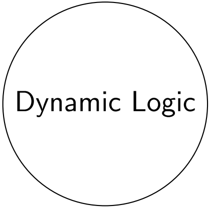
Dynamic Logic

How loops are actually recursion
Hype for Types

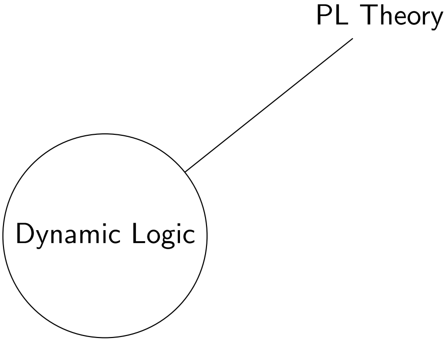
Jacob Neumann

08 October 2019

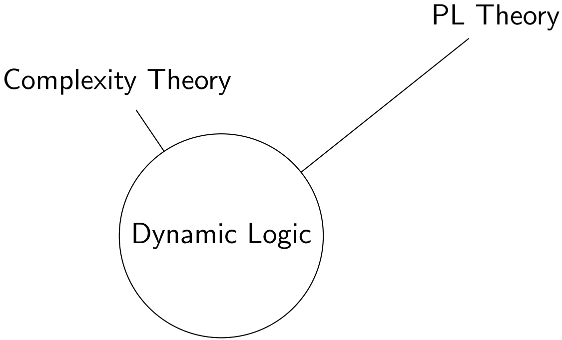
Dynamic Logic is everywhere



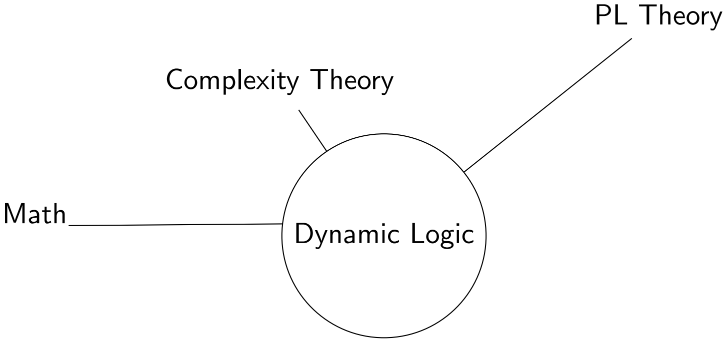
Dynamic Logic is everywhere



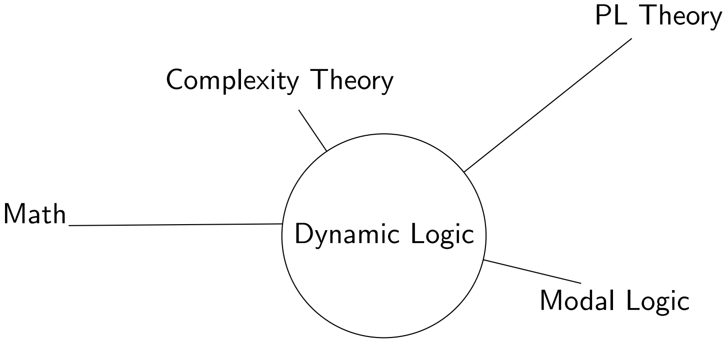
Dynamic Logic is everywhere



Dynamic Logic is everywhere



Dynamic Logic is everywhere



Dynamic Logic is everywhere

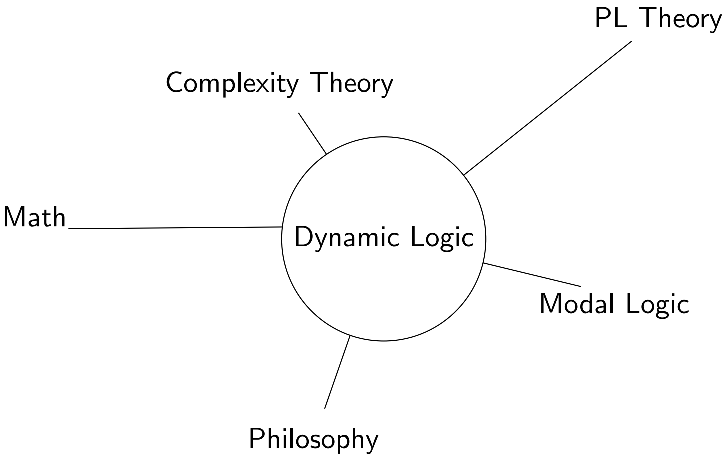


Table of Contents

1 Syntax and Semantics

2 Deterministic PDL

3 Proving Behavior in DPDL

4 Hoare Logic

5 Other Cool Stuff


WARNING

IMPERATIVE CODE AHEAD

IMPERATIVE CODE AHEAD
(also math)

Section 1

Syntax and Semantics

Some philosophy...

