Software Engineering Tools 01
Syntax, Semantics, Types

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

16. April 2014

What do programs mean?

Let's run the following program in some language:

```
print(32767 + 1);
```

Which of the following outputs is correct?

- `32768`
- `32767 + 1`
- `-32768`
- banana
- *no* visible output

Must know the program's *meaning*

Semantics

Semantics: The study of *meaning* (logic, linguistics)

- "meaning should follow structure"
  - This is a *hypothesis* in linguistics (seems to hold)
  - And a *proposal* in logic (turns out to work reasonably well)

Example:

- If expression '`X`' has meaning 'v'
- And expression '`Y`' has meaning 'w'
- Then expression '`(X) / (Y)`' has meaning 'whatever number you get when you compute $\frac{v}{w}$'

What if 'v' is not a number, or 'w' is zero?

Overview

Today we will look at:

- *syntax*: Describe structure of programs
- *semantics*: Derive meaning from syntax
  - *static semantics*: Meaning that is assigned before the program runs
    (mostly types, errors)
  - *dynamic semantics*: Run-time meaning
  - We won't explore this separation in detail today
- *types*: Describe 'semantic structure' of programs

Backus-Naur Form: Specifying Syntax

Assume *nat* is a natural number:
Formalise the rules with *Backus-Naur-Form* (BNF):

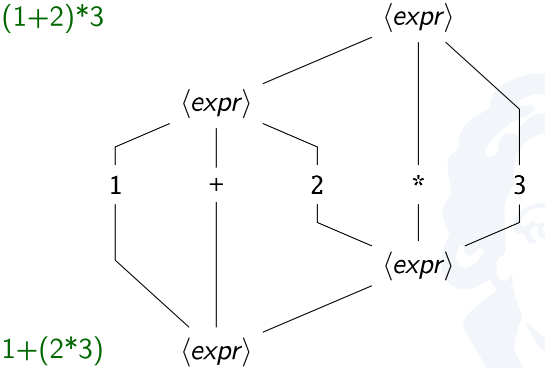
- 'Any number is an expression.'
  - $expr ::= nat$
- 'Any two expressions with a + in between is also an expression.'
  - $expr ::= \langle expr \rangle \text{'+'} \langle expr \rangle$
- 'Any two expressions with a * in between is also an expression.'
  - $expr ::= \langle expr \rangle \text{'*'} \langle expr \rangle$

Or in short:

$$expr ::= nat \mid \langle expr \rangle \text{'+'} \langle expr \rangle \mid \langle expr \rangle \text{'*'} \langle expr \rangle$$

Backus-Naur Form: Example

$$expr ::= nat \mid \langle expr\rangle\text{'+'}\langle expr\rangle \mid \langle expr\rangle\text{'*'}\langle expr\rangle$$



Ambiguity! Parsers must know which parse we mean!

Syntax of a simple toy language

Syntax of language STOL:

$$\begin{array}{rcl} \mathit{expr} & ::= & \mathit{nat} \\ & \mid & \langle \mathit{expr} \rangle \text{`}+\text{'} \langle \mathit{expr} \rangle \\ & \mid & \text{`}\texttt{ifnz}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{then}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{else}\text{'} \langle \mathit{expr} \rangle \end{array}$$

Examples:

- `5`
- `5 + 27`
- `ifnz 5 + 2 then 0 else 1`

Meaning of our toy language: examples

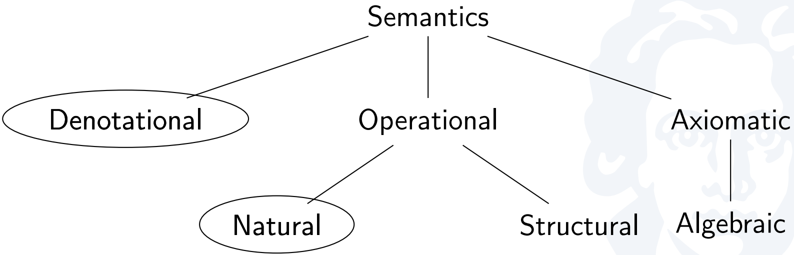
What we want the meaning to be:

