Programming Language Design Paradigms

Introduction

- Programming languages are tools for creating software.
- Programming languages can be used in the design phase of software, although some languages are better suited:
  - Older languages do not support a specific design method (eg. Fortran)
  - Newer languages support specific methods:
    - Pascal → Top-down
    - Haskell → Functional
    - Java → OO
- When a compatible design method and language are used, design components can be mapped easily to code.

Paradigms

- A language may force a certain programming style, known as a programming paradigm.
- Imperative Languages: programs are decomposed into computation steps and routines are used to modularise the program.
  Eg. Fortran, Pascal, C.
- Functional Languages: based on mathematical theory of functions. Focus on what should be computed, not how. They use expressions that are solved by simplification.
  Eg. Haskell, SML, Caml, Clean.
- Object-Oriented Languages - emphasise a hierarchy of objects.
  Eg. Java, Smalltalk
- Logic Languages: define a problem rather than defining an algorithmic implementation
  Eg. Prolog
  Another Variant: Constraint Logic Languages
  Eg. Chip, Eclipse

Definition of a Programming Language

- Three main components:
  - Syntax: defines the form of programs; how expressions, commands and declarations are put together.
  - Semantics: gives the meanings of programs; how they behave when executed.
  - Implementation: a software system that can read a program and execute it on a machine

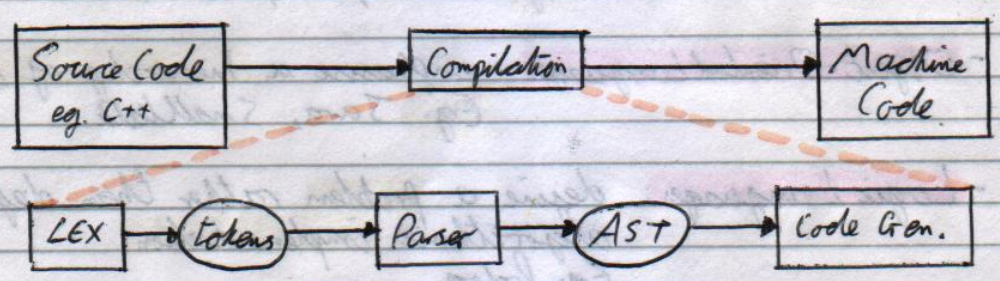
Implementation

- We are concerned mostly with high-level languages, which are more-or-less machine independent. These can be implemented by:
  - Compiling programs into machine language,
  - Interpreting programs,
  - or a Hybrid method combining the two above

- Compilation

- The compiler translates the source language into machine language that can be executed directly on the machine.
- Main compilation steps:
  - Lexical analysis
  - Syntax analysis (parsing)
  - Type + Semantics analysis
  - Machine code generation

- This method is fast



* AST = Abstract Syntax Tree

- Machine code execution takes place via the fetch-decode-execute cycle (CS1)

- Interpretation

  - There is no translation; a program is executed by a process that simulates a FDE cycle dealing with high-level instructions.
  - The interpreter produces a virtual machine for the language.
  - Pros: easier to implement, easier to debug, better error messages
  - Cons: very slow

- Hybrid Approaches

  - Some languages compile into an intermediate language, which is then interpreted.
  - This is more portable: programs can run on any machine with an interpretor for the intermediate language.
  - Eg: Java, with the intermediate bytecode.
  - Some languages have an interpreted version for dev and a compiled version for speed.

Syntax

- Concerned with the form of programs, built with an alphabet and rules.
- There are two forms:
  - Concrete Syntax: describes which chains of characters are well-formed programs.
  - Abstract Syntax: describes the syntax trees, to which a semantic is related

- Example: Integer Arithmetic

  - Concrete:
    $Exp ::= Num \mid Exp\ Op\ Exp$
    $Op ::= + \mid - \mid \times \mid \div$
    $Num ::= Digit \mid Digit\ Num$
    $Digit ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
  - Abstract:
    $e ::= n \mid op(e, e)$
    $op ::= + \mid - \mid \times \mid \div$

  - The abstract syntax is unambiguous. (consider 5-2-1)

Imperitive Languages

- These are abstractions of an underlying Vonn Neuman computer architecture.

- One of the main components is memory, storing instructions and data.
  - Memory cells are abstracted into variables.

Variables

- A variable is characterised by:
  - Name - usually a string of characters with some constraints.
  - Type - indicates the acceptable range of values.
  - Address - the memory address to which the variable is associated (aka. the l-value)

    The association is not simple:
    - Multiple names may map to one address
    - One name may map to different addresses in different parts of the program.
  - Value - the contents of the associated memory (aka. the r-value).
  - Lifetime - the time during which the variable is allocated a specific memory address.

- There are four classes of variable:
  - Static - bound to a memory location before execution and available throughout.

    Pro: efficient, as all addressing can be resolved at compile time.

    Con: storage cannot be reused.
  - Stack Dynamic - storage is allocated when the declaration is processed at run time

- Explicit Heap-dynamic - nameless memory cells that are allocated and deallocated by program instructions, using pointers.

  In Java, objects are EHD, and accessed via reference variables, but cannot be destroyed explicitly.

- Implicit Heap-dynamic - bound to storage only when assigned a value

- A variable has a specific scope, defining where in a program that variable is visible.
  - Most often local or global.

- There are two main types of scoping:

  - Static Scope - can be determined at compile time. Every time a variable is referenced, the compiler looks for its declaration locally, and then looks for declaration in the "closest parent"

  - Dynamic Scope - scope depends on the calling sequence of sub-programs instead of declaration. This means that static type checking is not possible and makes programs hard to understand.

- Variables are assigned as:

  varName = expr or varName := expr

Control Statements

- These are used to direct the flow of execution in a program.

- We consider these three:

  - Selection constructs
  - Looping constructs
  - Branching constructs

  } Any sequential algo could be coded with just these two.

- **Selection Statements**: choose between two or more execution paths
  - Two-way selectors: if-then-else
  - Multiple selectors: switch

- **Looping Statements**: repeat a set of statements
  - Counter-controlled: for
  - Logically-controlled: while (pre-test)
    do-while (post-test)

Structuring a Program

- **Blocks**: a group of statements, delimited by keywords like begin and end or symbols like { and }

  A block may contain declarations. If the program has static scope, these will be visible only in that block.

- **Subprograms**: a named block that can be invoked explicitly.

  We can associate a name and parameters.
  - When declared: formal parameters.
  - When invoked: actual parameters.

Formal Descriptions

- Informal descriptions can be imprecise.

- For example, the order of evaluation for function parameters is not specified for C, so

  f(++i, i++)

  may produce different results for different compilers.

- The first approach to giving a precise description of a programming construct's behaviour ~~was~~ with an abstract machine.

- An abstract machine is a transition system, which specifies an interpreter for the programming language.

- We give an example for the small imperative language SIMP.

Abstract Definition of SIMP

Programs

$$P ::= C \mid E \mid B$$

Commands

$$C ::= \text{skip} \mid l := E \mid C;C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C$$

Integer Expressions

$$E ::= !l \mid n \mid E \text{ op } E$$

$$\text{op} ::= + \mid - \mid * \mid \div$$

Boolean Expressions

$$B ::= \text{True} \mid \text{False} \mid E \text{ bop } E \mid \neg B \mid B \wedge B$$

$$\text{bop} ::= > \mid < \mid =$$

Where...

- $n \in \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (integers)
- $l \in L = \{l_0, l_1, l_2, \ldots\}$ (Locations / variables)
- $!l_0$ = the value stored in l_0

Example Programs

Swapping contents of x and y:

```
z := !x;
x := !y;
y := !z;
```

Factorial:

Assuming l contains n and we want $n!$

```
factorial := 1;
while !l > 1 do (
    factorial := !factorial * !l;
    l := !l - 1;
)
```

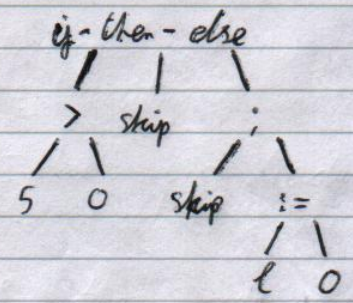
- Note on Abstract Syntax

  - The grammar given specifies the abstract syntax of SIMP, therefore trees not strings.