Question: What is programming?

Some philosophy...

Question: What is programming?

(One possible) answer:

- Programming is the art of communicating with computers

Some philosophy...

Question: What is programming?

(One possible) answer:

- Programming is the art of communicating with computers
- We communicate with computers using otherwise-meaningless strings of symbols

Some philosophy...

Some philosophy...

```
while true:  print("AHHHH")
```

Some philosophy...

```
fun fact 0 = 1
```

```
while true:  print("AHHHH")
```

Some philosophy...

```
fun fact 0 = 1
```

```
001001101010001
```

```
while true:  print("AHHHH")
```

Some philosophy...

```
(lambda (arg) (+ arg 1))
```

```
fun fact 0 = 1
```

```
001001101010001
```

```
while true:  print("AHHHH")
```

Some philosophy...

```
(lambda (arg) (+ arg 1))
```

```
fun fact 0 = 1
```

```
/([a-z0-9\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})/
```

```
001001101010001
```

```
while true:  print("AHHHH")
```

S stands for...

The symbols used in a language is called the **syntax**.

S stands for...

The symbols used in a language is called the **syntax**.

The study of how to assign computational meaning to these symbols is called **semantics**.

S stands for...

The symbols used in a language is called the **syntax**.

The study of how to assign computational meaning to these symbols is called **semantics**.

Formally specifying semantics for our programming languages allows us to *mathematically prove* properties about how our code works. This allows us to:

S stands for...

The symbols used in a language is called the **syntax**.

The study of how to assign computational meaning to these symbols is called **semantics**.

Formally specifying semantics for our programming languages allows us to *mathematically prove* properties about how our code works. This allows us to:

- Be sure our code will return the right result

S stands for...

The symbols used in a language is called the **syntax**.

The study of how to assign computational meaning to these symbols is called **semantics**.

Formally specifying semantics for our programming languages allows us to *mathematically prove* properties about how our code works. This allows us to:

- Be sure our code will return the right result
- Know how long our code will take to run

S stands for...

The symbols used in a language is called the **syntax**.

The study of how to assign computational meaning to these symbols is called **semantics**.

Formally specifying semantics for our programming languages allows us to *mathematically prove* properties about how our code works. This allows us to:

- Be sure our code will return the right result
- Know how long our code will take to run
- Be sure that we won't run into unforeseen bugs at runtime

Operational vs. Denotational

There are two main approaches to specifying programming language semantics: **operational semantics** and **denotational semantics**.

Operational vs. Denotational

There are two main approaches to specifying programming language semantics: **operational semantics** and **denotational semantics**.

- Operational semantics specifies the steps a program takes in executing code

$$\frac{s \mapsto s' \qquad s' \mapsto^* s''}{s \mapsto^* s''}$$

Operational vs. Denotational

There are two main approaches to specifying programming language semantics: **operational semantics** and **denotational semantics**.

- Operational semantics specifies the steps a program takes in executing code

$$\frac{s \mapsto s' \qquad s' \mapsto^* s''}{s \mapsto^* s''}$$

- Denotational semantics interprets the syntax of a programming language as a mathematical object

$$\|\pi\| : X \rightharpoonup X$$

Operational vs. Denotational

There are two main approaches to specifying programming language semantics: **operational semantics** and **denotational semantics**.

- Operational semantics specifies the steps a program takes in executing code

$$\frac{s \mapsto s' \qquad s' \mapsto^* s''}{s \mapsto^* s''}$$

- Denotational semantics interprets the syntax of a programming language as a mathematical object

$$\|\pi\| : X \rightharpoonup X$$

In this lecture, I'll be focusing on the *denotational* approach.

Section 2

Deterministic PDL

DPDL

Deterministic Propositional Dynamic Logic (DPDL) is a formal semantics for interpreting a basic programming language.

DPDL

Deterministic Propositional Dynamic Logic (DPDL) is a formal semantics for interpreting a basic programming language.It consists of:

- A state space

DPDL

Deterministic Propositional Dynamic Logic (DPDL) is a formal semantics for interpreting a basic programming language.It consists of:

- A state space
- Interpretations of all the programs as *partial functions* on the state space

DPDL

Deterministic Propositional Dynamic Logic (DPDL) is a formal semantics for interpreting a basic programming language.It consists of:

- A state space
- Interpretations of all the programs as *partial functions* on the state space
- A apparatus for formulating logical statements about the state space