`5`	5
`5 + 27`	32
`ifnz 5 + 2 then 1 else 0`	1

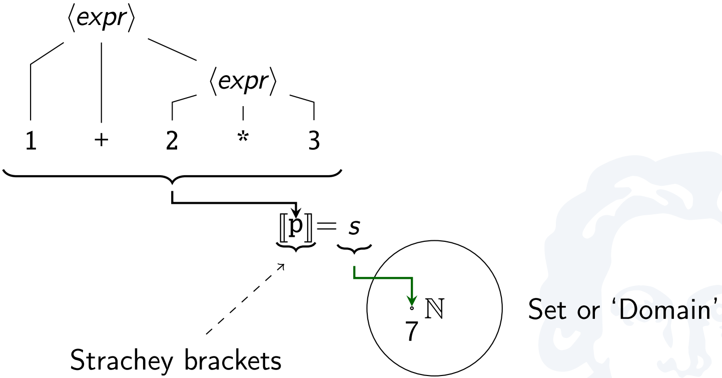
Can we describe this formally?

Defining Meaning

The principal schools of semantics:



Denotational Semantics



- Maps program to mathematical object
- Equational theory to reason about programs

Directly maps program to its mathematical 'meaning'

Denotational semantics of STOL

Distinguish:

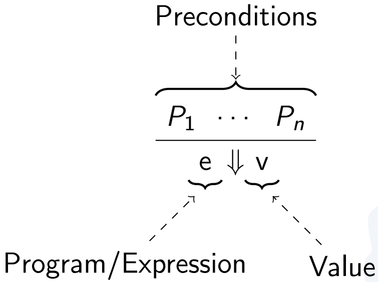
- `nat` is set of program numbers (`0`, `1`, `2`, ...)
  (In compilers: *character strings*)
- $\mathbb{N}$ is set of natural numbers (0, 1, 2, ...)
  (In compilers: *unsigned int* or *BigInt* types)

$$
\begin{array}{rcl}
n & \in & \mathtt{nat} \\
e, e_1, e_2, e_3 & \in & \mathtt{expr} \\
[\![n]\!] & = & \begin{cases} 0 & \Longleftrightarrow \quad n = \mathtt{0} \\ 1 & \Longleftrightarrow \quad n = \mathtt{1} \\ 2 & \Longleftrightarrow \quad n = \mathtt{2} \\ & \quad\quad\quad \ldots \end{cases} \\
[\![e_1 \mathtt{+} e_2]\!] & = & [\![e_1]\!] + [\![e_2]\!] \\
[\![\mathtt{\textbf{ifnz}}\ e_1\ \mathtt{\textbf{then}}\ e_2\ \mathtt{\textbf{else}}\ e_3]\!] & = & \begin{cases} [\![e_2]\!] & \Longleftrightarrow \quad [\![e_1]\!] \neq 0 \\ [\![e_3]\!] & \Longleftrightarrow \quad [\![e_1]\!] = 0 \end{cases}
\end{array}
$$

Operational Semantics: The two branches

- Natural Semantics (Big-Step Semantics)
  - $p \Downarrow v$: p evaluates to v
  - Describes *complete* evaluation
  - Compact, useful to describe interpreters
- Structural Operational Semantics (Small-Step Semantics)
  - $p_1 \rightarrow p_2$: p_1 evaluates one step to p_2
  - Captures individual *evaluation steps*
  - Verbose/detailed, useful for formal proofs

Natural (Operational) Semantics



If $P_1, \ldots, P_n$ all hold, then e evaluates to v.

