NTNU – Trondheim
Norwegian University of
Science and Technology

# Denotational semantics

# What we're doing today

- We're looking at how to reason about the effect of a program by mapping it into mathematical objects
  - Specifically, answering the question “which function does this program compute?”
- We'll run into some issues when we get to programs that potentially never stop with a result
  - We're going for functions between environment states, they can only be *partial* functions when there are states that produce no end state

# What is a program, anyway?

- As far as the machine is concerned: instructions, data, memory, yadda yadda...
- Those are all configurations of tiny switches, oblivious to the computation they represent in the same way that a traffic light doesn't know what its states and transitions tell people
- Independent of the machine, a program is also a description of a method to compute a result
  - To programmers, at least

# What can we compute?

- A *primitive recursive function* is defined in terms of
  - The *constant function* 0 (which takes no arguments, and outputs 0)
  - The *successor function* S(k) = k+1 (which adds 1 to a number)
  - The *projection function* $P_i^n$ (x1, …, xi, …, xn ) = xi (which selects value number *i* out of a bunch of values
- These are enough to define a bit of arithmetic:
  - The most tedious addition method in the world...

    add ( 0, x ) = x ← base: x+0 = x

    add ( S(n), x ) = S ( $P_1^3$ ( add(n,x), n, x ) ) ← step: x+(n+1)=(x+n)+1
  - The most tedious subtraction method follows, from sub. by differences
  - Multiply and divide can be built from add & sub, and so on and so forth...
  - It all boils down to simple schemes of counting one step at a time

NTNU – Trondheim
Norwegian University of Science and Technology

# The primitive side of it

- Primitive recursive functions can compute anything which maps uniquely onto all the natural numbers, under some kind of encoding/interpretation
- That is, they're *total*, meaning “uniquely defined for all admissible sets of inputs”
- Everything which maps to natural numbers is quite a bunch of stuff, but it's restricted to programs that terminate with a defined result
  - Hence, no branching and nothing fancy, please
  - That's kind of primitive

# *Partial* recursive functions

- If we add the power of saying something like

  (∃y) R(y,x)

  to mean

  “The smallest x such that R(y,x) is true”, or

  “0” if no such y exists

  we get a conditional, of sorts.
- We also have equivalence with Turing machines: conditionals + jumps can be written as conditionals + recursion
  - Writing out anything nontrivial in this notation is also the equivalent amount of fun as writing them out in terms of Turing machines
  - Let's not go there, the point is that they're equivalent

NTNU – Trondheim
Norwegian University of
Science and Technology

# That's the edge of the world

(computationally speaking)

- With enough spare time on your hands, it can be proven that the partial recursive functions are also exactly what can be computed by
  - Lambda calculus
  - Register machines
  - A few more exotic models of computation
- At a point where he must have been tired of proving things, Alonzo Church ($\lambda$-calculus Guy) made his mind up that these are the functions we can get from any computational model, and left it at that. We'll take his word for it.
- As we know, loops can be infinite, so these functions don't have values for *all* inputs any more

# What a program is

- Hence, one way of looking at “a program” is that it's an evaluation of a partial recursive function.
- Neither programmer nor program may care, it just means that you can always write it out that way
  - Programs which stop have their function's value for the given input
  - Programs which don't stop don't have any kind of value, because they never produce one
- Infinite loops can be very annoying
  - At least when you wanted to calculate a result
- Infinite loops can be very useful
  - I will be upset if my laptop halts to conclude that the value of the operating system is 42

# Which programs stop?

- We can not compute the answer to that
  - Suppose that we could, and had a function

```
halts ( p(x) ) =
    if magical_analysis(p(x)) then yes
    else no
```

  - Never mind how it works, just suppose that it can take any function p with any input x, and answer whether or not it returns
  - This lets us write a function that answers only about programs which have themselves as input:

```
halts_on_self ( p ) =
    if ( halts (p(p)) ) then yes
    else no
```

NTNU – Trondheim
Norwegian University of
Science and Technology

# I have a cunning plan...

- We can easily make a function run forever on purpose, so write one which does that when a function-checking function halts on itself:

```
trouble ( p ) =
    if ( halts_on_self(p) ) then loop_forever
    else yes
```

- Since 'trouble' is a function-checking function, we can see what it would make of itself:

```
trouble ( trouble ) =
    if ( halts_on_self(trouble) ) then loop_forever
    else yes
```

  which is equivalent to

```
trouble ( trouble ) =
    if ( halts(trouble(trouble)) ) then loop_forever
    else yes
```

- If it halts, it should loop forever ; if it loops forever, it should halt.
- This program can not exist, so the halting function can not.