The State Space

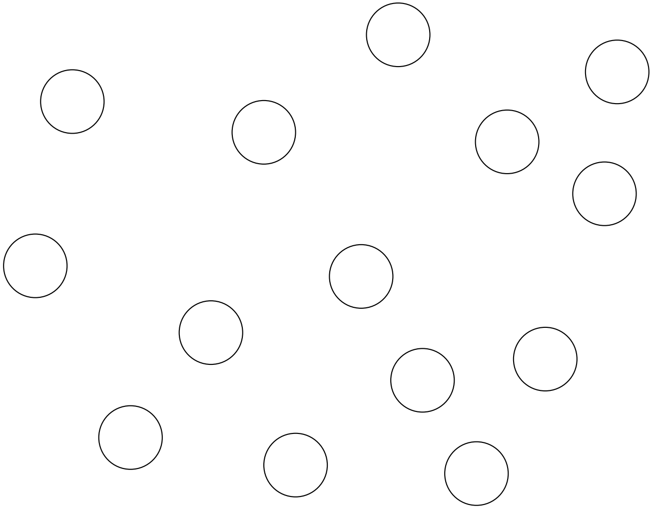
We mathematically model a computer as a *set X of internal states (or configurations)*. The behavior of our programs will depend on the state of the computer.

The State Space

We mathematically model a computer as a *set X of internal states (or configurations)*. The behavior of our programs will depend on the state of the computer.

X is often either finite or countably infinite, although in some applications we will want to have more states.

The State Space



The Programs

Let $\Pi = \{\pi_0, \pi_1, \ldots\}$ be a set of "program names".

The Programs

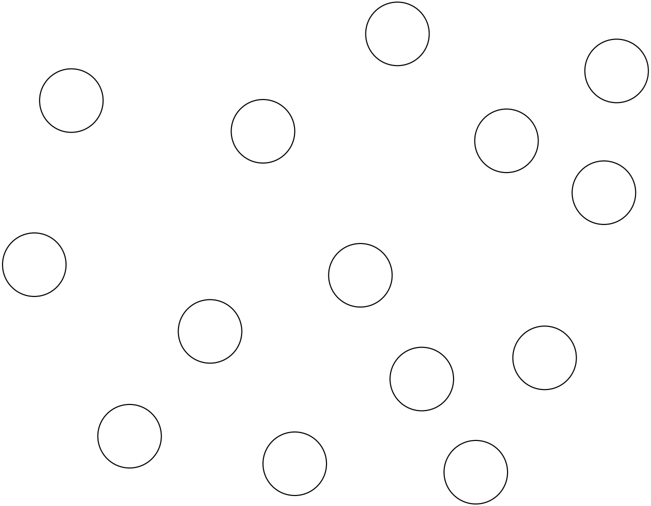
Let $\Pi = \{\pi_0, \pi_1, \ldots\}$ be a set of "program names".

Each program symbol $\pi \in \Pi$ *denotes* a partial function on our state space. We write this as:

$$\|\pi\| : X \rightharpoonup X$$

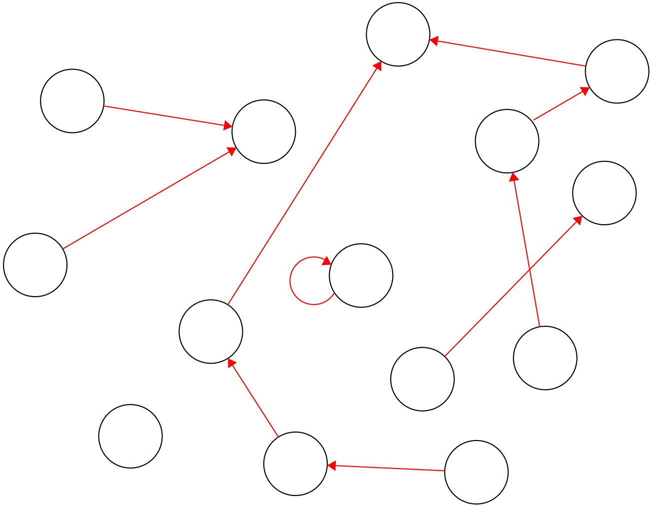
So, for each state $x \in X$, "executing π at x" will either succeed (terminate) and result in a new state $\|\pi\|(x)$, or it will crash (encoded by $\|\pi\|(x)$ being undefined).