- e: Arbitrary program (expression, in our example)
- v: Value that can't be evaluated any further (natural number, in our example)

Natural Semantics of our simple toy language

$$\begin{array}{rcl} n, n_1, n_2, n_3 & \in & \texttt{nat} \\ e, e_1, e_2, e_3 & \in & \texttt{expr} \end{array}$$

$$\frac{}{n \Downarrow n}\ (val) \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{e_1 + e_2 \Downarrow n}\ (add)$$

$$\frac{e_1 \Downarrow n \quad n \neq 0 \quad e_2 \Downarrow n_2}{\textbf{ifnz}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow n_2}\ (ifnz)$$

$$\frac{e_1 \Downarrow 0 \quad e_3 \Downarrow n_3}{\textbf{ifnz}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow n_3}\ (ifz)$$

Note:

- $(+)$ is arithmetic addition
- $+$ is a symbol in our language
- For simplicity, we set $\texttt{nat} = \mathbb{N}$

Natural Semantics: Example

$$
\dfrac{
  \dfrac{
    \dfrac{}{3 \Downarrow 3}\,(\mathit{val}) \quad
    \dfrac{}{2 \Downarrow 2}\,(\mathit{val}) \quad
    5 = 3 + 2
  }{3 + 2 \Downarrow 5}\,(\mathit{add}) \quad
  \dfrac{}{0 \Downarrow 0}\,\mathit{val}
}{\mathbf{ifnz}\ 3 + 2\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \Downarrow 0}\,(\mathit{ifnz})
$$

What's the point?

- Denotational and natural semantics *look* very similar
- Structural differences:
  - Denotational semantics describe a *function* $[\![-]\!]$
  - Natural semantics define a *relation* ($\Downarrow$)
  - Denotational semantics relies on mathematical *domain* with underlying equational theory
- Practical differences:
  - Natural Semantics requires less formal apparatus to describe (no domains)
  - Natural Semantics can't describe partial progress in non-terminating programs

Extending our language with 'let'

Name bindings $x \in name$:

$$\begin{array}{rcl} expr & ::= & nat \\ & \mid & \langle expr\rangle \text{`+'} \langle expr\rangle \\ & \mid & \text{`}\texttt{ifnz}\text{'}\langle expr\rangle\text{`}\texttt{then}\text{'}\langle expr\rangle\text{`}\texttt{else}\text{'}\langle expr\rangle \\ & \mid & name \\ & \mid & \text{`}\texttt{let}\text{'}name\text{`='}\langle expr\rangle\text{`}\texttt{in}\text{'}\langle expr\rangle \end{array}$$

Example:

$$[\![\texttt{let x = 2 + 3 in x + x}]\!] = 10$$

But what is $[\![x]\!]$ by itself?

Environments

The meaning of a variable depends on what value we bind it to.

Environment: $E : \mathbf{name} \rightarrow \mathbf{value}$

- Environments are *partial functions* from names to 'values'
- In our running example, $\mathtt{value} = \mathtt{nat}$

Notation:

let $E' = E + x \mapsto v$
then:

$$E'(y) = \begin{cases} v & \iff y = x \\ E(y) & \textit{otherwise} \end{cases}$$

Environments in Denotational Semantics

Introduce E as index to semantic function:

$$[\![-]\!]_E = \ldots$$

$$
\begin{array}{rcl}
n & \in & \texttt{nat} \\
e, e_1, e_2, e_3 & \in & \texttt{expr} \\
x & \in & \texttt{name} \\
[\![n]\!]_E & = & n \text{ interpreted in } \mathbb{N} \\
[\![e_1 + e_2]\!]_E & = & [\![e_1]\!]_E + [\![e_2]\!]_E \\
[\![\texttt{ifnz}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]_E & = & \begin{cases} [\![e_2]\!]_E & \Longleftrightarrow \quad [\![e_1]\!]_E \neq 0 \\ [\![e_3]\!]_E & \Longleftrightarrow \quad [\![e_1]\!]_E = 0 \end{cases} \\
[\![x]\!]_E & = & E(x) \\
[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]_E & = & [\![e_2]\!]_{E + x \mapsto [\![e_1]\!]_E}
\end{array}
$$

