Chapter 1

- Description of background ideas, and the module itself.
- Review some mathematics.

Overview: Background Introduction to MC308

- What is a Language?
- What is a Programming Language?
- What is Syntax?
- What is Semantics?

Some Answers

- Two kinds of language
  - Natural language:
    - Recognized method of communicating thoughts and feelings:
    - **speech**, **hand signals**, **sending gifts** . . .
  - Formal language: A rigourously defined "system" to convey meaning or information.
- We do not have a precise definition of *language*. Try looking up *language* in, say, the **Cambridge Encyclopaedia of Language**.

- Programming Languages are formal languages used to “communicate” with a “computer”.
- Programming languages may be “low level”. They give direct instructions to the computer (machine code).
- Programming languages may be “high level”. The instructions given to the computer are indirect, but much closer to general concepts understood by the user (Java, C++, . . .).

- **Syntax** refers to particular arrangements of "words and letters" eg *David hit the ball* or

$$if\, t > 2\ then\ H = \text{“}Off\text{”}.$$

- A **grammar** is a set of rules which can be used to specify how syntax is created.
- Examples can be seen in automata theory, or programming manuals.
- Theories of syntax and grammars can be developed—ideas are used in compiler construction.

- **Semantics** is the study of "meaning".

- In particular, syntax can be given meaning. The word *run* can mean
  - execution of a computer program,
  - spread of ink on paper, …

- Programming language syntax can be given a semantics. We need this to write programs.

Semantic descriptions are often informal. Consider

while (*expression*) *command* ;

adapted from Kernighan and Ritchie 1978/1988, p 224:

The command is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must have arithmetic or pointer type. The test, including all side effects from the expression, occurs before each execution of the command.

We want to be more precise, more succinct.

High Level view of MC308

For various languages we shall

- define syntax for *programs* $\boxed{P}$ and *types* $\boxed{\sigma}$;
- define *type assignments* $\boxed{P :: \sigma}$;
- define *operational semantics* looking like

$$\boxed{P \rightsquigarrow P'} \qquad \text{or} \qquad \boxed{P \Downarrow V}$$

- define algorithms to check that $P :: \sigma$; and
- compile P to a list of machine instructions $\boxed{P \mapsto [\![P]\!]}$.

Overview: Discrete Mathematics

We briefly review

- Logic
- Sets
- Relations
- Functions

Logic

- If P and Q are propositions, we can form new propositions as follows:
  - P *implies* Q (sometimes written $P \Rightarrow Q$ or $P \rightarrow Q$);
  - $\ldots$ see the notes.
  - *for all* x, P (sometimes written $\forall x.\ P$);
- We shall often prove propositions of the form $\forall x \in X . P(x)$ where $P(x)$ is a proposition depending on x, and X is a given set. Eg

$$\forall n \in \mathbb{N}. \boxed{2 * n + 1 \text{ is odd}}$$

Sets

- We assume a *set* is understood.
- A or B or $\ldots$ often used to denote sets. Write $a \in A$ for **element of**. If a is not an element of A, we write $a \notin A$.
- *Union* $A \cup B$, *intersection* $A \cap B$, should already be known.
- The **cartesian product** of A and B is a set given by

$$A \times B \stackrel{\text{def}}{=} \{\ (a, b)\ \mid\ a \in A \textit{ and } b \in B\ \}.$$

Relations

- A **relation** R between sets A and B is a subset $R \subseteq A \times B$. A **binary relation** R on A is a relation between A and A.

- If $R \subseteq A \times B$ is a relation, it is convenient to write $a\ R\ b$ instead of $(a, b) \in R$.

- R is **reflexive** iff for all $a \in A$ we have $a\ R\ a$;

- R is **transitive** iff for all $a, b, c \in A$, $a\ R\ b$ and $b\ R\ c$ implies $a\ R\ c$;

- For example $\leq \quad \subset \quad \mathbb{N} \times \mathbb{N}$.

Functions

- You should know what a **(total) function** $f : A \to B$ is.
- You should know what a **partial function** $f : A \to B$ is.
- Recall undefinedness and application notation, composition, and domain of definition.

Chapter 2

- Define *abstract syntax trees* – a bit like parse trees.
- Explain *inductive definitions*.
- Explain *Rule Induction*.

Overview: Abstract Syntax

- Outline the ideas of concrete syntax (eg programs as ascii files) and abstract syntax (the parse trees of programs).

Abstract and Concrete Syntax

- The text string

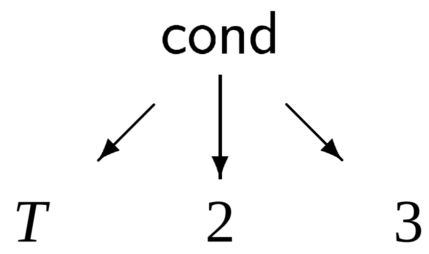
```
if true then 2 else 3
```

is *concrete* syntax.