The Programs



The Programs

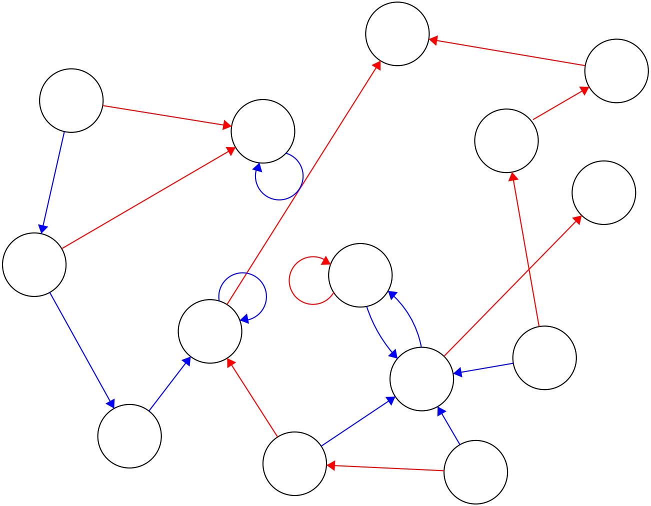
π_0



The Programs

π_0

π_1



The Propositions

Let $\Phi = \{p_0, p_1, \ldots\}$ be a countable set of "propositional variables". These propositional variables denote logical statements we might want to make about a state x.

The Propositions

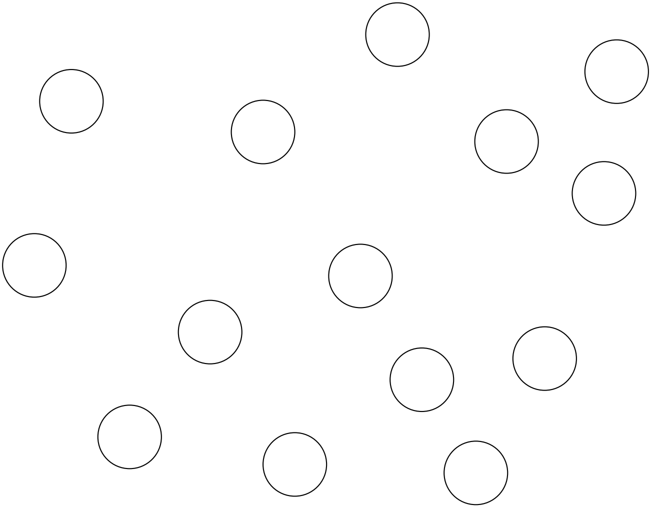
Let $\Phi = \{p_0, p_1, \ldots\}$ be a countable set of "propositional variables". These propositional variables denote logical statements we might want to make about a state x.

Each propositional variable $p \in \Phi$ *denotes* a subset of our state space. We write this as:

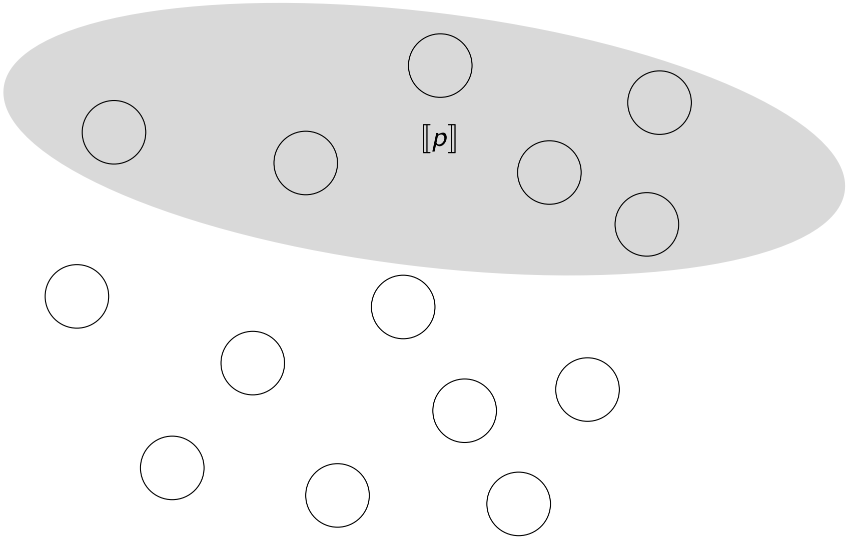
$$[\![p]\!] \subseteq X$$

Think of $[\![p]\!]$ as the set of states where p is true.