  - Leaf nodes are marked by elements of $Z \cup B \cup L \cup \{skip\}$

  - Internal nodes are marked by commands and operators.

  - Example:



$\Longrightarrow$ if $5 > 0$ then
skip
else
(skip; $l := 0$)

An Abstract Machine for SIMP

- We define a machine with:

  - a control stack, c

  - a results/auxiliary stack, r

  - a memory store, modelled by a partial function m mapping each address to an integer.

    - $m[l \mapsto n]$ is the function that maps from each $l' \neq l$ to the value $m(l')$, and from l to the value n.

    - More precisely: $m[l \mapsto n](l) = n$
      $m[l \mapsto n](l') = m(l')$ if $l \neq l'$

- The abstract machine is defined by a set of configurations and a set of transition rules.

Abstract Machine Configuration (SIMP)

- A configuration is a triple: $\langle$Control, Results, Memory$\rangle$
- The formal definition of the stack is given by:

$$c ::= nil \mid i \cdot c$$

$$i ::= P \mid op \mid \neg \mid \wedge \mid bop \mid := \mid if \mid while$$

$$r ::= nil \mid P \cdot r \mid l \cdot r$$

where...

P, bop and op are the non-terminals used in the rules defining programs.

- To model a program's execution, we define transitions between initial and final configurations.
- Initial Configuration: $\langle C \cdot nil, nil, m \rangle$
- Final Configuration: $\langle nil, nil, m \rangle$

Semantics of SIMP Programs

- If there is a sequence of transitions

$$(C \cdot nil, nil, m) \longrightarrow^{+} (nil, nil, m')$$

then we say that the program C executed in the state m terminates successfully producing the state m'.

- The semantics of a command C in the state m are described by giving the sequence of transitions that transform the configuration $(C \cdot nil, nil, m)$ into (nil, nil, m').
- If there is a sequence of transitions

$$(E \cdot c, r, m) \longrightarrow^{*} (c, v \cdot r, m')$$

then we say that the value of the expression E in the state m is v.

Abstract Machine Pros and Cons

- Pro: step-by-step explanations are useful if we have to implement the language.

- Con: not intuitive for programmers
  most transitions aren't doing any real computation.

Structural Approach

- To overcome this problem, a new approach to operation semantics of transition systems was developed: the structural approach.

- Idea: the structure of transition systems should reflect the structure of the language.

- In other words, the definition should be inductive.

Transition Rules

1. **Evaluation of Expressions:**

$$
\begin{aligned}
\langle n \cdot c, r, m\rangle &\rightarrow \langle c, n \cdot r, m\rangle \\
\langle b \cdot c, r, m\rangle &\rightarrow \langle c, b \cdot r, m\rangle \\
\langle \neg B \cdot c, r, m\rangle &\rightarrow \langle B \cdot \neg \cdot c, r, m\rangle \\
\langle (B_1 \ \wedge \ B_2) \cdot c, r, m\rangle &\rightarrow \langle B_1 \cdot B_2 \cdot \wedge \cdot c, r, m\rangle \\
\langle \neg \cdot c, b \cdot r, m\rangle &\rightarrow \langle c, b' \cdot r, m\rangle \quad \text{if } b' = not\ b \\
\langle \wedge \cdot c, b_2 \cdot b_1 \cdot r, m\rangle &\rightarrow \langle c, b \cdot r, m\rangle \text{ if } b_1\ and\ b_2 = b \\
\langle (E_1\ op\ E_2) \cdot c, r, m\rangle &\rightarrow \langle E_1 \cdot E_2 \cdot op \cdot c, r, m\rangle \\
\langle (E_1\ bop\ E_2) \cdot c, r, m\rangle &\rightarrow \langle E_1 \cdot E_2 \cdot bop \cdot c, r, m\rangle \\
\langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle &\rightarrow \langle c, n \cdot r, m\rangle \text{ if } n_1\ op\ n_2 = n \\
\langle bop \cdot c, n_2 \cdot n_1 \cdot r, m\rangle &\rightarrow \langle c, b \cdot r, m\rangle \text{ if } n_1\ bop\ n_2 = b \\
\langle !l \cdot c, r, m\rangle &\rightarrow \langle c, n \cdot r, m\rangle \quad \text{if } m(l) = n
\end{aligned}
$$

2. **Evaluation of commands:**

$$
\begin{aligned}
\langle skip \cdot c, r, m\rangle &\rightarrow \langle c, r, m\rangle \\
\langle (l := E) \cdot c, r, m\rangle &\rightarrow \langle E \cdot := \cdot c, l \cdot r, m\rangle \\
\langle := \cdot c, n \cdot l \cdot r, m\rangle &\rightarrow \langle c, r, m[l \mapsto n]\rangle \\
\langle (C_1; C_2) \cdot c, r, m\rangle &\rightarrow \langle C_1 \cdot C_2 \cdot c, r, m\rangle \\
\langle (if\ B\ then\ C_1\ else\ C_2) \cdot c, r, m\rangle &\rightarrow \langle B \cdot if \cdot c, C_1 \cdot C_2 \cdot r, m\rangle \\
\langle if \cdot c, True \cdot C_1 \cdot C_2 \cdot r, m\rangle &\rightarrow \langle C_1 \cdot c, r, m\rangle \\
\langle if \cdot c, False \cdot C_1 \cdot C_2 \cdot r, m\rangle &\rightarrow \langle C_2 \cdot c, r, m\rangle \\
\langle (while\ B\ do\ C) \cdot c, r, m\rangle &\rightarrow \langle B \cdot while \cdot c, B \cdot C \cdot r, m\rangle \\
\langle while \cdot c, True \cdot B \cdot C \cdot r, m\rangle &\rightarrow \langle C \cdot (while\ B\ do\ C) \cdot c, r, m\rangle \\
\langle while \cdot c, False \cdot B \cdot C \cdot r, m\rangle &\rightarrow \langle c, r, m\rangle
\end{aligned}
$$

Structural Operational Semantics

Reminder: Induction

- For any property $P(n)$ of natural numbers, to prove that $\forall n \in N. P(n)$ it is sufficient to show:
  - Base: $P(0)$
  - Inductive Step: $\forall n \in N, P(n) \Rightarrow P(n+1)$

Structural Induction

- The above practise can be applied to various data structures...

Lists

- An empty list is denoted by nil
- A non-empty list is denoted by cons(h, l), where h and l are the head and tail.
- To prove a property P of lists:
  - Base: $P(nil)$
  - Ind. Step: $P(l) \Rightarrow P(cons(h,l))$ for all h, l

Finite Labelled Trees

- To prove a property P of trees:
  - Base: $P(e)$ for all leaf nodes e
  - Ind. Step: for each tree constructor c (with $n \geq 1$ arguments):

$$\forall t_1 \ldots t_n . P(t_1) \wedge \ldots \wedge P(t_n) \Rightarrow P(c(t_1 \ldots t_n))$$

Integer Expressions

- To prove a property P for integer expressions in SIMP:
  - Base: - Prove $P(n)$ for all $n \in \mathbb{Z}$
    - Prove $P(!l)$ for all locations l
  - Ind. Step: for all integer expressions E, E' and operators op:

    Prove $P(E)$ and $P(E')$ implies $P(E \; op \; E')$

Note

- This all works because we deal with finite trees, so we can understand structural induction as induction on the size of the tree.

Application of Structural Induction

- We will prove the following:

  "SIMP guarantees that for any integer expression E appearing in a program working on memory m, if E uses locations that are defined in m then E is defined."