- A compiler will recognize a *conditional expression* (an “if-then-else”) and three *data*, namely the Boolean and the two numbers.

- The three *data*, together with the knowledge that the string denotes a *conditional*, make up the *semantic content* of the expression.

- We can capture this semantic content as a tree



- which can be denoted by the formal notation

$$\mathsf{cond}(T, 2, 3)$$

- and informally by the *sugared* notation

$$\mathsf{if}\ T\ \mathsf{then}\ 2\ \mathsf{else}\ 3$$

- *Lexers* and *parsers* transform text programs into parse trees, sometimes referred to as *abstract syntax*.

- Here is another example of sugared tree notation

$$\mathsf{if}\ \ \mathsf{elist}(l)\ \ \mathsf{then}\ \ \underline{0}\ \ \mathsf{else}\ \ (\mathsf{hd}(l)+\mathsf{sum}(\mathsf{tl}(l)))$$

- It has the form if B then E_1 else E_2 where, for example, B is $\mathsf{elist}(l)$.

- The abstract syntax tree is

$$\mathsf{cond}(\mathsf{elist}(l)\ ,\ \underline{0}\ ,\ +(\mathsf{hd}(l),\mathsf{sum}(\mathsf{tl}(l))))$$

- Think of the conditional as a *constructor* which acts on three arguments (subprograms) to “construct” a new program.

- In CO3008 we need to give precise definitions of abstract syntax trees. An example:

- Let $C = \{ l_1, l_2, l_3, c_1, c_2 \}$ be a set of *constructors*, which are *labels* for tree *nodes*. We can specify a set of finite trees built from this set by a grammar of the form

$$T ::= l_1 \mid l_2 \mid l_3 \mid c_1(T,T) \mid c_2(T,T,T)$$

- You need to understand the definitions of
  - **node**
  - **leaf**
  - **root**
  - **constructor** (a label for any node)
  - **children** (of non-leaf nodes)
  - **subtree**
- We also talk about
  - **subprogram**, **subexpression**
  - **outermost constructor** (= root label).

Overview: Inductively Defined sets

- Specify **inductively defined sets**; programs, types etc will be defined this way. BNF grammars are a form of inductive definition; abstract syntax trees were defined inductively.
- Define **Rule Induction**; properties of programs will be proved using this. It is *important*.

Example Inductive Definition

Let *Var* be a set of **propositional variables**. Then the set *Prpn* of **propositions** of propositional logic is *inductively* defined by the rules

$$\frac{}{P}\ [P \in Var]\ \ (A) \qquad \frac{\phi \quad \psi}{\phi \wedge \psi}\ (\wedge)$$

$$\frac{\phi \quad \psi}{\phi \vee \psi}\ (\vee) \qquad \frac{\phi \quad \psi}{\phi \to \psi}\ (\to) \qquad \frac{\phi}{\neg\phi}\ (\neg)$$

Each proposition is created by a *deduction* ...

Two More Examples

- A set $\mathcal{R}$ of rules for defining the set $E \subseteq \mathbb{N}$ of even numbers is $\mathcal{R} = \{ R_1, R_2 \}$ where

$$\frac{-}{0}\ (R_1) \qquad\qquad \frac{e}{e+2}\ (R_2)$$

$6 \in E$ *iff* there is a deduction of 6.

- Suppose that Σ is any set, which we think of as an **alphabet**. Each element l of Σ is **letter**. We inductively define the set Σ^* of **words** over the alphabet Σ by

$$\frac{-}{l}\ [l \in \Sigma]\ \ (1) \qquad\qquad \frac{w \quad w'}{ww'}\ (2)$$

Some Notation for Rules

- A **rule** R is a pair (H, c) where H is any finite set.

- Note that H might be $\varnothing$, in which case we say that R is a **base** rule.

$$\frac{-}{c}\ (R)$$

- If H is non-empty (say $H = \{h_1, \ldots, h_k\}$ where $1 \leq k$) we say R is an **inductive** rule.

$$\frac{h_1 \quad h_2 \quad \ldots \quad h_k}{c}\ (R)$$

Inductively Defined Sets

- Given a set of rules, a **deduction** is a finite tree such that
  - each leaf node label c occurs as a base rule $(\emptyset, c) \in \mathcal{R}$
  - for any non-leaf node label c, if H is the set of children of c then $(H, c) \in \mathcal{R}$ is an inductive rule.
- The set I **inductively defined** by $\mathcal{R}$ consists of those elements e which have a deduction with root node e.