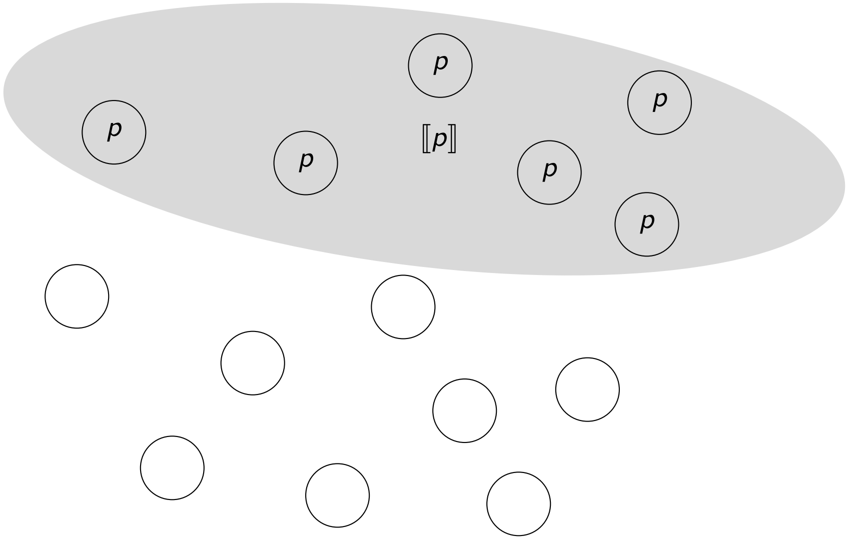
The Propositions



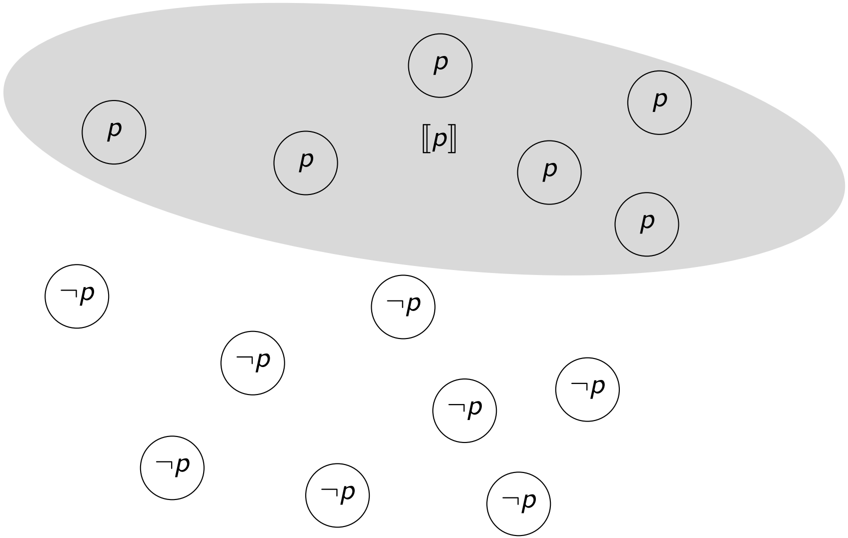
The Propositions



The Propositions



The Propositions



Expanding The Propositions

We can make the set of “statements” more interesting, using **recursive definitions**!

Expanding The Propositions

We can make the set of "statements" more interesting, using **recursive definitions**!

- If φ is some statement, then $\neg\varphi$ is its negation: the statement that φ is not true:

Expanding The Propositions

We can make the set of "statements" more interesting, using **recursive definitions**!

- If φ is some statement, then $\neg\varphi$ is its negation: the statement that φ is not true:

$$x \in [\![\neg\varphi]\!] \iff$$

Expanding The Propositions

We can make the set of "statements" more interesting, using **recursive definitions**!

- If φ is some statement, then $\neg\varphi$ is its negation: the statement that φ is not true:

$$x \in [\![\neg\varphi]\!] \iff x \notin [\![\varphi]\!]$$

Expanding The Propositions

We can make the set of "statements" more interesting, using **recursive definitions**!

- If φ is some statement, then $\neg\varphi$ is its negation: the statement that φ is not true:

$$x \in [\![\neg\varphi]\!] \iff x \notin [\![\varphi]\!]$$

- If φ and ψ are some statements, then $\varphi \wedge \psi$ is their *conjunction*: the statement that both φ and ψ are true:

Expanding The Propositions

We can make the set of "statements" more interesting, using **recursive definitions**!

- If φ is some statement, then $\neg\varphi$ is its negation: the statement that φ is not true:

$$x \in [\![\neg\varphi]\!] \iff x \notin [\![\varphi]\!]$$

- If φ and ψ are some statements, then $\varphi \wedge \psi$ is their *conjunction*: the statement that both φ and ψ are true:

$$x \in [\![\varphi \wedge \psi]\!] \iff$$

Expanding The Propositions

We can make the set of "statements" more interesting, using **recursive definitions**!

