac{[\Gamma, v : \sigma \vdash e : \tau]_v = n}{[\Gamma \vdash \lambda v. e : \sigma \rightarrow^n \tau]_v = 0} \\
\text{[app]} & \quad \frac{[\Gamma \vdash e_1 : \sigma \rightarrow^n \tau]_v = n_1 \quad [\Gamma \vdash e_2 : \sigma]_v = n_2}{[\Gamma \vdash e_1 e_2 : \tau]_v = n + n_1 + n_2}
\end{align*}
\]

• Has the shape of traditional effect system

\[
\begin{align*}
\text{abs} & \quad \frac{\Gamma, x : \sigma \vdash e : \tau, F}{\Gamma \vdash \lambda x. e : \sigma \rightarrow^F \tau, \emptyset} \\
\text{app} & \quad \frac{\Gamma \vdash e_1 : \sigma \rightarrow^F \tau, G \quad \Gamma \vdash e_2 : \sigma, H}{\Gamma \vdash e_1 e_2 : \tau, F \sqcup G \sqcup H}
\end{align*}
\]
Higher-order & dependent

- Time may depend on parameters:

```ruby
define :foo do |t|
  sleep t
  ...
end
```

\[
\begin{align*}
[e_1 : (x : \sigma) \xrightarrow{f(x)} \tau]_v &= n_1 \\
[e_2 : \sigma]_v &= n_2 \\
[e_1 e_2 : \tau]_v &= f(e_2) + n_1 + n_2
\end{align*}
\]

- If implicit, dependent-type style formulation not necessary
Other benefits...

- IDE feedback on analysis to aid programming, e.g.,

<table>
<thead>
<tr>
<th>Command</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>play :C4</td>
<td>0.5</td>
</tr>
<tr>
<td>sleep 0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>play :es4</td>
<td>0.75</td>
</tr>
<tr>
<td>sleep 0.25</td>
<td>0.75</td>
</tr>
<tr>
<td>play :g4</td>
<td>0.90</td>
</tr>
<tr>
<td>sleep 0.15</td>
<td>0.90</td>
</tr>
<tr>
<td>play :as4</td>
<td>1.40</td>
</tr>
<tr>
<td>sleep 0.5</td>
<td>1.40</td>
</tr>
<tr>
<td>play :ds4</td>
<td>1.525</td>
</tr>
<tr>
<td>sleep 0.125</td>
<td></td>
</tr>
<tr>
<td>play :c5</td>
<td></td>
</tr>
</tbody>
</table>
Preventing deadlocks

- Session-type style analysis
  - cue ~ send
  - sync ~ receive

\[
\text{cue} : n : n! \quad \text{sync} : n : n?
\]

- Duality \(\Rightarrow\) compatibility \(\Rightarrow\) no deadlock
Cue/sync session types

<table>
<thead>
<tr>
<th>live_loop :foo do</th>
<th>live_loop :bar do</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync :B</td>
<td>sync :A</td>
</tr>
<tr>
<td>cue :A</td>
<td>cue :B</td>
</tr>
<tr>
<td>play :C3</td>
<td>play :E4</td>
</tr>
<tr>
<td>sleep 1.0</td>
<td>sleep 0.5</td>
</tr>
</tbody>
</table>

• B?.A!.0 ≠ dual(A?.B!.0)

• Here, incompatibility => deadlock
Cue/sync session types

live_loop :foo do
  cue :A
  sync :B
  play :C3
  sleep 1.0
end

live_loop :bar do
  cue :B
  sync :A
  play :E4
  sleep 0.5
end

: A!.B?.0  ;  B!.A?.0

A!.B?.0 ≠ dual(B!.A?.0)

• But this time there is no deadlock
Cue/sync session types

- cue is asynchronous
- use sub-typing on sessions

\[
A!.B?.0 \neq \text{dual}(B!.A?.0)
\]

\[
A!.P <: P.A!
\]

\[
A!.B?.0 <: B?.A!.0
\]

\[
B?.A!.0 = \text{dual}(B!.A?.0)
\]
Putting it together

• Virtual time and sessions as effects

\[ \Gamma \vdash e : \tau \mid vtime, session \]