NTNU – Trondheim
Norwegian University of Science and Technology

# That's why this gets messy

- We just looked at a pseudocode-y variant of Turing's proof that the halting problem is not computable
- It can also be written out in terms of a counting scheme and partial recursive functions, but this way may be a bit more intuitive
- Bottom line: we can't expect to find well behaved functions for every arbitrary program
- Without that, we have to take extra care of how to define a program in terms of its function

# Revisiting the operational approach

- Focus was on *how a program is executed*
- Each syntactic construct is interpreted in terms of the steps taken to modify the state it runs in
- The *semantic function* is described by a recipe for how to compute its value (the final state), when it has one

# “Denote” (verb):

- To serve as an indication of
- To serve as an arbitrary mark for
- To stand for

# *Denotational* semantics

- The program is a way to symbolize a semantic function
- Its characters are arbitrary, as long as we can systematically map them onto the mathematical objects they represent
  - The string “10” can mean natural number 10 (decimal), 2 (binary), 16 (hexadecimal)...
  - ...in Roman numerals, 10 is “X”...
  - The symbol is one thing, what it denotes is another

# Basic parts

- The hallmarks of denotational semantics are
  - There is a semantic clause for all basis elements in a category of things to symbolize
  - For each method of combining them, there is a semantic clause which specifies how to combine the semantic functions of the constituents

# The simplest illustration

- Take this grammar for arbitrary binary strings:

```
b → 0
b → 1
b → b 0
b → b 1
```

- ...and let `b,0,1` stand for the symbols in our grammar, while {0,1,2,...} are the natural numbers...

# A semantic function

- We can write a function N to attach the natural numbers to valid statements in the grammar:

  N ( `0` ) = 0

  N ( `1` ) = 1

  N ( `b 0` ) = 2 * N ( `b` )

  N ( `b 1` ) = 2 * N ( `b` ) + 1
- This is just the ordinary interpretation of binary strings as unsigned integers, written out all formal-like
- Each notation is related to the mathematical object it denotes (here, it's a natural number)

# Finding a value

- Using this formalism, we can write out what the value of "`1001`" is:

N ( `1001` )
= 2 * N ( `100` ) + 1
= 2 * ( 2 * N ( `10` ) ) + 1
= 2 * ( 2 * ( 2 * N ( `1` ) ) ) + 1
= 2 * ( 2 * ( 2 * 1 ) ) + 1
= 2 * ( 4 ) + 1
= **9**

N ( `0` ) = 0
N ( `1` ) = 1
N ( `b 0` ) = 2 * N ( `b` )
N ( `b 1` ) = 2 * N ( `b` ) + 1

# Finding a value

Symbols from grammar are systematically replaced with their semantic interpretations

<u>N ( 1001 )</u>
= 2 * <u>N ( 100 )</u> + 1
= 2 * ( 2 * <u>N ( 10 )</u> ) + 1
= 2 * ( 2 * ( 2 * <u>N ( 1 )</u> ) ) + 1
= 2 * ( 2 * ( 2 * 1 ) ) + 1
= 2 * ( 4 ) + 1
= **<u>9</u>**

Result is a thing the input can't contain, and the compiler can't understand

# Is this a valuable thing?

- Well... the example is so small that it's almost pointless
- *In principle,* however:
  - Assume an implementation which sets lowest order bit according to last symbol in string, and shifts left to multiply by 2
  - In a signed byte-wide register w. 2's complement, this would make the value of `11111111` = -1, whereas N(`11111111`) = 255
  - With semantics defined by the implementation, whatever comes out is the standard of what's correct
  - Semantic specification in hand, we can say that such an implementation doesn't do what it's supposed to

# Remember the *While* language:

- Syntax:

  a → n | x | a1 + a2 | a1 * a2 | a1 – a2

  b → true | false | a1 = a2 | a1 ≤ a2 | ¬b | b1 & b2

  S → x := a | skip | S1 ; S2

  S → if b then S1 else S2 | while b do S
- Syntactic categories:
  - n is a numeral
  - x is a variable
  - a is an arithmetic expression, valued A[a]
  - b is a boolean expression, valued B[b]
  - S is a statement

# Denotational semantics for *While*

- What we attach to the statements should be *a function which describes the effect of a statement*
  - The steps taken to create that effect is presently not our concern
- Skip and assignment are still easy:

  $S_{ds}$ [ x:=a ] s = s [ x → A[a]s ]    (as before)