- If φ is some statement, then $\neg\varphi$ is its negation: the statement that φ is not true:

$$x \in [\![\neg\varphi]\!] \iff x \notin [\![\varphi]\!]$$

- If φ and ψ are some statements, then $\varphi \wedge \psi$ is their *conjunction*: the statement that both φ and ψ are true:

$$x \in [\![\varphi \wedge \psi]\!] \iff x \in [\![\varphi]\!] \text{ and } x \in [\![\psi]\!]$$

Expanding The Propositions

- If φ is a statement and $\pi \in \Pi$ is some program name, then $[\pi]\varphi$ is the statement "if π terminates, then φ will be true after π terminates":

$$x \in [\![\, [\pi]\varphi \,]\!] \iff$$

Expanding The Propositions

- If φ is a statement and $\pi \in \Pi$ is some program name, then $[\pi]\varphi$ is the statement "if π terminates, then φ will be true after π terminates":

$$x \in [\![\, [\pi]\varphi \,]\!] \iff \|\pi\|(x) \in [\![\varphi]\!] \text{ or } \|\pi\|(x) \text{ is undefined}$$

Expanding The Propositions

- If φ is a statement and $\pi \in \Pi$ is some program name, then $[\pi]\varphi$ is the statement "if π terminates, then φ will be true after π terminates":

$$x \in [\![\, [\pi]\varphi \,]\!] \iff \|\pi\|(x) \in [\![\varphi]\!] \text{ or } \|\pi\|(x) \text{ is undefined}$$

Expanding The Propositions

- If φ is a statement and $\pi \in \Pi$ is some program name, then $[\pi]\varphi$ is the statement "if π terminates, then φ will be true after π terminates":

$$x \in [\![\, [\pi]\varphi \,]\!] \iff \|\pi\|(x) \in [\![\varphi]\!] \text{ or } \|\pi\|(x) \text{ is undefined}$$

The formula $\langle\pi\rangle\varphi$, which is defined to be $\neg[\pi]\neg\varphi$, expresses the statement "π terminates, and results in a φ state".

Sanity Check: What does this formula say?

$$\varphi \rightarrow \langle \pi \rangle \psi$$

(here, $p \rightarrow q$ is used as an abbreviation for $\neg p \vee q$).

Sanity Check: What does this formula say?

$$\varphi \to \langle \pi \rangle \psi$$

(here, $p \to q$ is used as an abbreviation for $\neg p \vee q$).

REQUIRES: φ
ENSURES: ψ

Expanding The Program Suite

To encode (and study) more interesting behavior, we can recursively define new programs out of old ones:

Expanding The Program Suite

To encode (and study) more interesting behavior, we can recursively define new programs out of old ones:

- Given programs π_1 and π_2, we can make the program $\pi_1; \pi_2$, and give it the following semantics:

$$\|\pi_1; \pi_2\|(x) = \|\pi_2\|\big(\|\pi_1\|(x)\big)$$

Expanding The Program Suite

To encode (and study) more interesting behavior, we can recursively define new programs out of old ones:

- Given programs π_1 and π_2, we can make the program $\pi_1; \pi_2$, and give it the following semantics:

$$\|\pi_1; \pi_2\|(x) = \|\pi_2\|\left(\|\pi_1\|(x)\right)$$

- Given programs π_1 and π_2, and a formula φ, we can make the program **if** φ **then** π_1 **else** π_2, with the following semantics:

Expanding The Program Suite

To encode (and study) more interesting behavior, we can recursively define new programs out of old ones:

- Given programs π_1 and π_2, we can make the program $\pi_1;\pi_2$, and give it the following semantics:

$$\|\pi_1;\pi_2\|(x) = \|\pi_2\|\left(\|\pi_1\|(x)\right)$$

- Given programs π_1 and π_2, and a formula φ, we can make the program **if** φ **then** π_1 **else** π_2, with the following semantics:

$$\|\textbf{if } \varphi \textbf{ then } \pi_1 \textbf{ else } \pi_2\|(x) = \begin{cases} \|\pi_1\|(x) & \text{if } x \in [\![\varphi]\!] \\ \|\pi_2\|(x) & \text{if } x \notin [\![\varphi]\!] \end{cases}$$

Expanding The Program Suite

- Given a program π and a formula φ, we can make the program **while** φ **do** π, with the following semantics:

Expanding The Program Suite

- Given a program π and a formula φ, we can make the program **while** φ **do** π, with the following semantics:

$$\|\textbf{while } \varphi \textbf{ do } \pi\|(x) = \begin{cases} x & \text{if } x \notin [\![\varphi]\!] \\ \|\textbf{while } \varphi \textbf{ do } \pi\|\big(\|\pi\|(x)\big) & \text{if } x \in [\![\varphi]\!] \end{cases}$$

What we now have

So we have given semantics for a simple programming language, with:

What we now have

So we have given semantics for a simple programming language, with:

- A (possibly large) set of program states

What we now have

So we have given semantics for a simple programming language, with:

- A (possibly large) set of program states
- Whatever basic programs we might want

What we now have

So we have given semantics for a simple programming language, with:

- A (possibly large) set of program states
- Whatever basic programs we might want
- Sequencing, conditionals, and loops

What we now have

So we have given semantics for a simple programming language, with:

