Holonomic Programs

Jacques Carette\textsuperscript{1}

\textsuperscript{1}McMaster University

Oct. 20, 2006
Overview

1. From (holonomic) recurrences to programs
2. Now: from programs to recurrences / closed-forms
3. Classical Semantics of Programming Languages
4. Symbolic Semantics
5. Approximation
6. Open Questions
Holonomy

Definition
A sequence $s_n$ is holonomic iff it satisfies a linear recurrence with polynomial coefficients.

Definition
A function $f(x)$ is holonomic iff it satisfies a linear differential equation with polynomial coefficients.

Theorem
Holonomic sequences are effectively closed under $+,$ $\times,$ shifting, etc. Holonomic functions are effectively closed under $+,$ $\times,$ differentiation, etc.

Proposition
$L$ and $M$ effectively translate between linear recurrences and linear differential equations.
\textit{gfun} is your friend!

Over to Maple...
Over to Maple...

All the code we get is

- Particularly simple,
- Even has a nice (triangular) structure
Over to Maple...

All the code we get is

- Particularly simple,
- Even has a nice (triangular) structure
- But that’s because we reduce to a first-order system...
From code to recurrences

The problem we’ll study is to go the other way around.

We will be happy first with recurrences, then with closed-forms.

Applications: program understanding, reverse engineering, debugging.

Remark: There is no natural notion of differentiation in programs – but there is of shifting: loops
From code to recurrences

Off to Maple one more time!

\begin{verbatim}
Fundamental observation:
while cond ( state )
do
state := F( state )
end do;
\end{verbatim}

at first \( t \) such that \( \text{cond}(state) = \text{false} \)

First-order, generally non-linear recurrence
Off to Maple one more time!

**Fundamental observation:**

\[
\text{while } \text{cond}(\text{state}) \text{ do } \\
\quad \text{state} := F(\text{state}) \\
\text{end do;}
\]

\[
s_{t+1} = [F](s_t)
\]

at first \(t\) such that \(\text{cond}(s_t) = \text{false}\)
From code to recurrences

Off to Maple one more time!

Fundamental observation:

```markdown
while cond(state) do
    state := F(state)
end do;
```

\[ s_{t+1} = [F](s_t) \]

at first \( t \) such that \( \text{cond}(s_t) = \text{false} \)

First-order, generally non-linear recurrence
Programming language semantics

- Rough view of classical PL semantics:
  - **Operational Semantics** tells you how to run a program
  - **Denotational Semantics** tells you what a program means (functionally)
  - **Axiomatic Semantics** tells you what a program means (logically)

- We want to introduce *symbolic* semantics

  | Representation | language contains usual mathematical operators (+, *, <, ⩽) but also ∑, ∏, special functions, recurrences, etc. |
Rough view of classical PL semantics:

- **Operational Semantics** tells you how to run a program
- **Denotational Semantics** tells you what a program means (functionally)
- **Axiomatic Semantics** tells you what a program means (logically)

We want to introduce *symbolic* semantics

For simple PL semantics, this formalizes as:

- A *State* is a value assignment for all identifiers
- Operational Semantics: $\text{Statement} \times \text{State} \rightarrow \text{State}$
Rough view of classical PL semantics:

**Operational Semantics** tells you how to run a program

**Denotational Semantics** tells you what a program means (functionally)

**Axiomatic Semantics** tells you what a program means (logically)

We want to introduce *symbolic* semantics

For simple PL semantics, this formalizes as:

A State is a value assignment for all identifiers

Operational Semantics: $\text{Statement} \times \text{State} \rightarrow \text{State}$

Denotational Semantics: $\text{Statement} \rightarrow (\text{State} \Rightarrow \text{State})$
Programming language semantics

- Rough view of classical PL semantics:
  - **Operational Semantics** tells you how to run a program
  - **Denotational Semantics** tells you what a program means (functionally)
  - **Axiomatic Semantics** tells you what a program means (logically)

- We want to introduce *symbolic* semantics

- For simple PL semantics, this formalizes as:
  
  A *State* is a value assignment for all identifiers

  Operational Semantics: \( \text{Statement} \times \text{State} \rightarrow \text{State} \)

  Denotational Semantics: \( \text{Statement} \rightarrow (\text{State} \Rightarrow \text{State}) \)

  Symbolic Semantics: \( \text{Statement} \rightarrow (\text{State} \rightarrow \text{State}) \)
Rough view of classical PL semantics:

**Operational Semantics** tells you how to run a program

**Denotational Semantics** tells you what a program means (functionally)

**Axiomatic Semantics** tells you what a program means (logically)

We want to introduce *symbolic* semantics

For simple PL semantics, this formalizes as:

A State is a value assignment for all identifiers

Operational Semantics: \( \text{Statement} \times \text{State} \rightarrow \text{State} \)

Denotational Semantics: \( \text{Statement} \rightarrow (\text{State} \Rightarrow \text{State}) \)

Symbolic Semantics: \( \text{Statement} \rightarrow (\text{State} \rightarrow \text{State}) \)