Environments in Natural Semantics

We borrow the turnstile ($\vdash$) from formal logic:

$$\frac{}{E \vdash n \Downarrow n}\ (val) \qquad \frac{E \vdash e_1 \Downarrow n_1 \quad E \vdash e_2 \Downarrow n_2 \quad n = n_1 + n_2}{E \vdash e_1 + e_2 \Downarrow n}\ (add)$$

$$\frac{E \vdash e_1 \Downarrow n \quad n \neq 0 \quad E \vdash e_2 \Downarrow n_2}{E \vdash \mathbf{ifnz}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow n_2}\ (ifnz)$$

$$\frac{E \vdash e_1 \Downarrow 0 \quad E \vdash e_3 \Downarrow n_3}{E \vdash \mathbf{ifnz}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow n_3}\ (ifz)$$

$$\frac{E(x) = v}{E \vdash x \Downarrow v}\ (var)$$

$$\frac{E \vdash e_1 \Downarrow v \quad (E + x \mapsto v) \vdash e_2 \Downarrow v'}{E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow v'}\ (let)$$

STOL-S: Extending our language with assignments

Side effects play an important role in realistic programs

- Must be modelled, for realism
- Tricky to model ⇒ purely functional languages have simpler semantic models

We extend STOL to STOL-S:

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{nat} \\
& | & \langle \mathit{expr} \rangle \text{`+'} \langle \mathit{expr} \rangle \\
& | & \text{`}\texttt{ifnz}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{then}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{else}\text{'} \langle \mathit{expr} \rangle \\
& | & \mathit{name} \\
& | & \text{`}\texttt{let}\text{'} \mathit{name} \text{`='} \langle \mathit{expr} \rangle \text{`}\texttt{in}\text{'} \langle \mathit{expr} \rangle \\
& | & \text{`}\texttt{ref}\text{'} \langle \mathit{expr} \rangle \\
& | & \text{`}\texttt{!}\text{'} \langle \mathit{expr} \rangle \\
& | & \langle \mathit{expr} \rangle \text{`}\texttt{:=}\text{'} \langle \mathit{expr} \rangle \\
& | & \text{`}\texttt{(}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{;}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{)}\text{'}
\end{array}
$$

STOL-S: State updates

- `ref 42` allocates memory cell, stores 42 (cf. `malloc()` or `new`).
- `! p` Reads memory from memory cell in variable `p` (cf. `*p` for pointers `p` in C).
- `p := 23` Updates memory cell in `p` with 23 (cf. `*p = 23` in C).
- `(p := 23; !p)` Sequence: assigns, then reads&returns (Sequencing operation, cf. `{ *p = 23; return *p; }`)

Example:

```
let r = ref 7
in  (
  r := !r + !r;
  !r + 1)
```

Stores

Store: $S : \mathtt{ref} \to \mathtt{value}$

- Analogous to environments
- Store maps memory references (‘`ref`’) to ‘values’
- Again, `value` = `nat` (for now)

Stores in Natural Semantics (1)

- Recursive evaluation may update the store...
- ...which the caller must be able to see.
- We adjust $\Downarrow$ to evalute tuples $\langle e, S \rangle$:
  $E \vdash \langle e, S \rangle \Downarrow \langle v, S' \rangle$
  means:
  - Given an environment E and a store S:
  - e evaluates to v, and
  - S is updated to S' in the process

Example:

$$\frac{E \vdash \langle e_1, S \rangle \Downarrow \langle n_1, S' \rangle \quad E \vdash \langle e_2, S' \rangle \Downarrow \langle n_2, S'' \rangle \quad n = n_1 + n_2}{E \vdash \langle e_1 + e_2, S \rangle \Downarrow \langle n, S'' \rangle} \ (add)$$

State is *threaded through* the rule: *evaluation order*

Stores in Natural Semantics (2)

$$\frac{E \vdash \langle e, S\rangle \Downarrow \langle v, S'\rangle \quad \rho \textit{ fresh in } S'}{E \vdash \langle \texttt{ref } e, S\rangle \Downarrow \langle \rho, S' + \rho \mapsto v\rangle}\ (\textit{ref})$$

$$\frac{E \vdash \langle e, S\rangle \Downarrow \langle \rho, S'\rangle \quad v = S'(\rho)}{E \vdash \langle !e, S\rangle \Downarrow \langle v, S'\rangle}\ (\textit{read})$$