- In other words, for any configuration $\langle E \cdot c, r, m \rangle$ with some arbitrary c and r, then there is a configuration $\langle c, n \cdot r, m \rangle$ such that:

$$\langle E \cdot c, r, m \rangle \xrightarrow{*} \langle c, n \cdot r, m \rangle$$

- Formally, we must prove $\forall E. P(E)$ were P is...

$$\boxed{\begin{array}{l} \forall m. \text{locations}(E) \in \text{dom}(m) \Rightarrow \\ \qquad \exists n. \forall c. \forall r. \langle E \cdot c, r, m \rangle \xrightarrow{*} \langle c, n \cdot r, m \rangle \end{array}}$$

- This is proved by induction on the structure of E

(next page)

- **Base:** If E is a number n or $!\ell$ then the rules/transitions for constants and locations prove $P(E)$.

- **Ind. Step:** - Assume $P(E_1)$ and $P(E_2)$ hold

  - We now prove $P(E_1 \text{ op } E_2)$:

$$\langle (E_1 \text{ op } E_2) \cdot c, r, m \rangle \rightarrow \langle E_1 \cdot E_2 \cdot \text{op} \cdot c, r, m \rangle$$

$$\rightarrow^* \langle E_2 \cdot \text{op} \cdot c, n_1 \cdot r, m \rangle \text{ for some } n_1 \text{ by } P(E_1)$$

$$\rightarrow^* \langle \text{op} \cdot c, n_2 \cdot n_1 \cdot r, m \rangle \text{ for some } n_2 \text{ by } P(E_2)$$

$$\rightarrow^* \langle c, n \cdot r, m \rangle \text{ where } n = n_1 \text{ op } n_2$$

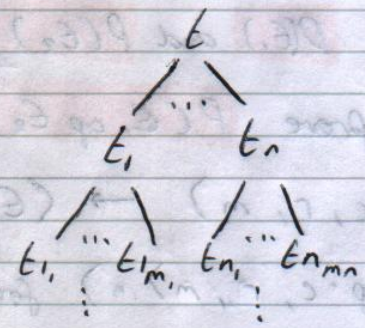
Inductive Definitions

- Induction can define subsets of a given set T.
- We write these definitions with axioms (base cases) and rules (ind. steps).

Definition

- An axiom is an element of T
- A rule is a pair (H, c) where...
  - H is a non-empty subset of T, called the rule hypothesis
  - c is an element of T, called the rule conclusion.
- The subset I of T defined by axioms A and rules R consists of those $t \in T$ such that:
  - $t \in A$ or
  - there are $\{t_1 \dots t_n\} \in I$ and a rule (H, c) such that $H = \{t_1 \dots t_n\}$ and $c = t$.

- To show that an element t of T is in I, show that t is an axiom, or that there is a proof:



where the leaves are axioms and for each non-leaf t_i there is a rule $(\{t_{i_1} \dots t_{i_n}\}, t_i)$.

- This style of proof is usually written as:

$$\dfrac{\dfrac{\vdots}{t_1} \quad \cdots \quad \dfrac{\vdots}{t_{1_{n_1}}} \qquad \cdots \qquad \dfrac{\vdots}{t_n}}{t}$$

Example: Natural Numbers

- **Axiom:** 0
- **Rule:** $\dfrac{n}{n+1}$ → "If you have n, you can deduce $n+1$."

Example: SIMP Int. Expressions

- **Notation:** $(E, m) \Downarrow n$ means E evaluates to n in the state m.
- **Axiom:** $(n, m) \Downarrow n$ for all $n \in \mathbb{Z}$
  $(!l, m) \Downarrow n$ if $l \in dom(m)$ and $m(l) = n$
- **Rule:** $\dfrac{(E_1, m) \Downarrow n_1 \qquad (E_2, m) \Downarrow n_2}{((E_1 \text{ op } E_2), m) \Downarrow n}$ if $n_1 \text{ op } n_2 = n$

Note

- We can write a principle for these kinds of definition, called Rule Induction.

Principle of Rule Induction

- Let I be a set defined by induction with the axioms and rules (A, R).

- To show that $P(i)$ holds for all $i \in I$:

  - Base: $\forall a \in A. P(a)$

  - Ind. Step: $\forall(\{h_1 \dots h_n\}, c) \in R. P(h_1) \wedge \dots \wedge P(h_n) \Rightarrow P(c)$

Example

"Each integer expression in SIMP has a unique value under the evaluation $\Downarrow$".

- Basis: trivial, as numbers have unique values.

- Ind. Step: for non-atomic expressions, the value of an arithmetic expression is uniquely determined by the value of its components, which are unique by induction of the hypothesis.

Special Case

- Sometimes we only want to prove a property for a subset J of the inductive set I defined by (A, R).

- Then we use a special case to show that $Q(x)$ holds for all $x \in J \subset I$.

- Basis: $\forall a \in (J \cap I). Q(a)$

- Ind. Step: for all $(H, c) \in R$ such that $c \in J$,

$$(\forall h \in H \cap J. Q(h)) \Rightarrow Q(c)$$

Structural Operational Semantics (for SIMP)

- This is an alternative definition for the semantics of SIMP (alternative to an abstract machine).

- Two styles of SOS:
  - Small-step semantics - using a reduction relation
  - Big-step semantics - using an evaluation relation

Small-Step Semantics

- We define a transition system with the configuration $\langle P, s \rangle$ where P is a SIMP program and s is a memory store, mapping locations to integers.

- Notation: $s[\ell \mapsto n]$ is the function that coincides with s, except it maps ℓ to n.

$$\boxed{\begin{aligned} s[\ell \mapsto n](\ell) &= n \\ s[\ell \mapsto n](\ell') &= s(\ell') \quad \text{if } \ell \neq \ell' \end{aligned}}$$

Basically, updating ℓ doesn't change anything else.

- The language is defined in small-step semantics by the axioms and rules on the next page.

- Note: there are no axioms or rules for:
  - $\langle n, s \rangle$ where n is an integer
  - $\langle b, s \rangle$ where b is a boolean
  - $\langle !\ell, s \rangle$ where $\ell \in dom(s)$
  - $\langle skip, s \rangle$

  These are terminal states.

- If we find the state $\langle !\ell, s \rangle$ where $\ell \notin dom(s)$, we say that the program is blocked.

For Expressions:

$$\frac{}{\langle !l, s\rangle \rightarrow \langle n, s\rangle,\ \text{if } s(l) = n}\ (\mathsf{var})$$

$$\frac{}{\langle n_1\ op\ n_2, s\rangle \rightarrow \langle n, s\rangle,\ \text{if } n = (n_1\ op\ n_2)}\ (\mathsf{op})$$

$$\frac{}{\langle n_1\ bop\ n_2, s\rangle \rightarrow \langle b, s\rangle,\ \text{if } b = (n_1\ bop\ n_2)}\ (\mathsf{bop})$$

$$\frac{\langle E_1, s\rangle \rightarrow \langle E_1', s'\rangle}{\langle E_1 op E_2, s\rangle \rightarrow \langle E_1' op E_2, s'\rangle}\ (\mathsf{op_L}) \quad \frac{\langle E_2, s\rangle \rightarrow \langle E_2', s'\rangle}{\langle E_1 op E_2, s\rangle \rightarrow \langle E_1 op E_2', s'\rangle}\ (\mathsf{op_R})$$

$$\frac{\langle E_1, s\rangle \rightarrow \langle E_1', s'\rangle}{\langle E_1 bop E_2, s\rangle \rightarrow \langle E_1' bop E_2, s'\rangle}\ (\mathsf{bop_L}) \quad \frac{\langle E_2, s\rangle \rightarrow \langle E_2', s'\rangle}{\langle E_1 bop E_2, s\rangle \rightarrow \langle E_1 bop E_2', s'\rangle}\ (\mathsf{bop_R})$$

$$\frac{}{\langle b_1 \wedge b_2, s\rangle \rightarrow \langle b, s\rangle,\ \text{if } b = (b_1\ and\ b_2)}\ (\mathsf{and})$$

$$\frac{}{\langle \neg b, s\rangle \rightarrow \langle b', s\rangle,\ \text{if } b' = not\ b}\ (\mathsf{not}) \quad \frac{\langle B_1, s\rangle \rightarrow \langle B_1', s'\rangle}{\langle \neg B_1, s\rangle \rightarrow \langle \neg B_1', s'\rangle}\ (\mathsf{notArg})$$

$$\frac{\langle B_1, s\rangle \rightarrow \langle B_1', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \rightarrow \langle B_1' \wedge B_2, s'\rangle}\ (\mathsf{and_L}) \quad \frac{\langle B_2, s\rangle \rightarrow \langle B_2', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \rightarrow \langle B_1 \wedge B_2', s'\rangle}\ (\mathsf{and_R})$$