- A (possibly large) set of program states
- Whatever basic programs we might want
- Sequencing, conditionals, and loops
- A logical syntax to talk about state properties before and after executing a function

Section 3

Proving Behavior in DPDL

Based on how we set up the logic, the following rules are true **at every state of every model**, for any programs π, π_1, π_2 and any formulas φ, ψ, θ:

Based on how we set up the logic, the following rules are true **at every state of every model**, for any programs π, π_1, π_2 and any formulas φ, ψ, θ:

- $$\frac{(\varphi \rightarrow [\pi_1]\psi) \qquad (\psi \rightarrow [\pi_2]\theta)}{\varphi \rightarrow [\pi_1; \pi_2]\theta}$$

Based on how we set up the logic, the following rules are true **at every state of every model**, for any programs π, π_1, π_2 and any formulas φ, ψ, θ:

- $$\frac{(\varphi \to [\pi_1]\psi) \qquad (\psi \to [\pi_2]\theta)}{\varphi \to [\pi_1; \pi_2]\theta}$$

- $$\frac{(\varphi \to [\pi_1]\psi) \qquad (\neg\varphi \to [\pi_2]\psi)}{[\textbf{if } \varphi \textbf{ then } \pi_1 \textbf{ else } \pi_2]\psi}$$

Based on how we set up the logic, the following rules are true **at every state of every model**, for any programs π, π_1, π_2 and any formulas φ, ψ, θ:

- $$\frac{(\varphi \to [\pi_1]\psi) \qquad (\psi \to [\pi_2]\theta)}{\varphi \to [\pi_1;\pi_2]\theta}$$

- $$\frac{(\varphi \to [\pi_1]\psi) \qquad (\neg\varphi \to [\pi_2]\psi)}{[\textbf{if } \varphi \textbf{ then } \pi_1 \textbf{ else } \pi_2]\psi}$$

- $$\frac{(\varphi \wedge \psi) \to [\pi]\psi}{\psi \to [\textbf{while } \varphi \textbf{ do } \pi](\neg\varphi \wedge \psi)}$$

Section 4

Hoare Logic

From PDL to Hoare Logic

It's kinda tedious to write $\varphi \to [\pi]\psi$ over and over, and so we can adopt the precondition-postcondition notation established by Tony Hoare: $\{\varphi\}\,\pi\,\{\psi\}$.

From PDL to Hoare Logic

It's kinda tedious to write $\varphi \rightarrow [\pi]\psi$ over and over, and so we can adopt the precondition-postcondition notation established by Tony Hoare: $\{\varphi\}\,\pi\,\{\psi\}$.

Hoare Logic is more powerful than PDL because it allows for *variable binding* and *integer arithmetic*. For example, we can say stuff like:

- $\{\mathtt{n} \geq 0\}\, i := n\, \{\mathtt{i} \geq 0\}$

From PDL to Hoare Logic

It's kinda tedious to write $\varphi \to [\pi]\psi$ over and over, and so we can adopt the precondition-postcondition notation established by Tony Hoare: $\{\varphi\}\,\pi\,\{\psi\}$.

Hoare Logic is more powerful than PDL because it allows for *variable binding* and *integer arithmetic*. For example, we can say stuff like:

- $\{\mathtt{n} \geq 0\}\, i := n \,\{\mathtt{i} \geq 0\}$
- $\left\{\mathtt{a} = \mathtt{b}^{\mathtt{i}}\right\} a := a * b \left\{\mathtt{a} = \mathtt{b}^{\mathtt{i+1}}\right\}$

Here are our rules from earlier, in the Hoare notation:

- $$\frac{\{\varphi\}\,\pi_1\,\{\psi\} \qquad \{\psi\}\,\pi_2\,\{\theta\}}{\{\varphi\}\,\pi_1;\pi_2\,\{\theta\}}$$

- $$\frac{\{\varphi\}\,\pi_1\,\{\psi\} \qquad \{\neg\varphi\}\,\pi_2\,\{\psi\}}{\{\}\,\textbf{if }\varphi\textbf{ then }\pi_1\textbf{ else }\pi_2\,\{\psi\}}$$

- $$\frac{\{\varphi \wedge \psi\}\,\pi\,\{\psi\}}{\{\psi\}\,\textbf{while }\varphi\textbf{ do }\pi\,\{\neg\varphi \wedge \psi\}}$$

A 122-style example

```
i:=n;
res:=1;
(while (i>0)
do

   res := res * b;
   i := i-1

);
```

A 122-style example

```
i:=n;
res:=1;
(while (i>0)
do
   {i > 0  ∧ i ≥ 0  ∧  res * b^i = b^n}
   res := res * b;
   i := i-1
   {i ≥ 0  ∧ res * b^i = b^n}
);
```