  $S_{ds}$ [ skip ] = id    (identity function)
- Composition of statements corresponds to composition of functions:

  $S_{ds}$ [ S1; S2 ] = $S_{ds}$ [ S2 ] ○ $S_{ds}$ [ S1 ]

  "S2-function applied to the result of S1-function", *cf.* how f ○ g (x) ↔ f ( g ( x ) )

# Conditions need a notation

- Specifically, a function which goes from one boolean and two other functions, and results in one of the two functions
- Let's call it *cond*, and write

  $S_{ds}$ [ if b then S1 else S2 ] = cond ( B[b], $S_{ds}$ [S1], $S_{ds}$ [S2] )

  with the understanding that, for example,

  cond ( B[true], $S_{ds}$ [x:=2], $S_{ds}$ [skip] ) s = s [ x $\rightarrow$ A[2]s ]

  and

  cond ( B[false], $S_{ds}$ [x:=2], $S_{ds}$ [skip] ) s = id s

# 'while b do S' gets a little tricky

- What we need is a function applied to a function applied to a function... as many times as the condition is true
- Problems:
  - The program text does not always determine how many times the condition will be true
  - It is not guaranteed that it ever will be false
- The function we are looking for is specific to each program
  - We have a notation to denote “the outcome of the loop body”: $S_{ds}[S]$
  - We need one to denote “the outcome of repeating the loop body an unknown number of times”

# Calculating with *functionals*

- In the manner that a variable is a named placeholder for a range of values...
- ...and a function is a named placeholder for a way to combine variables...
- ...so a *functional* F is a generalized range of functions, which can stand for any of them

# Functions as unknowns

- This lets us treat a functional F as “the function which fits our constraints”
  - in the same way we can write x for “the value which fits the constraint x*2+12 = 42”, and treat x as the solution to that
- Looking at how to read 'while b do S', we can write out its halting condition in terms of *cond* (from before), and an unknown function g:

  $F\ g = \text{cond}\ (\ B[b],\ g \circ S_{ds}[S],\ \text{id}\ )$
- That is: given any function *g* (as “input”), the functional F represents either the effect of applying *g* to the outcome of the loop body, or the identity function, depending on B[b].
- The resulting function can be applied to states where B[b] has a value

# Definition of a "fixed point"

- This is mercifully simple
- A *fixed point* is where taking an argument and doing some stuff to it results in the argument itself
- *i.e.* **when f(x) = x, then x is a fixed point of f**
- 2 is a fixed point of $f(x) = (x^2 / 2x) + 1$
- It's "fixed" since it doesn't change no matter how many times you apply the function:

  $x = f(x) = f(f(x)) = f(f(f(x))) =$ *…and so on*

NTNU – Trondheim
Norwegian University of Science and Technology

# Thus, we can (partly) describe the effect of 'while b do S'

- $S_{ds}$ [while b do S] = FIX F

  where F g = cond ( B[b], $g \circ S_{ds}$[S], id )

- That is, it's a function where it may be the case that

  cond(B[b],$S_{ds}$[S], id ) s = s'

  cond(B[b],$S_{ds}$[S], id ) s' = s''

  ...

  cond(B[b],$S_{ds}$[S], id ) $s^{(n-1)} = s^{(n)}$

  but eventually,

  cond(B[b],$S_{ds}$[S], id ) $s^{(n)} = s^{(n)}$

  and the loop doesn't alter anything any more.

  – That will be the case when it has ended
  – When it doesn't end, we can't describe the effect, and no solution should be defined

# So, what's the outcome of a loop?
## (Without running it?)

- Take the factorial program we looked at for the operational case:

  while ¬(x=1) do ( y:=y*x; x:=x-1 )

- We're interested in functions g that satisfy

  cond ( B[b], $g \circ S_{ds}$[S], id ) s = s

  that is,

  cond ( B[b], g ○ [x → A[x:=x-1]] ○ [y → A[y*x]], id ) s = s

- Generally, these have the form of the functional

| | | |
|---|---|---|
| (F g) s = g s | if x is different from 1 | (do something to the state) |
| (F g) s = s | if x = 1 | (that's the loop halting condition) |

NTNU – Trondheim
Norwegian University of Science and Technology

# What kind of g fits FIX (F g)?

- Here's one:

  g1 = g1 s    if x>1
  g1 = s    if x=1
  g1 = undef    if x<1

Intuitive from program, Loop eternally into neg. x if it starts out too small

- Here's another:

  g2 = g2 s    if x>1
  g2 = s    if x=1
  g2 = s    if x<1

Also a function which gives s back when x=1

- These are both fixed points of the functional (F g)
  - Substitute g1 and g2 into it, you get that