For Commands:

$$\frac{\langle E, s\rangle \rightarrow \langle E', s'\rangle}{\langle l := E, s\rangle \rightarrow \langle l := E', s'\rangle}\ (:=_{\mathsf{R}}) \quad \frac{}{\langle l := n, s\rangle \rightarrow \langle skip, s[l \mapsto n]\rangle}\ (:=)$$

$$\frac{\langle C_1, s\rangle \rightarrow \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \rightarrow \langle C_1'; C_2, s'\rangle}\ (\mathsf{seq}) \quad \frac{}{\langle skip; C, s\rangle \rightarrow \langle C, s\rangle}\ (\mathsf{skip})$$

$$\frac{\langle B, s\rangle \rightarrow \langle B', s'\rangle}{\langle if\ B\ then\ C_1\ else\ C_2, s\rangle \rightarrow \langle if\ B'\ then\ C_1\ else\ C_2, s'\rangle}\ (\mathsf{if})$$

$$\frac{}{\langle if\ True\ then\ C_1\ else\ C_2, s\rangle \rightarrow \langle C_1, s\rangle}\ (\mathsf{if_T})$$

$$\frac{}{\langle if\ False\ then\ C_1\ else\ C_2, s\rangle \rightarrow \langle C_2, s\rangle}\ (\mathsf{if_F})$$

$$\frac{}{\langle while\ B\ do\ C, s\rangle \rightarrow \langle if\ B\ then\ (C; while\ B\ do\ C)\ else\ skip, s\rangle}\ (\mathsf{while})$$

Comparison to Abstract Machine

- Each transition does some part of the computation leading to the result.
  - In an AM, some transitions are just manipulating syntax.
- To show a sequence of reductions is valid, we need a proof.
  - Example: $\langle \text{if } !l > 0 \text{ then } (c';c) \text{ else skip}, s\rangle$

$$\rightarrow \langle \text{if } 4 > 0 \text{ then } (c';c) \text{ else skip}, s\rangle$$

This can be proved if $s(l) = 4$.

To prove this is valid, we use the transition rules to build a proof as follows:

$$\dfrac{\dfrac{\dfrac{}{\langle !l, s\rangle \rightarrow \langle 4, s\rangle \quad \text{if } s(l)=4}\ (\text{var})}{\langle !l > 0, s\rangle \rightarrow \langle 4 > 0, s\rangle}\ (\text{bop}_L)}{\langle \text{if } !l > 0 \text{ then } (c';c) \text{ else skip}, s\rangle \rightarrow \langle \text{if } 4 > 0 \text{ then } (c';c) \text{ else skip}, s\rangle}\ (\text{if})$$

Big-Step Semantics

- The previous system is deterministic: for a program $\langle P, s\rangle$ there is a unique set of transitions with a maximal length.
  - This is called the evaluation sequence for $\langle P, s\rangle$.
- The evaluation sequence may be finite or infinite.
- We classify three types of evaluation sequence:
  - Terminating: the sequence reaches a terminating, non-blocked state (see previous page).
    - Eg. $\langle$ if $4>0$ then skip else skip, $s\rangle$
  - Blocked: the sequence reaches a state $\langle !l, s\rangle$ where $l \notin dom(s)$.
    - Eg. $\langle !l, s\rangle$ where $s(l)$ is not defined.
  - Diverges: the sequence is infinite.
    - Eg. $\langle$ while True do skip, $s\rangle$

Transition System:

- The system on the next page defines all terminating evaluation sequences.
- It defines by induction the relation:

  $\langle P, s\rangle \xrightarrow{*} \langle P', s'\rangle$ such that $\langle P', s'\rangle$ is terminal.
- The notation for this is $\langle P, s\rangle \Downarrow \langle P', s'\rangle$.
- In short....

$$\boxed{\begin{array}{c}\langle P, s\rangle \Downarrow \langle P', s'\rangle \text{ if } \langle P, s\rangle \xrightarrow{*} \langle P', s'\rangle \\ \text{where } \langle P', s'\rangle \text{ is terminal}\end{array}}$$

$$\frac{}{\langle c, s\rangle \Downarrow \langle c, s\rangle,\ \textit{if}\ c \in Z \cup \{True, False\}}\ (\mathsf{const})$$

$$\frac{}{\langle !l, s\rangle \Downarrow \langle n, s\rangle,\ \textit{if}\ s(l) = n}\ (\mathsf{var})$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle \quad \langle B_2, s'\rangle \Downarrow \langle b_2, s''\rangle}{\langle B_1 \wedge B_2, s\rangle \Downarrow \langle b, s''\rangle,\ \textit{if}\ b = b_1\ \textit{and}\ b_2}\ (\mathsf{and})$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle}{\langle \neg B_1, s\rangle \Downarrow \langle b, s''\rangle,\ \textit{if}\ b = \textit{not}\ b_1}\ (\mathsf{not})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1\ op\ E_2, s\rangle \Downarrow \langle n, s''\rangle,\ \textit{if}\ n = n_1\ op\ n_2}\ (\mathsf{op})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1\ bop\ E_2, s\rangle \Downarrow \langle b, s''\rangle,\ \textit{if}\ b = n_1\ bop\ n_2}\ (\mathsf{bop})$$

$$\frac{}{\langle skip, s\rangle \Downarrow \langle skip, s\rangle}\ (\mathsf{skip}) \qquad \frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle l := E, s\rangle \Downarrow \langle skip, s'[l \mapsto n]\rangle}\ (:=)$$

$$\frac{\langle C_1, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\mathsf{seq})$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle \textit{if}\ B\ \textit{then}\ C_1\ \textit{else}\ C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\mathsf{if_T})$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle \textit{if}\ B\ \textit{then}\ C_1\ \textit{else}\ C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\mathsf{if_F})$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C, s'\rangle \Downarrow \langle skip, s''\rangle \quad \langle \textit{while}\ B\ \textit{do}\ C, s''\rangle \Downarrow \langle skip, s'''\rangle}{\langle \textit{while}\ B\ \textit{do}\ C, s\rangle \Downarrow \langle skip, s'''\rangle}\ (\mathsf{while_T})$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle}{\langle \textit{while}\ B\ \textit{do}\ C, s\rangle \Downarrow \langle skip, s'\rangle}\ (\mathsf{while_F})$$

Example (Big-Step)

- Consider the program P: $(z := !x \,;\, x := !y) \,;\, y := !z$

- Let s be the memory such that: $s(z) = 0$
  $s(x) = 1$
  $s(y) = 2$

- We can prove that $\langle P, s\rangle \Downarrow \langle skip, s'\rangle$

  where $s'(z) = 1$
  $s'(x) = 2$
  $s'(y) = 1.$

- Taking the first section:

seq rule

$$\dfrac{\dfrac{\langle !x, s\rangle \Downarrow \langle 1, s\rangle}{\langle z := !x, s\rangle \Downarrow \langle skip, s[z \mapsto 1]\rangle} \qquad \dfrac{\langle !y, {}^{*}s\rangle \Downarrow \langle 2, s[z \to 1]\rangle}{\langle x := y, {}^{*}s\rangle \Downarrow \langle skip, s\begin{bmatrix} z \mapsto 1 \\ x \mapsto 2 \end{bmatrix}\rangle}}{\langle z := !x \,;\, x := !y, s\rangle \Downarrow \langle skip, s\begin{bmatrix} z \mapsto 1 \\ x \mapsto 2 \end{bmatrix}\rangle}$$

