Formal Semantics: Transition Systems

Specifying Programs in CIC

- Recursion must be structural

```plaintext
# n ≥ 0
fixpoint fact n :=
  if n == 0 then 1
  else n * fact (n-1)
```

Not structurally decreasing 😞

```plaintext
# n ≥ 0
fixpoint fact n :=
  match n with
  0  => 1
  | k => n * fact k
```

Structurally decreasing 😊

Specifying Programs in CIC

- Sometimes there's no obvious way

```plaintext
# a,b > 0
while a != b:
  if a < b:
    b = b - a
  else:
    a = a - b
```

Iterative

```plaintext
# a,b > 0
fixpoint f a b :=
  if a == b then a
  else if a < b
    then f a (b - a)
    else f (a - b) b
```

Recursive

The Need for Termination

- We could prove termination of f...
  - ...but we need to define it first!
  - It can get very annoying to prove termination of every single program we write.

- Also, what do you know,
  - Some programs don't terminate.
Define Programs via Relations

- A program relates inputs to outputs.

\[
\text{Inductive euclid : } \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{P} : = \\
\text{base : } \forall a, \text{euclid } a \ a \ a \\
\text{step}_a : \forall a b z, a < b \rightarrow \text{euclid } a (b - a) z \rightarrow \text{euclid } a b z \\
\text{step}_b : \forall a b z, a > b \rightarrow \text{euclid } (a - b) b z \rightarrow \text{euclid } a b z.
\]

Proving termination reduces to:

Lemma euclid_terminates :
\[
\forall a b : \mathbb{N}, a > 0 \rightarrow b > 0 \rightarrow \exists z : \mathbb{N}, \text{euclid } a b z.
\]

Transition Systems

- A transition relation describes how the program state changes as statements are executed.

Programming Language Semantics

- WHILE language
  
  Syntax:

  \[
  S \rightarrow x := E | S | S | \text{skip} | \text{if } E \text{ then } S \text{ else } S | \text{while } E \text{ do } S \\
  E \rightarrow x | \# | E \bowtie E
  \]

Semantics: Expressions

- \( \sigma : \text{Var} \rightarrow \mathbb{Z} \)
  
  A store — assigns one value per variable

- \( \Sigma \) — the set of all such stores

- \([e] : \Sigma \rightarrow \mathbb{Z} \)
  
  \([e]_\sigma\) is the value of \( e \) when interpreted in state \( \sigma \)

\[
[a - b]([18, 16]) = 3
\]
Semantics: Expressions

Base cases

\[ [x] \sigma = \sigma x \quad (x \in \text{Var}) \]
\[ [n] \sigma = n \quad (n \in \mathbb{Z}) \]

Recursive case

\[ [e_1 \circ e_2] \sigma = [e_1] \sigma \circ [e_2] \sigma \]

Structural Operational Semantics

Small-step Semantics

- Define a new relation:
  \[ \sigma, c \rightarrow o', c' \]
  - running statement \( c \) in state \( \sigma \) results in a new state \( o' \) and a new ("remaining") command \( c' \).
  - \( c' \) serves as program counter
- An execution is a trace:
  \[ \langle \sigma, c \rangle \rightarrow \langle \sigma_1, c_1 \rangle \rightarrow \langle \sigma_2, c_2 \rangle \rightarrow \ldots \rightarrow \langle \sigma_n, c_n \rangle \]

Rule-based definition of \( \rightarrow \):

- \[ \sigma, x := e \rightarrow \sigma[x := [e] \sigma], \text{skip} \]
- \[ \sigma, c_1 \rightarrow o', c_1' \]
- \[ \sigma, c_1; c_2 \rightarrow o', c_1'c_2' \]
- \[ \sigma, \text{skip}; c_2 \rightarrow \sigma, c_2 \]

Back to Transition Relation

- We can choose the granularity we want.
  - Single statement, basic block, etc.
  - A popular choice: loop-free blocks

Example

\[ \Sigma \times \{a, b\} \rightarrow \{ \} = \Sigma \times \Sigma \]

\[ \text{step} \alpha \circ \sigma \rightarrow \langle c_1, c \rangle \rightarrow \alpha \circ \sigma \]

\[ \text{assume } a < b \]
- if \( a < b \) then
  \( b := b - a \)
- else
  \( a := a - b \)
### Reasoning with Transition Relation

- **We want to prove:**
  \[ \forall s : \Sigma. \ step^* s_0 s \rightarrow P s \]

- **Where:**
  - \( s_0 \) is (the/an) initial state
  - \( \text{step}^* \) is the transitive closure of \( \text{step} \)
  - \( P \) is a safety property, \( P : \Sigma \rightarrow \mathbb{P} \)

- **E.g.,**
  \[ s_0 = \begin{pmatrix} 1 & 2 \\ 5 & 4 \end{pmatrix} \]
  \[ P (a, b) \triangleq (a = b) \rightarrow a = \text{gcd} a, b \]

### Lab #5

- **McCarthy’s “91 function”**

  \[
  M(n) = \begin{cases} 
  n - 10 & \text{if } n > 100 \\
  M(M(n + 11)) & \text{if } n \leq 100 
  \end{cases}
  \]

  **def m(n):**

  ```python
  def m(n):
      a = 1
      while a > 2:
          if n > 100: n -= 10 \\
          else: n += 11
          a += 2
      return n
  ```

- **Prove:** For inputs 101 or less, \( M \) always returns 91.