\[ \tau \xrightarrow{t,S} \sigma \]
Putting it together

\[
\begin{align*}
\text{[seq]} & \quad \frac{\Gamma \vDash P \mid s, S \quad \Gamma \vDash e : \tau \mid t, T}{\Gamma \vDash P; v = e \mid s + t, S.T} \\
\text{[null]} & \quad \frac{}{\Gamma \vDash \emptyset \mid 0, 0} \\
\text{[var]} & \quad \frac{}{\Gamma \vDash v : \tau \mid 0, 0} \\
\text{[sleep]} & \quad \frac{}{\Gamma \vDash \text{sleep} t \mid t, 0} \\
\text{[act]} & \quad \frac{}{\Gamma \vDash A^i \mid 0, 0} \\
\text{[sync]} & \quad \frac{}{\Gamma \vDash \text{sync} n \mid 0, n!} \\
\text{[cue]} & \quad \frac{}{\Gamma \vDash \text{cue} n \mid 0, n?} \\
\text{[spawn]} & \quad \frac{}{\Gamma \vDash P \mid t, S} \\
\text{[loop]} & \quad \frac{}{\Gamma \vDash \text{loop} P \mid \infty, \mu p.(S.p)} \\
\text{iterate} & \quad \frac{}{\Gamma \vDash P \mid t, S} \\
\frac{}{n \cdot \text{times} P \mid nt, \mu p.(S.p)} \\
\text{[cond]} & \quad \frac{\Gamma \vDash e_1 : \tau \mid s, S \quad \Gamma e_2 : \tau \mid t, T}{}
\end{align*}
\]
Putting it together (2)

\[
\begin{align*}
\text{[abs]} & \quad \Gamma, x : \sigma \vdash e : \tau \mid n(x), S(x) \\
\Gamma & \vdash \lambda x.e : (x : \sigma) \quad \frac{n(x), S(x)}{\tau} \mid 0, 0
\end{align*}
\]

\[
\begin{align*}
\text{[app]} & \quad \Gamma \vdash e_1 : (x : \sigma) \quad \frac{n(x), S(x)}{\tau} \mid n_1, S_1 \\
\Gamma & \vdash e_2 : \sigma \mid n_2, S_2 \\
\Gamma & \vdash e_1 e_2 : \tau \mid n(e_2) + n_1 + n_2, S_1.S_2.S(e_2)
\end{align*}
\]
Time safety - extending

\[ [\_\_]_t : \text{actual time} \]

\[ [\emptyset]_t \approx 0 \]

\[ [P; \text{sleep } t]_t \approx ([P]_v + t) \max [P]_t \]

\[ [P; v = A^i]_t \approx [P]_t + [A^i]_t \]

Example 1.

It may wait for anywhere between the sleep statement are roughly the same, which are limited to a very small part of the model. Crucially, these operations changing virtual time: virtual time is specified for statements of Sonic Pi according to Definition 1, and expressions, since the evaluation and expressions, and specifications of virtual time and actual time of a single subprogram is not a straightforward recursive decomposition on programs, statements, and expressions for the core subset of the language described here. We'll prove this sound semantics with respect to the axiomatic model of this section, linking the two levels of model.

Consider subprograms: (a) by Definition 1, (b) by Definition 2, (c) by (b). Trivial since \( \emptyset \approx 0 \).

\[ [P; \text{sleep } t]_t \approx ([P]_v + t) \max [P]_t \]

\[ [P; v = A^i]_t \approx [P]_t + [A^i]_t \]

Note that this proof only makes use of basic properties on relations and subsequent derivations and the specifications of the virtual time and actual time of a single subprogram. Instead, the actual time of a kernelSleep comes before it (the tail of the "snoc"-list). Thus, the syntactic kernels, by monotonicity and (c) we have that:

\[ [P]_t \approx 0 \]

\[ [P; \text{sleep } t]_t \approx ([P]_v + t) \max [P]_t \]

\[ [P; v = A^i]_t \approx [P]_t + [A^i]_t \]
Time safety - extending

\[ [P \mid Q]_t \quad \text{where} \quad P = \text{cue} : A; \ P' \\
Q = \text{sync} : A; \ Q' \]

\[ [Q]_t \approx \text{if} \ [P']_t \leq [Q']_t \]

\[ \text{then} \ [P']_t \times \left( [Q']_t / [P']_t \right) \]

\[ \text{else} \ [P']_t \]

- Complicated multiple (non-leading) cues/syncs
- In practice, one sync/cue per looped thread is fine
  - Auto cue on `live_loop`
  - Optional sync at head
Challenges - “liveness”

- Responsive analysis
- Update AST with changed code, rather than complete re-analyse
- Online analysis? (during typing)
Conclusions

- Programming with music is really fun
- Great for education
- Interesting verification challenges for music
- Need fast analysis for live-programming
http://sonic-pi.net/
@fib_crisis