- Combine with the second section:

seq rule

$$\dfrac{\langle z := !x \,;\, x := !y, s\rangle \Downarrow \langle skip, s\begin{bmatrix} z \to 1 \\ x \to 2 \end{bmatrix}\rangle \qquad \dfrac{\langle !z, s\begin{bmatrix} z \mapsto 1 \\ x \mapsto 2 \end{bmatrix}\rangle \Downarrow \langle 1, s\begin{bmatrix} z \mapsto 1 \\ x \mapsto 2 \end{bmatrix}\rangle}{\langle y := !z, s\begin{bmatrix} z \mapsto 1 \\ x \mapsto 2 \end{bmatrix}\rangle \Downarrow \langle skip, s\begin{bmatrix} z \mapsto 1 \\ x \mapsto 2 \\ y \mapsto 1 \end{bmatrix}\rangle}}{\langle P, s\rangle \Downarrow \langle skip, s\begin{bmatrix} z \mapsto 1 \\ x \mapsto 2 \\ y \mapsto 1 \end{bmatrix}\rangle}$$

* Should be $s[z \mapsto 1]$

Adding Variable Declaration to SIMP

- We can add local declarations to SIMP by using a local state in which the scope of a newly created location corresponds to the block where it was created (i.e. static scope).
- For this, we need a stack-based implementation of the state.
- Syntax: we extend SIMP to add blocks:

$$C ::= \text{begin loc } x := E;\ C \text{ end}$$

"loc" = "local var"

- Semantics: we add the following big-step rule:

$$\frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle \qquad \langle (\{x \mapsto \ell\}, s'[\ell \mapsto n]\rangle \Downarrow \langle \text{skip}, s''[\ell \mapsto n']\rangle}{\langle \text{begin loc } x := E;\ C \text{ end}, s\rangle \Downarrow \langle \text{skip}, s''\rangle}$$

where...

- $\ell \notin \text{dom}(s') \cup \text{dom}(s'') \cup \text{locations}(C)$ i.e. ℓ is a fresh name.
- $C\{x \mapsto \ell\}$ is the program C where all occurrences of x are replaced by ℓ to avoid confusion with other variables with the same name in other parts of the program.

Example:

- This program P swaps x and y using a local variable:

```
begin
    loc temp := !x;
    x := !y;
    y := !temp;
end
```

- To show that P is correct, first we prove:

$$\langle x := !y;\ y := !\ell,\ s[\ell \mapsto s(x)]\rangle \Downarrow \langle \text{skip}, s[x \mapsto s(y), y \mapsto s(x)]\rangle$$

as in the previous example.

- Lets call s' the store $s[x \mapsto s(y), y \mapsto s(x)]$, then:

$$\frac{\langle !x, s\rangle \Downarrow \langle s(x), s\rangle \qquad \langle x := !y;\ y := !t,\ s[t \mapsto s(x)]\rangle \Downarrow \langle skip, s'\rangle}{\langle P, s\rangle \Downarrow \langle skip, s'\rangle}$$

Functional Programming

General Concepts

- Programs consist entirely of functions.
- A function can be defined in terms of other functions, either previously defined, in a library, or language primitives.
- Focus is on what to compute, not how to compute it
- Modern functional languages are strongly typed and have built-in memory management.
- Pros:
  - shorter
  - easy to read
  - easy to maintain
- Cons:
  - slower

Syntax

- The notation is defined by mathematical notation of functions, using equations.
- We use a generic syntax, similar to Haskell
- Eg: square x = $x * x$
  - When we type square 5 the computer will print 25.

Evaluation

- To evaluate square 5, the computer replaced the expression with 5*5, using the definition of square. Using the predefined * operation, the computer replaced 5*5 with 25.
- The evaluation process is a series of these steps, called either a simplification, reduction or evaluation process.
- The goal is to obtain the value or normal form associated with an expression.
- There may be several possible sequences for an expression.

Properties

- Unicity of Normal Forms : the value of an expression is solely determined by its components, irrespective of the order of simplification.

- Non-Termination : some reduction sequences do not terminate.

Strategies

- The strategy of a language defines the reduction sequence it implements.

- Two popular options:

  - Call-by-name (normal order):
    reduce the application using the function's definition, then the arguments.

  - Call-by-value (applicative order):
    reduce the arguments, then the application.

- Different strategies require more or less reduction steps.

- Call-by-name always finds the value, if there is one.

- Call-by-value is often more efficient, but may fail to find a value.

Functional Values

- Functions are values, although we cannot display them.

- A function maps an item of type A (domain) to type B (codomain).

$$f : A \rightarrow B$$

- The Haskell notation to define this is:

```
square :: Integer -> Integer

prime :: Integer -> Boolean
```

Notation Notes

- Application is denoted by juxtaposition. Eg. (square 3)
- Application has precedence over other operators:

  square 3 + 1 means (square 3) + 1

- Application associates to the left:

  square square 3 means (square square) 3

  ← which is wrong!

- Outermost brackets are ignored.

Function Definition Syntax

- Functions are defined as equations.
  - Eg: square x = x * x

    min x y = if x ≤ y then x else y

- We can also write conditional / guarded equations:
  - Eg.

```
min x y
  | x ≤ y  = x
  | x > y  = y

sign x
  | x < 0  = -1
  | x == 0 = 0
  | x > 0  = 1
```

- Recursion is fine:
  - eg

```
fact :: Integer → Integer
fact n
  | n > 0  = n × (fact (n-1))
  | n == 0 = 1
  | n < 0  = error "negative input"
```

- error is a predefined function to terminate with an error message.

Local definitions

- We can write:

$$\left.\begin{array}{l} f\ x = a + 1 \text{ where } a = x \div 2 \\ \quad \text{or} \\ f\ x = \text{let } a = x \div 2 \text{ in } a + 1 \end{array}\right\} \text{equivalent.}$$

- They keywords where and let/in allow us to create local definitions, valid only on the RHS of the current expression.

- We can write several at once:

```
f x = square (next x) where
        square x = x * x;
        next x = x + 1
```

- Evaluation:
  - $e[a]$ where $a = e'$
  - evaluate e' to obtain r
  - replace a with r in e
  - evaluate $e\{a \mapsto r\}$

Arithmetic Functions

- Arith. operators are also functions
- Usually used in infix form: $5 + 2 \Rightarrow 7$
- Can be used prefix using brackets: $(+)\ 5\ 2 \Rightarrow 7$

Function Composition

- Functions can be combined by composition, denoted by $\cdot$
- Composition is itself a function:

$$(\cdot) :: (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

$$(f \cdot g)\ x = f\ (g\ x)$$

- We can only compose functions with matching types.

Types: General Concepts

- Values are divided into classes called types
- Predefined types:
  - Booleans, characters, numbers, etc.
    Haskell: Bool, Char, Int, Integer, Float...
  - Structured types: tuples, strings, lists
    Haskell: the type [Int] is a list of integers
  - Function types: Int → Int, Int → Bool, etc are arrow types.
    Arrows associate to the right.
- User defined types: yeah, these are a thing
- Static typing
  - Every valid expression must have a type.
  - Expressions that cannot be typed are rejected by the compiler without evaluation.
  - Pro: fail early.
  - Pro: help in s/w design.

Polymorphism

- Type systems can be:
  - monomorphic - every expression has one type
  - polymorphic - an expression can have multiple types
- Eg:

$$(.) :: (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

α, β, γ are type variables, so $\cdot$ is polymorphic.

Overloading

- Formally, the language of polymorphic types is defined as a set of terms built out of type variables (eg. α, β, γ) and type constructors, which are atomic (Bool, Int, ...) or take arguments (eg. $T_1 \to T_2$ [T]).

- A polymorphic type represents the set of its instances, obtained by substituting type variables by types.

- Overloading (aka. ad-hoc polymorphism) is a related notion where several functions with different types share a name.