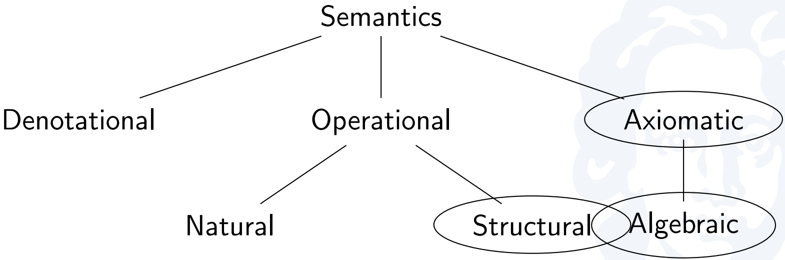
$$\frac{E \vdash \langle e_1, S\rangle \Downarrow \langle \rho, S'\rangle \quad E \vdash \langle e_2, S'\rangle \Downarrow \langle v, S''\rangle \quad \rho \in \textit{dom}\,(S'')}{E \vdash \langle e_1 := e_2, S\rangle \Downarrow \langle 0, S'' + \rho \mapsto v\rangle}\ (\textit{update})$$

$$\frac{E \vdash \langle e_1, S\rangle \Downarrow \langle v, S'\rangle \quad E \vdash \langle e_2, S'\rangle \Downarrow \langle v', S''\rangle}{E \vdash \langle (e_1\,;e_2), S\rangle \Downarrow \langle v', S''\rangle}\ (\textit{seq})$$

Analogously for the other rules.

Defining Meaning

Let's consider the other schools of semantics now:



Structural Operational Semantics (SOS)

(Definition on STOL)

$$\frac{e_1 \rightarrow^\star 0}{\textbf{ifnz}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightarrow e_3}\ (\mathit{ifz})$$

$$\frac{e_1 \rightarrow^\star n \quad \exists n'.n \rightarrow n' \quad n \neq 0}{\textbf{ifnz}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightarrow e_2}\ (\mathit{ifnz})$$

Comparison to Natural Semantics:

$\Downarrow \subseteq \texttt{expr} \times \texttt{nat}$	$\rightarrow \subseteq \texttt{expr} \times \texttt{expr}$
rhs is alwyas *fully* evaluated	rhs can be intermediate result

SOS can capture intermediate computational results

Axiomatic Semantics

Describe *statements*– not good fit for our current langauge

$$\{P\}statement\{Q\}$$

- P: Precondition
- Q: Postcondition
- if P holds, then *statement* ensures that Q holds

Example:

$$\{x \geq 0\}\texttt{x := x + 1;}\{x > 0\}$$

Frequently used for "design-by-contract" software development

Algebraic Semantics

Specification using techniques of *abstract algebra*, e.g.:

Sorts $\mathsf{list}, \mathtt{int}, \mathsf{string}$

Operations

$$
\begin{aligned}
\mathsf{empty} &: \mathsf{list} \\
\mathsf{add} &: \mathsf{list} \times \mathsf{string} \to \mathsf{list} \\
\mathsf{get} &: \mathsf{list} \times \mathtt{int} \to \mathsf{string} \\
\mathsf{size} &: \mathsf{list} \to \mathtt{int} \\
\mathsf{concat} &: \mathsf{list} \times \mathsf{list} \to \mathsf{list}
\end{aligned}
$$

Axioms

$$
\begin{aligned}
&\mathsf{size}(\mathsf{empty}) = 0 \\
&\forall \ell : \mathsf{list}, s : \mathsf{string}.\mathsf{get}(\mathsf{add}(\ell, s), \mathsf{size}(\ell)) = s \\
&\forall \ell : \mathsf{list}, s : \mathsf{string}.\mathsf{size}(\mathsf{add}(\ell, s)) = \mathsf{size}(\ell) + 1 \\
&\forall \ell_1, \ell_2 : \mathsf{list}.\mathsf{size}(\mathsf{concat}(\ell_1, \ell_2)) = \mathsf{size}(\ell_1) + \mathsf{size}(\ell_2)
\end{aligned}
$$

Comparison

- *Denotational Semantics*
  Equational theory, also describes nontermination
- *Natural Semantics*
  Compact, describes interpreter, doesn't give semantics to nonterminating programs
- *Structural Operational Semantics*
  Describes evaluation strategy, approximates semantics for nontermination
- *Axiomatic Semantics*
  Describes effect of *statements* (before/after), no nontermination
- *Algebraic Semantics*
  Describes effect of *operations* on opaque data structures, no nontermination