A 122-style example

```
i:=n;
res:=1;
```
$\{\mathtt{i} \geq 0 \ \wedge \ \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$
```
(while (i>0)
do
```
$\{\mathtt{i} > 0 \ \wedge \mathtt{i} \geq 0 \ \wedge \ \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$
```
  res := res * b;
  i := i-1
```
$\{\mathtt{i} \geq 0 \ \wedge \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$
```
);
```
$\{\neg(\mathtt{i} > 0) \ \wedge \ \mathtt{i} \geq 0 \ \wedge \ \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$

A 122-style example

$\{\mathtt{n} \geq 0\}$

i:=n;
res:=1;
$\{\mathtt{i} \geq 0 \ \wedge \ \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$

(while (i>0)
do
  $\{\mathtt{i} > 0 \ \wedge \ \mathtt{i} \geq 0 \ \wedge \ \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$
  res := res * b;
  i := i-1
  $\{\mathtt{i} \geq 0 \ \wedge \ \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$
);
$\{\neg(\mathtt{i} > 0) \ \wedge \ \mathtt{i} \geq 0 \ \wedge \ \mathtt{res} * \mathtt{b}^{\mathtt{i}} = \mathtt{b}^{\mathtt{n}}\}$

Other Cool Stuff

Hoare Logic and DPDL are really only just the beginning of things you can do with denotational and axiomatic semantics!

Other Cool Stuff

Hoare Logic and DPDL are really only just the beginning of things you can do with denotational and axiomatic semantics!

- Nondeterministic program semantics (replace the partial functions with binary relations)

Other Cool Stuff

Hoare Logic and DPDL are really only just the beginning of things you can do with denotational and axiomatic semantics!

- Nondeterministic program semantics (replace the partial functions with binary relations)
- Heap Allocation

Other Cool Stuff

Hoare Logic and DPDL are really only just the beginning of things you can do with denotational and axiomatic semantics!

- Nondeterministic program semantics (replace the partial functions with binary relations)
- Heap Allocation
- Concurrency

Other Cool Stuff

Hoare Logic and DPDL are really only just the beginning of things you can do with denotational and axiomatic semantics!

- Nondeterministic program semantics (replace the partial functions with binary relations)
- Heap Allocation
- Concurrency
- Cost Semantics

Other Cool Stuff

Hoare Logic and DPDL are really only just the beginning of things you can do with denotational and axiomatic semantics!

- Nondeterministic program semantics (replace the partial functions with binary relations)
- Heap Allocation
- Concurrency
- Cost Semantics
- More complex mathematics to make the modal logic more powerful (topological structure, structure-preserving maps and category theory, etc.)

Why is this in Hype for Types?

Why is this in Hype for Types?

- Remember the central idea of denotational semantics: interpret computer programs as *mathematical objects*

Why is this in Hype for Types?

- Remember the central idea of denotational semantics: interpret computer programs as *mathematical objects*
- We can do this for functional languages too!

Why is this in Hype for Types?

- Remember the central idea of denotational semantics: interpret computer programs as *mathematical objects*
- We can do this for functional languages too!
- In a few weeks, we'll

Why is this in Hype for Types?

- Remember the central idea of denotational semantics: interpret computer programs as *mathematical objects*
- We can do this for functional languages too!
- In a few weeks, we'll
  - Introduce the mathematical language of *category theory*

Why is this in Hype for Types?

- Remember the central idea of denotational semantics: interpret computer programs as *mathematical objects*
- We can do this for functional languages too!
- In a few weeks, we'll
  - Introduce the mathematical language of *category theory*
  - Develop denotational semantics for the simply-typed lambda calculus *inside certain nice categories*

Why is this in Hype for Types?

- Remember the central idea of denotational semantics: interpret computer programs as *mathematical objects*
- We can do this for functional languages too!
- In a few weeks, we'll
  - Introduce the mathematical language of *category theory*
  - Develop denotational semantics for the simply-typed lambda calculus *inside certain nice categories*
  - This approach can be extended to type theories which are *much* fancier than simple lambda calculus (e.g. dependent type theory, higher-inductive dependent type theory (homotopy type theory)). This is an area of very active research.

Why is this in Hype for Types?

- Remember the central idea of denotational semantics: interpret computer programs as *mathematical objects*
- We can do this for functional languages too!
- In a few weeks, we'll
  - Introduce the mathematical language of *category theory*
  - Develop denotational semantics for the simply-typed lambda calculus *inside certain nice categories*
  - This approach can be extended to type theories which are *much* fancier than simple lambda calculus (e.g. dependent type theory, higher-inductive dependent type theory (homotopy type theory)). This is an area of very active research.
- So stay tuned...

Thank you!