- Example: (+) can operate on Integers or Reals.
  - $(+) :: \alpha \to \alpha \to \alpha$ is too general; it would allow anything.
  - There are 3 solutions to this:
    - Use two different symbols, eg + and +. in Caml.
    - Enrich the language of type definition:

      $+ :: (Int \to Int \to Int) \wedge (Float \to Float \to Float)$

    - Define a notion of a type class:

      $(+) :: Num\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$

      (+) has the type $\alpha \to \alpha \to \alpha$ where α is in the class Num.

Type Inference

- Modern languages do not require the type to be explicitly stated, because the compiler is able to infer a type, if one exists.

- Expressions are decomposed to their atoms, then the available information is used to determine a type.

Type System for a Subset of Haskell

- **Function:**
  If $f\ x_1 \ldots x_n = t$ and $\Gamma, x_1 :: \sigma_1 \ldots x_n :: \sigma_n \vdash t :: \tau$
  then $\Gamma \vdash f :: \sigma_1 \to \ldots \to \sigma_n \to \tau$

- **Variable:**
  $\Gamma, x :: \sigma \vdash x :: \sigma$

- **Application:**
  If $\Gamma \vdash s :: \sigma \to \tau$ and $\Gamma \vdash t :: \sigma$
  then $\Gamma \vdash (s\ t) :: \tau$

- **Eg.**

  square $x = x * x$

  It is a function, so the most general type is $\gamma \to \beta$

  The result is $x * x$, so $x * x :: \beta$ and $x :: \gamma$

  $(*) :: Int \to Int \to Int$ (primitive)

  $\therefore\ x :: Int$ and $x * x :: Int$

  $\therefore\ \gamma = \beta = Int$

  $\therefore$ square $:: Int \to Int$

Disjoint Sums

- A type can have several constructors.
- Eg. data Nat = Zero | Succ Nat ← Not polymorphic

  "Zero" and "Succ" are constructors
- Eg. data Seq α = Empty | Cons α (Seq α) ← Polymorphic

  "Empty" and "Cons" are constructors
- Constructors are used to build terms.
- Constructors are different from functions because they have no definition and can be used in patterns, eg.

  isempty Empty = true

  isempty (Cons x q) = false

Induction

- We can reason with recursive types using structural induction.
- In the case of Nat:
  - Prove that P(Zero) holds
  - Prove that if P(n) holds then P(Succ n) holds.

SFUN

- A basic functional language dealing with integers and booleans.

Syntax

- Let V be a set of variables $\{x, y, z, x_1, x_2, \dots\}$
- Let F be a set of functions $\{f_1 \dots f_k\}$ each with fixed arity $ar(f_i) = a_i$
- Terms in SFUN are defined by:

$$t ::= n \mid b \mid x \mid t_1 \; op \; t_2 \mid t_1 \; bop \; t_2 \mid \neg t_1 \mid t_1 \wedge t_2 \mid \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid f_i(t_1 \dots t_{a_i})$$

where...

n = an integer constant
b = a boolean constant
$x \in V$
$op ::= + \mid - \mid * \mid /$
$bop ::= > \mid < \mid =$

- $Var(t)$ denotes the variables used in term t
- A term t is "closed" if $Var(t) = \{\}$
- A program in SFUN is a set of recursive equations:

$$f_1(x_1 \dots x_{a_1}) = t_1$$
$$\vdots$$
$$f_k(x_1 \dots x_{a_k}) = t_k$$

such that...

- $t_1 \dots t_k$ at terms
- $Var(t_i) \subseteq \{x_1 \dots x_{a_i}\}$, $\forall i$ in the range $1 \leq i \leq k$
- Each f_i has only one equation

- Equations are recursive, so a term t_i may contain occurrences of $f_1 \dots f_k$.

Call-by-Value Evaluation. of SFUN

$$\frac{}{n \Downarrow_P n}\,(\mathsf{n}) \qquad \frac{}{b \Downarrow_P b}\,(\mathsf{b})$$

$$\frac{t_1 \Downarrow_P n_1 \quad t_2 \Downarrow_P n_2}{t_1\ op\ t_2 \Downarrow_P n}\,(\mathsf{op})\ \texttt{if}\ n_1\ op\ n_2 = n$$

$$\frac{t_1 \Downarrow_P n_1 \quad t_2 \Downarrow_P n_2}{t_1\ bop\ t_2 \Downarrow_P b}\,(\mathsf{bop})\ \texttt{if}\ n_1\ bop\ n_2 = b$$

$$\frac{t_1 \Downarrow_P b_1 \quad t_2 \Downarrow_P b_2}{t_1\ \wedge\ t_2 \Downarrow_P b}\,(\mathsf{and})\ \texttt{if}\ b_1 \wedge b_2 = b$$

$$\frac{t \Downarrow_P b}{\neg t \Downarrow_P b'}\,(\mathsf{not})\ \texttt{if}\ b' = \neg b$$

$$\frac{t_0 \Downarrow_P True \quad t_1 \Downarrow_P v_1}{\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Downarrow_P v_1}\,(\mathsf{If_T})$$

$$\frac{t_0 \Downarrow_P False \quad t_2 \Downarrow_P v_2}{\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Downarrow_P v_2}\,(\mathsf{If_F})$$

$$\frac{t_1 \Downarrow_P v_1 \quad \ldots \quad t_{a_i} \Downarrow_P v_{a_i} \quad d_i\{x_1 \mapsto v_1, \ldots, x_{a_i} \mapsto v_{a_i}\} \Downarrow_P v}{f_i(t_1, \ldots, t_{a_i}) \Downarrow_P v}\,(\mathsf{fn})$$

Example: Call By Value

- Let P be the program:

$$f_1 = f_1 + 1$$

$$f_2(x) = 1$$

$$f_3(x) = x * x$$

- In this program, $f_2(0) \Downarrow_P 1$

  - Proof

$$\dfrac{\dfrac{}{0 \Downarrow_P 0}(n) \qquad \dfrac{}{1\{x \mapsto 0\} \Downarrow_P 1}(n)}{f_2(0) \Downarrow_P 1}(fn)$$

- In this program, $f_2(f_1)$ has no value.

  - There is no such v that $f_2(f_1) \Downarrow_P v$

- In this program, $f_3(2+1)$ has the value 9

  - Proof:

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}(n) \quad \dfrac{}{1 \Downarrow_P 1}(n)}{2+1 \Downarrow_P 3}(op) \qquad \dfrac{\dfrac{}{3 \Downarrow_P 3}(n) \quad \dfrac{}{3 \Downarrow_P 3}(n)}{x * x\{x \mapsto 3\} \Downarrow_P 9}(op)}{f_3(2+1) \Downarrow_P 9}(fn)$$

Determinism

- For any term t, $t \Downarrow_P v_1$ and $t \Downarrow_P v_2$ imply $v_1 = v_2$
- Proof by induction:

Base:
- if t is a number n, then $n \Downarrow_P n$ using the (n) axiom, and this is the only axiom/rule that applies.
- If t is a boolean b, then $b \Downarrow_P b$ using the (b) axiom, and this is the only axiom/rule that applies.
- There is only one value for t in both cases.

Ind:
- For each term, there is only one rule that can be applied.
- For the (fn) rule:
  - Assume t is the term $f(t_1 \ldots t_{a_i})$
  - $t \Downarrow_P v$ iff...

    $t_1 \Downarrow_P v_1, \ldots, t_{a_i} \Downarrow_P v_{a_i}$

    and

    $d_i\{x_1 \mapsto v_1, \ldots, x_{a_i} \mapsto v_{a_i}\} \Downarrow_P v$
  - By induction, there is at most one value for $t_1 \ldots t_{a_i}$
  - " " , " " " " " " " $d_i\{\ldots\}$
  - $\therefore$ v is uniquely determined in $t \Downarrow_P v$.

Call-By-Name Evaluation

- We replace only the (fn) rule:

$$\frac{d_i\{x_1 \mapsto t_1, \ldots, x_{a_i} \mapsto t_{a_i}\} \Downarrow_P v}{f_i(t_1, \ldots, t_{a_i}) \Downarrow_P v}\ (fn_N)$$

- The system is still deterministic.