Types

$$e : \tau$$

Types are *contracts*: e must keep any promise made by τ

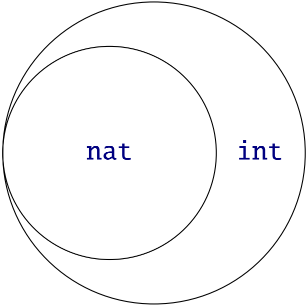
Typical promises:

- 'Any $e : \tau$ is a number between 0 and 42'
- 'Any $e : \tau$ is a record with a field `x`'
- 'Any $e : \tau$ has a method `m()`'

Types as Sets

Example:

- int: The type of integers, $\mathbb{Z}$
- nat: The type of natural numbers, $\mathbb{N}$



STOL with Subtraction

Let's introduce subtraction to STOL:

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{nat} \\
& | & \langle \mathit{expr} \rangle \text{`+'} \langle \mathit{expr} \rangle \\
& | & \text{`}\mathtt{ifnz}\text{'} \langle \mathit{expr} \rangle \text{`}\mathtt{then}\text{'} \langle \mathit{expr} \rangle \text{`}\mathtt{else}\text{'} \langle \mathit{expr} \rangle \\
& | & \mathit{name} \\
& | & \text{`}\mathtt{let}\text{'} \mathit{name} \text{`='} \langle \mathit{expr} \rangle \text{`}\mathtt{in}\text{'} \langle \mathit{expr} \rangle \\
& | & \langle \mathit{expr} \rangle \text{`-'} \langle \mathit{expr} \rangle
\end{array}
$$

A type system for STOL

- **Goal**: Detect which variables may be negative.
- **Approach**: Type analysis

Type environment: Γ : name $\rightarrow$ *type*

where *type* is the set of all types.

$$\frac{}{\Gamma \vdash n : \mathtt{nat}} \; (nat)$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \; (var)$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad (\Gamma + x \mapsto \sigma) \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let} \;\; x = e_1 \;\; \mathbf{in} \;\; e_2 : \tau} \; (let)$$

Addition and Subtraction

$$\frac{}{\Gamma \vdash n : \texttt{nat}}\ (nat)$$

$$\vdots$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 - e_2 : \texttt{int}}\ (sub)$$

How can we pass **nat** values to subtraction, though?

Subtypes and Implicit Conversion

One option:

- Introduce *subtyping*
- $\tau <: \sigma$ iff τ is subtype of σ.
- Meaning: if $e : \tau$, then we can use e anywhere we need a σ.

Formalised in the *Subsumption rule:*

$$\frac{\Gamma \vdash n : \tau \quad \tau <: \sigma}{\Gamma \vdash n : \sigma}\ (\textit{subsumption})$$

We set: nat $<:$ int

Example

$$
\dfrac{\Gamma \vdash 1 : \mathtt{nat} \qquad \dfrac{\dfrac{\dfrac{\Gamma'(z) = \mathtt{nat}}{\Gamma' \vdash z : \mathtt{nat}} \quad \mathtt{nat} <: \mathtt{int}}{\Gamma + z \mapsto \mathtt{nat} \vdash z : \mathtt{int}} \qquad \dfrac{\dfrac{}{\Gamma' \vdash 2 : \mathtt{nat}} \quad \mathtt{nat} <: \mathtt{int}}{\Gamma \vdash 2 : \mathtt{int}}}{\Gamma' \vdash z\text{-}2 : \mathtt{int}}}{\Gamma \vdash \mathbf{let}\ z = 1\ \mathbf{in}\ z\text{-}2 : \mathtt{int}}
$$

where $\Gamma' = \Gamma + z \mapsto \mathtt{nat}$

STOL-S and assignments

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{nat} \\
 & & \ldots \\
 & | & \text{`}\texttt{ref}\text{'}\langle \mathit{expr} \rangle \\
 & | & \text{`}\texttt{!}\text{'}\langle \mathit{expr} \rangle \\
 & | & \langle \mathit{expr} \rangle \text{`}\texttt{:=}\text{'}\langle \mathit{expr} \rangle \\
 & | & \text{`}\texttt{(}\text{'}\langle \mathit{expr} \rangle \text{`}\texttt{;}\text{'}\langle \mathit{expr} \rangle \text{`}\texttt{)}\text{'}
\end{array}
$$

How do we type references?