    (F g1) s = g1 s
    and
    (F g2) s = g2 s

# An additional constraint

- We can create any number of g-s like this, we want to narrow them down into one which reflects what the program means
- Since we've abstracted away the implementation, we need to say something about which fixed points are admissible

# When things loop forever

- If the execution of (while b do S) in state s never halts, there is an infinite number of states $s_1$, $s_2$, … such that
  - $B[b]\ s_i = tt$ (*i.e.* the condition is true)
  - $S_{ds}[S]\ s_i = s_{i+1}$ (*i.e.* the loop continues to churn through states)
- An immediate example is

  while ¬(x=0) do skip

  and its matching functional

  (F g) s = g s    if x is different from 0 in s

  (F g) s = s    if x = 0 in s

NTNU – Trondheim
Norwegian University of Science and Technology

# Which fixed point are we after?

- The reason we have an infinity to choose from:
  - Any g where g s = s if x=0 in s is a fixed point
- The intuition we aim to capture is that

  g s = undef          if x is different from 0

  g s = s     if x=0 in s
- Every other g will have to say something about s in at least some cases when x isn't 0:

  g' s = undef          if x > 0

  g' s = s          if x = 0

  g' s = s[y → A[y+1]s]          if x < 0
  - This also captures the effect of the program when it is defined, but adds a bunch of unrelated nonsense about y when it is not defined
  - Still a function that captures the effect of the program as much as the other one

# Between the lines

- There is an *ordering* of all possible choices of g, comparing them by how much they specify
- The relationship that

  g0 s = s' implies g s = s'  (but not the other way around)

  indicates that all the effects of g0 are also in g
- Writing this as g0 $\preccurlyeq$ g,

  (with a slightly bent 'smaller-or-equal' character, to signify that this is a different type of comparison than that between numbers)

  we get a notion that there is a 'minimal' g

# Making a unique choice

- Add the understanding that 'undef' implies anything and everything
  - Like 'false' does for the implication in boolean logic
- The least fixed point in this sense is the most concise description of a loop's effect
  - We'll take that one as the semantic function, then

# Sum total

- Denotational semantics for *While*:

  $S_{ds}$ [ x:=a ] s = s [ x $\rightarrow$ A[a]s ]

  $S_{ds}$ [ skip ] = id

  $S_{ds}$ [ S1; S2 ] = $S_{ds}$ [ S2 ] $\circ$ $S_{ds}$ [ S1 ]

  $S_{ds}$ [ if b then S1 else S2 ] = cond ( B[b], $S_{ds}$ [S1], $S_{ds}$ [S2] )

  $S_{ds}$ [while b do S] = FIX F

  where F g = cond ( B[b], g$\circ S_{ds}$[S], id )

  and FIX F is the least fixed point

# "Precision of an analysis"

- I alluded at one point that there is a notion of more and less *precise* semantic analyses
  - and mentioned that it carries a particular meaning of "precise"
- The part about finding the desired fixed point is it.
  - "Most precise" is not the fixed point with the most information in
  - It is the one which most accurately represents what we know about the program

# But seriously, *why the...?*

- Once again, we have taken an idea that plays a part in the curriculum and stretched it, to see how it works out when applied to a whole (but small) language
- The result is an algebra of semantic functions
  - and a notion that our handle on halting is a fixed point of a semantic function
  - and an idea that such a function may have multiple fixed points
  - and that these relate to each other in an order determined by how much information they specify
  - ...which I will say just a tiny bit more about next time

# No <u>seriously</u>, *why the...?*

- Ok. The next (and last) part of theory is a framework for deciding on how control flow affects what we can say about the state of a program.
- Its function maps statements to sets of variables, values, *etc.* to reason about the program environment
- It halts on a fixed point of the function which produces those sets of things
- It relates that fixed point to other fixed points in a ranking of how precise their information is, using an unorthodox choice of operators
- It's pretty much a variant of what we just looked at, except it is restricted to capturing state information which enables optimizations

# So, that's what comes next?

- Yes.
- It'll be a little easier to anchor the state information in aspects of the source code, but we'll still deal with some properties that aren't embodied in the compiler program
- Hopefully, this overview may contribute a way to look at dataflow analysis which makes it easier to see a system among its details
- If it doesn't, you can figure things out anyway
  - Don't lose any sleep over denotational semantics if you can follow DF analysis without seeing the correspondence, it's meant as an alternate perspective