- Examples:

Let P be the program:
$$f_1 = f_1 + 1$$
$$f_2(x) = 1$$
$$f_3(x) = x * x$$

- $f_2(0) \Downarrow_P 1$
- $f_2(f_1) \Downarrow_P 1$ (compare with prev. example)
- $f_3(2+1) \Downarrow_P 9$

Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}(n) \quad \dfrac{}{1 \Downarrow_P 1}(n)}{2+1 \Downarrow_P 3}(op) \quad \dfrac{\dfrac{}{2 \Downarrow_P 2}(n) \quad \dfrac{}{1 \Downarrow_P 1}(n)}{2+1 \Downarrow_P 3}(op)}{x * x\{x \mapsto 2+1\} \Downarrow_P 9}(op)}{f_3(2+1) \Downarrow_P 9}(fn_N)$$

SFUN Type System

- The grammar for SFUN allows crap like 2 * True.
- The semantics consider only well-typed terms.
- The set of well-typed terms can be defined by:

$$\Gamma \vdash_{\varepsilon} M : T$$

where...

Γ is an environment (a finite partial function from variables to types) or "variable environment"

ε is a function environment, assigning a type to each function, respecting arity.

M is a SFUN term

T is a type:

$$b ::= int \mid bool$$

$$T ::= (b_1 \dots b_n) \rightarrow b$$

- $\Gamma \vdash_{\varepsilon} M : T$ can be read as "if the variable x has the type $\Gamma(x)$ for each $x \in dom(\Gamma)$ and the functions $f_1 \dots f_k$ have the types $\varepsilon(f_1) \dots \varepsilon(f_k)$, then the term M has the type T."

- A program in SFUN

$$f_1(x_1 \dots x_{a_1}) = t_1$$
$$\vdots$$
$$f_k(x_1 \dots x_{a_k}) = t_k$$

is typeable if for each equation $f_i(x_1 \dots x_{a_i}) = t_i$ there is a type T_i and an environment Γ_i such that $\Gamma_i \vdash_{\varepsilon} f_i(x_1 \dots x_{a_i}) : T_i$ and $\Gamma_i \vdash_{\varepsilon} t_i : T_i$

Typing SFUN Terms

Type System Axioms:

$$\overline{\Gamma \vdash_\varepsilon b{:}\mathsf{bool}} \qquad \overline{\Gamma \vdash_\varepsilon n{:}\mathsf{int}} \qquad \overline{\Gamma \vdash_\varepsilon x{:}\sigma}\ \texttt{if}\ \Gamma(x) = \sigma$$

Type System Rules:

$$\frac{\Gamma \vdash_\varepsilon M_1{:}\mathsf{int} \quad \Gamma \vdash_\varepsilon M_2{:}\mathsf{int}}{\Gamma \vdash_\varepsilon M_1\ op\ M_2{:}\mathsf{int}}$$

$$\frac{\Gamma \vdash_\varepsilon M_1{:}\mathsf{int} \quad \Gamma \vdash_\varepsilon M_2{:}\mathsf{int}}{\Gamma \vdash_\varepsilon M_1\ bop\ M_2{:}\mathsf{bool}}$$

$$\frac{\Gamma \vdash_\varepsilon M_1{:}\mathsf{bool} \quad \Gamma \vdash_\varepsilon M_2{:}\mathsf{bool}}{\Gamma \vdash_\varepsilon M_1 \wedge M_2{:}\mathsf{bool}} \qquad \frac{\Gamma \vdash_\varepsilon M{:}\mathsf{bool}}{\Gamma \vdash_\varepsilon \neg M{:}\mathsf{bool}}$$

$$\frac{\Gamma \vdash_\varepsilon B{:}\mathsf{bool} \quad \Gamma \vdash_\varepsilon M_1{:}\sigma \quad \Gamma \vdash_\varepsilon M_2{:}\sigma}{\Gamma \vdash_\varepsilon \texttt{if}\ B\ \texttt{then}\ M_1\ \texttt{else}\ M_2{:}\sigma}$$

$$\frac{\Gamma \vdash_\varepsilon t_1{:}\sigma_1 \quad \cdots \quad \Gamma \vdash_\varepsilon t_{a_i}{:}\sigma_{a_i}}{\Gamma \vdash_\varepsilon f_i(t_1, \ldots, t_{a_i}){:}\sigma}\ \texttt{if}\ \varepsilon(f_i) = (\sigma_1, \ldots, \sigma_{a_i}) \rightarrow \sigma$$

Logic Programming Languages

- Use logic to express knowledge / a problem
- Use inference to manipulate knowledge and obtain a solution
- Popular : Prolog
  - Logic : clausal classical first-order logic
  - Inference : resolution

Pros

- The program just describes the problem.
- It has a declarative style.
- Precise and simple semantics.
- The same formalism can specify, write and prove a program.
- The same program can have ~~many~~ many uses.

Cons

- Inefficient.
- Most languages are restricted to a given part of classical first-order logic

Domain of Computation

- A set of terms defined over a universal alphabet of
  - variables, such as X, Y, Z, etc
  - functions with fixed arity (0-arity = constant)
- A term is a variable or has the form $f_n(t_1 \dots t_n)$
- Eg. $f(f(X, g(a)), Y)$

  a = constant
  g = unary function
  f = binary function
  X, Y = variables

Programs

- Sets of definite clauses
  - A disjunction of literals with at most one positive literal
  - Literals are built from terms and predicates
- A definite clause

$$P_1 \vee \neg P_2 \vee \dots \vee \neg P_n$$

is written as

$$P_1 :- P_2, \dots, P_n.$$

- A clause with only P_1 is written as $P_1.$ ← a "fact"
- A clause with only negative literals is written $:- P_1 \dots P_n.$ ← a "goal" or "query"

Values

- Values are also terms, associated to variables by means of automatically generated substitutions, called most general unifiers
- Substitutions are partial mappings of variables to terms with a finite domain.
- We apply a substitution σ to a term t or literal l by simultaneously replacing each variable occurring in $dom(\sigma)$ by the corresponding term. The resulting term is $t\sigma$
- Eg. $\sigma = \{X \mapsto g(Y), Y \mapsto a\}$

$$t = f(f(X, g(Y)), Y)$$

$$t\sigma = f(f(g(Y), g(a)), a)$$

Control Mechanism

- In a pure form, logical languages are declarative.
- They admit two interpretations:
  - a declarative interpretation, in which the meaning of the program is defined with respect to a mathematical model - it corresponds to denotational semantics
  - a procedural interpretation, which explains how the program is used in computations - this is the operational semantics.
- The operational semantics of Prolog are based on SLD-Resolution with automatic backtracking.
- Prolog supports interactive programming: the user can submit a query and ask for one or more solutions.
- This is analogous to the way functional languages are used:
  - In func. languages, interaction is achieved by means of expressions that are evaluated with respect to a collection of function definitions.
  - In logic languages, queries are resolved with respect to a collection of predicate definitions.

Prolog Predicates

- Eg.

```
based (prolog, logic).
based (haskell, maths).
likes (max, logic).
likes (claire, maths).
likes (X, P) :- based (P, Y), likes (X, P).
```

- In the program

```
append ([], L, L).
append ([X|L], Y, [X|Z]) :- append (L, Y, Z).
```

the atomic formula append (S, T, U) expresses that the result of appending T on the end of S is U.

- The clauses in a logic program define predicates.

Solving Prolog Queries

- A goal such as append([0], [1,2], X). are questions to be solved using definitions in the program.
- Eg. ① Goal: append([0], [1,2], U)
  ② We know: append([X|L], Y, [X|Z]) :- append(L, Y, Z).
  ③ Substitution: $\{X \mapsto 0, L \mapsto [], Y \mapsto [1,2], U \mapsto [0|Z]\}$
  ④ This unifies both sides of the predicate ②
  ⑤ We now have to prove that append([], [1,2], Z) holds
  ⑥ We know append([], L, L) so we take $\{Z \mapsto [1,2]\}$
  ⑦ ∴ $\{U \mapsto [0,1,2]\}$ solves the problem.
- This is based on the principle of resolution.