STOL-S and assignments: parametric ‘ref’

- Refences need their own type
- But: must distinguish references to `nat` vs. references to `int`

  ...and reference-to-reference-to-`nat` etc.
- Solution: parametric type $\texttt{ref}\langle\alpha\rangle$
  - $\texttt{ref}\langle\texttt{nat}\rangle$: reference to natural numbers
  - $\texttt{ref}\langle\texttt{int}\rangle$: reference to integers
  - $\texttt{ref}\langle\texttt{ref}\langle\texttt{int}\rangle\rangle$: reference to references to integers

STOL-S: typing rules

$$\frac{E \vdash e : \alpha}{E \vdash \textbf{ref}\ e : \texttt{ref}\langle\alpha\rangle}\ (\mathit{ref})$$

$$\frac{E \vdash e : \texttt{ref}\langle\alpha\rangle}{E \vdash !e : \alpha}\ (\mathit{read})$$

$$\frac{E \vdash e_1 : \texttt{ref}\langle\alpha\rangle \quad E \vdash e_2 : \alpha}{E \vdash e_1 := e_2 : \texttt{nat}}\ (\mathit{update})$$

Subtyping ref and its parameters

Assume:

$$\begin{array}{lcl} i & : & \texttt{int} \\ r_i & : & \texttt{ref}\langle\texttt{int}\rangle \\ r_n & : & \texttt{ref}\langle\texttt{nat}\rangle \end{array}$$

Should $\texttt{ref}\langle\texttt{int}\rangle$ and $\texttt{ref}\langle\texttt{nat}\rangle$ be subtypes?

- $\texttt{ref}\langle\texttt{int}\rangle :> \texttt{ref}\langle\texttt{nat}\rangle$?
  - If so: $r_n := i$ typechecks
  - Can assign -1 to non-negative memory!
- $\texttt{ref}\langle\texttt{int}\rangle <: \texttt{ref}\langle\texttt{nat}\rangle$?
  - If so: $!(\textbf{ifnz}\ 1\ \textbf{then}\ r_i\ \textbf{else}\ r_n) : \texttt{nat}$
  - Type checker believes that read from (possibly negative) memory is nonnegative!

Covariance and Contravariance

$$\tau\langle\alpha\rangle <: \tau\langle\beta\rangle$$

...is allowed if τ's type parameter is...

	(no constraint)	$\alpha <: \beta$
(no constraint)	**bivariant**	**covariant**
$\alpha :> \beta$	**contravariant**	**invariant** ($\alpha = \beta$)

Rules of thumb:

- Type parameter occurs read-only: covariant
- Type parameter occurs write-only: contravariant

Using Type systems

Type systems should have the following properties:

- *preservation*: a well-typed program does not change its type during execution
- *progress*: a well-typed program does not 'get stuck' during execution

If these are guaranteed, we can:

- Use type systems to check for errors
- Use type systems to *analyse properties* (e.g., 'could this number ever be negative?')

Literature (1)

- Natural Semantics:
  - Gilles Kahn, "Natural Semantics"
- Structural Operational Semantics:
  - Gordon Plotkin, "Natural Semantics"
- Axiomatic Semantics:
  - David Gries, "The Science of Programming"
  - C.A.R. Hoare, "An axiomatic basis for computer programming". Communications of the ACM 12
- Algebraic Semantics:
  - S Antoy. "Systematic design of algebraic specifications". In Proceedings of the Fifth International Workshop on Software Specification and Design

Literature (2)

- Types:
    - Kim Bruce, “Foundations of Object-Oriented Programming”
    - Benjamin Pierce, “Types and Programming Languages”
- Proofs:
    - Jean-Yves Girard, Taylor, Lafont: “Proofs and Types”

Next week:

Static Program Analysis