*Representation* language contains usual mathematical operators (+, *, <, ≤) but also ∑, ⋀, special functions, recurrences, etc.
More formally

\[
E ::= \text{Var} \\
| \text{i} \\
| E + E | E \times E | E - E | E/E | E^i
\]

\[
B ::= E = E | E < 0 \\
| B \text{ and } B | B \text{ or } B | \text{not } B
\]

\[
S ::= \text{Var} := E \\
| S ; S \\
| \text{while } B \text{ do } S \text{ end}
\]

\[
P ::= \text{proc}(\text{Var}^*) \{\text{localVar}^+\} S ; \text{return } E \text{ end procedure}
\]

real or integer variable
integer literal
arithmetic operations
bool from expressions
boolean operations
variable assignment
sequencing
while loop
procedure
Operational Semantics

\[ \sigma(E_1) \Rightarrow E'_1 \quad \sigma(E_2) \Rightarrow E'_2 \]
\[ \sigma(E'_1 + E'_2) \Rightarrow E'_1 + E'_2 \]
\[ \sigma(v_1 + v_2) \Rightarrow v_1 + v_2 \]

\[ \sigma(E) \Rightarrow \text{false} \]
\[ \Rightarrow \sigma(\text{while } E \text{ do } S \text{ end}) \Rightarrow \sigma \]

\[ \sigma(E) \Rightarrow \text{true} \quad \sigma(S) \Rightarrow \sigma_1 \]
\[ \Rightarrow \sigma(\text{while } E \text{ do } S \text{ end}) \Rightarrow \sigma_1(\text{while } E \text{ do } S \text{ end}) \]
### Denotational Semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Var}]\sigma)</td>
<td>(\sigma(\text{Var}))</td>
</tr>
<tr>
<td>([E_1 + E_2])</td>
<td>([E_1] + [E_2])</td>
</tr>
<tr>
<td>([E_1 - E_2])</td>
<td>([E_1] - [E_2])</td>
</tr>
<tr>
<td>([E_1\ ^\wedge\ i])</td>
<td>([E_1][i])</td>
</tr>
<tr>
<td>([E_1 \text{ or } E_2])</td>
<td>([E_1] \land [E_2])</td>
</tr>
<tr>
<td>([E_1 = E_2])</td>
<td>([E_1] = [E_2])</td>
</tr>
<tr>
<td>([\text{Var} := E]\sigma)</td>
<td>(\sigma \oplus {\text{Var} \leftarrow [E]})</td>
</tr>
<tr>
<td>([i])</td>
<td>(i)</td>
</tr>
<tr>
<td>([E_1 \ast E_2])</td>
<td>([E_1] \ast [E_2])</td>
</tr>
<tr>
<td>([E_1 / E_2])</td>
<td>([E_1] / [E_2])</td>
</tr>
<tr>
<td>([\text{not } E_1])</td>
<td>(\neg [E_1])</td>
</tr>
<tr>
<td>([E_1 &lt; 0])</td>
<td>([E_1] &lt; 0)</td>
</tr>
<tr>
<td>([S_1 ; S_2]\sigma)</td>
<td>(<a href="%5BS_1%5D%5Csigma">S_2</a>)</td>
</tr>
</tbody>
</table>

\[
[\text{while } B \text{ do } S \text{ end}]\sigma = \text{Fix } F \quad \text{where } F g = \begin{cases} 
    g \circ [S] & [B]\sigma = \text{true} \\
    \text{id} & [B]\sigma = \text{false} \\
    \bot & \text{otherwise}
\end{cases}
\]

\[
[\text{proc}(x_1, x_2, \ldots, x_n) \{\text{local } l_1, l_2, \ldots, l_m\} S; \text{ return } E \text{ end}] = \\
\lambda x_1, \ldots, x_n. [E]([S]\sigma_{x,l})
\]

where \(\sigma_{x,l}\) denotes the state where identifiers \(x_1, \ldots, x_n\) and \(l_1, \ldots, l_m\) are in the range
Symbolic Semantics

\[
\begin{align*}
[\text{Var}]\sigma &= \sigma(\text{Var}) & [i] &= i \\
[E_1 + E_2] &= [E_1] + [E_2] & [E_1 \ast E_2] &= [E_1] \ast [E_2] \\
[E_1 \downarrow i] &= [E_1] \downarrow [i] & \text{not } E_1 &= \text{not } [E_1] \\
[E_1 \text{ or } E_2] &= [E_1] \text{ or } [E_2] & [E_1 \text{ and } E_2] &= [E_1] \text{ and } [E_2] \\
[E_1 = E_2] &= [E_1] = [E_2] & [E_1 < 0] &= [E_1] < 0 \\
[\text{Var} := E]\sigma &= \delta(\sigma, \{\text{Var} \leftarrow [E]\sigma\}) & [S_1 ; S_2]\sigma &= \delta([S_1], [S_2])([S_1]\sigma)
\end{align*}
\]

\[
\sigma(\text{Var}) =
\begin{cases}
\nu & \text{Var} = \nu \in \sigma \text{ and } \sigma \text{ is a store} \\
\nu & \sigma = \delta(\sigma', \text{Var} \leftarrow \nu) \\
\sigma'(\text{Var}) & \sigma = \delta(\sigma', x \leftarrow \nu) \text{ and } x \neq \text{Var} \\
\text{Var} & \text{otherwise}
\end{cases}
\]
Symbolic Semantics for `while`

\[
\begin{align*}
\llbracket \text{Var} \rrbracket_w \sigma &= \sigma(\text{Var}(t)) \\
\llbracket \text{Var} := E \rrbracket_w \sigma &= \delta(\sigma, \{\text{Var}(t + 1) \leftarrow \llbracket E \rrbracket_\sigma\}) \\
\llbracket \text{while } B \text{ do } S \text{ end} \rrbracket_\sigma &= \text{Let } B'_t = \llbracket B \rrbracket_w \sigma \quad S'_t = \llbracket S \rrbracket_w \\
&= e = \min(\{t \geq 0 \mid B'_t = \text{true}\}) \\
&\quad \mu(S'_t, t = e, e(\vec{s}, \sigma)) \\
\llbracket \text{proc}(\vec{x}) \{\text{local } \vec{l}\} S; \text{ return } E \text{ end} \rrbracket &= \llbracket E \rrbracket(\llbracket S \rrbracket_\sigma'_{x, l})
\end{align*}
\]