Operational Semantics: Unification

- A unification problem $\mathcal{U}$ is a set of equations between terms containing variables.
  - Eg. $\{s_1 = t_1, \dots, s_n = t_n\}$
- A solution/unifier to $\mathcal{U}$ is a substitution σ such that when we apply σ to all terms in $\mathcal{U}$ we obtain syntatical identities such that for any $s_i = t_i$ the terms $s_i\sigma$ and $t_i\sigma$ coincide.
- The most general unifier of $\mathcal{U}$ is a unifier σ such that any other unifier ρ is an instance of σ
- To find an MGU, the algo simplifies/transforms the equations until a substitution is generated.
- These transformations follow rules that either produce new equations or a failure.

Unification Algorithm:

- **Input**: a finite set of term equations
  - $\{s_1 = t_1, \ldots, s_n = t_n\}$
- **Output**: a substitution for the terms that is an MGU, or failure.
- **Transformation rules**: applied in a non-deterministic order until no rule can be applied or failure arises.

① $f(s_1 \ldots s_n) = f(t_1 \ldots t_n), E \longrightarrow s_1 = t_1, \ldots, s_n = t_n, E$

② $f(s_1 \ldots s_n) = g(t_1 \ldots t_m), E \longrightarrow$ failure

③ $X = X, E \longrightarrow E$

④ $t = X, E \longrightarrow X = t, E$ if t is not a variable

⑤ $X = t, E \longrightarrow X = t, E\{X \mapsto t\}$ if $X \notin t$ and $X \in E$

⑥ $X = t, E \longrightarrow$ failure if $X \in t$ and $X \neq t$

- Cases ① and ② also apply with constants: in the first case the equation is deleted; in the second we have failure.
  - Think of constants as zero-arity functions

- **Example:**

Start with: $\{f(a,a) = f(X,a)\}$

Apply ①: $\{a = X, a = a\}$

Apply ④: $\{X = a, a = a\}$

Apply ③: $\{X = a\}$

Done: $\{X \mapsto a\}$

Principle of Resolution

- To solve a query

  $:- A_1, \ldots, A_n$

  with respect to a set P of clauses, resolution tries to show that

  $P, \neg A_1, \ldots, \neg A_n$

  is a contradiction. It is based on negation.

- Contradiction is obtained when we find $A, \neg A$.

- If a contradiction does not arise directly, new literals are derived using resolution using the clauses until a contradiction is found, or the search goes on forever.

  - The derived literals are called resolvants

Computing Resolvants w/ SLD-Resolution

- If a query has several atoms $:- A_1 \ldots A_k$ the resolvant is computed between the first atom and a program clause to obtain $S_1\sigma, \ldots S_m\sigma, A_2\sigma \ldots A_k\sigma$.

  - When we compute a resolvant using a fact, (ie. when $m=0$) we remove that atom.

- An empty resolvant indicates contradiction, shown by $\diamond$, and the substitution computed is the answer to the goal.

- TL;DR: keep applying rules until we get an empty resolvant.

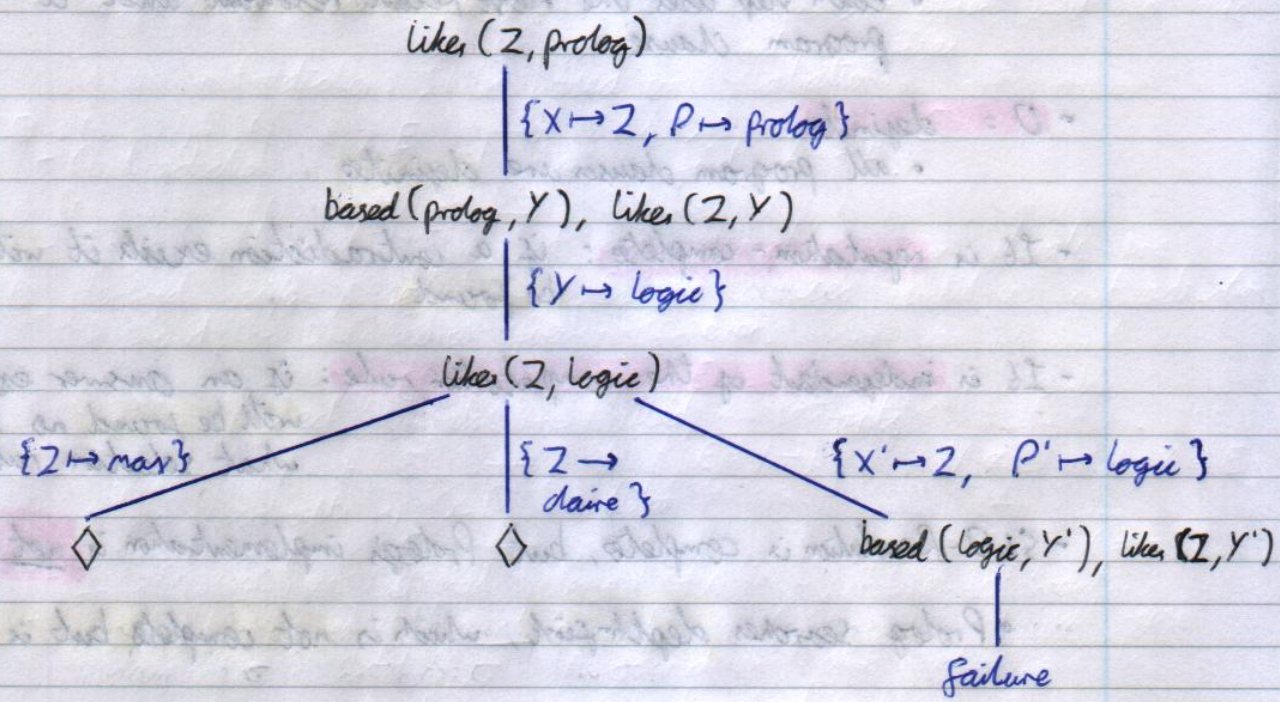
- When an empty resolvant is reached, the composition of all substitutions applied at each step (restricted to the variables of the query) is the answer.

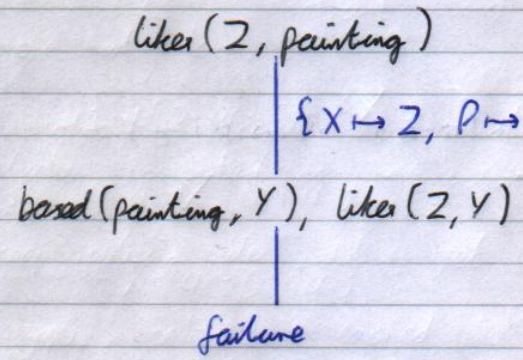
- Since there may be many clauses in the prog. that generate a resolvant, we get an SLD-resolution tree.

  - Each branch that leads to $\diamond$ = success
  - Each branch that doesn't = failure
  - Infinite branches may happen.

- Example.

```
based (prolog, logic).
based (haskell, maths).
likes (max, logic).
likes (claire, maths).
likes (claire, logic).
likes (X,P) :- based (P,Y), likes (X,Y).
```





- <u>NOTE</u>: Prolog builds the tree using clauses in the order they appear, in a depth-first manner.

  - It is very important to put "base case" clauses first to prevent infinite branches.

SLD-Resolution in Prolog

- S = selection rule
  - a fixed computation rule is applied in each step, in order to select an atom to resolve.
  - Prolog always picks the left-most atom

- L : linear
  - each step uses the most recent resolvant and a program clause.

- D = definite
  - all program clauses are definite

- It is refutation-complete : if a contradiction exists it will be found

- It is independent of the computation rule : if an answer exists, it will be found no matter what selection rule is used

- SLD-Resolution is complete, but Prolog's implementation is <u>not</u>.
  - Prolog searches depth-first, which is not complete but is efficient.