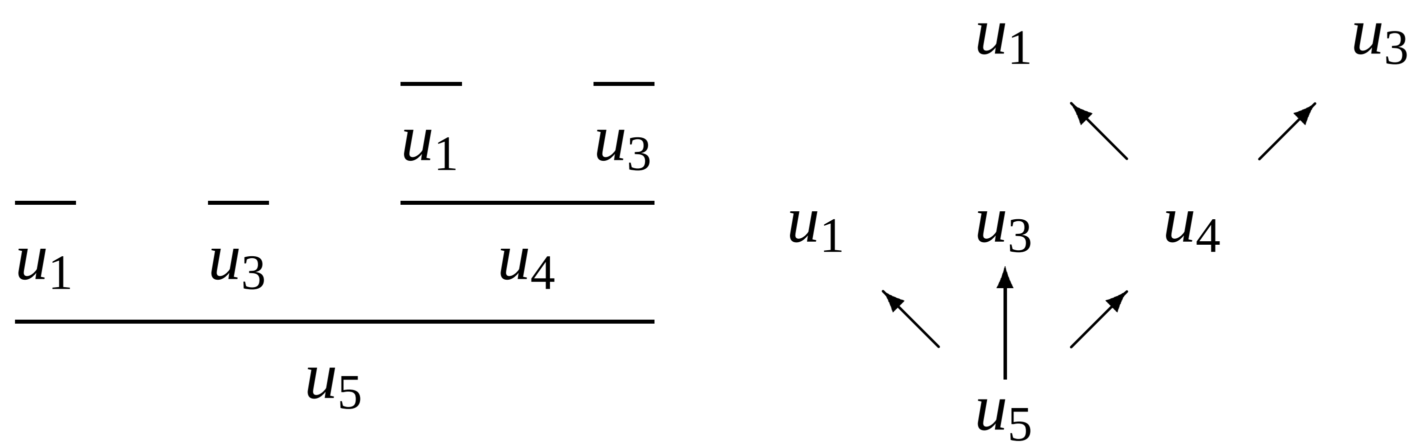
An Abstract Example

Let $\mathcal{R}$ be the set of rules $\{R_1, R_2, R_3, R_4\}$ where

$$R_1 = (\varnothing, u_1), \qquad R_2 = (\varnothing, u_3), \qquad R_3 = (\{u_1, u_3\}, u_4),$$

$$R_4 = (\{u_1, u_3, u_4\}, u_5) \qquad R_5 = (\{u_2, u_3\}, u_6)$$

Then a deduction for u_5 is given by



The inductively defined set is $I = \{u_1, u_3, u_4, u_5\}$

Rule Induction

Let I be inductively defined by a set of rules $\mathcal{R}$. Suppose we wish to show the truth of

$$\forall i \in I. \quad \boxed{\phi(i)}$$

To do this, it is enough to show

– for every base rule $\overline{b} \in \mathcal{R}$ that $\phi(b)$ holds; and

– for every inductive rule $\frac{h_1 \ldots h_k}{c} \in \mathcal{R}$ prove that whenever $h_i \in I$,

$$(\phi(h_1) \textit{ and } \phi(h_2) \textit{ and } \ldots \textit{ and } \phi(h_k)) \quad \textit{implies} \quad \phi(c)$$

We call $\phi(h_j)$ **inductive hypotheses**. We refer to carrying out the – tasks as “verifying **property closure**”.

Example

Consider the set of trees $\mathcal{T}$ defined inductively by

$$\frac{}{n}\ [n \in \mathbb{Z}] \qquad\qquad \frac{T_1 \quad T_2}{+(T_1, T_2)}$$

Let $L(T)$ be the number of leaves in T, and $N(T)$ be the number of $+$-nodes of T. We prove

$$\forall T \in \mathcal{T}. \quad \boxed{L(T) = N(T) + 1}$$

where the functions $L, N\colon \mathcal{T} \to \mathbb{N}$ are defined recursively by

- $L(n) = 1$ and $L(+(T_1, T_2)) = L(T_1) + L(T_2)$
- $N(n) = 0$ and $N(+(T_1, T_2)) = N(T_1) + N(T_2) + 1$

Chapter 3

- Describe the *programs* (syntax) of a simple imperative language called *IMP*.
- Review and motivate *types*.
- Give a *type system* to IMP.
- Describe compile time *type checking* and *type inference*.

Overview: Syntax for IMP Programs

- Describe the basic building blocks for programs.
- Specify the program expressions.
- Comment on some grammatical conventions.

Program Expressions for IMP

Syntax for $\mathbb{IMP}$ built out of elements of the sets

$$
\begin{aligned}
\mathbb{Z} &\stackrel{\text{def}}{=} \{\ldots,-1,0,1,\ldots\} \\
\mathbb{B} &\stackrel{\text{def}}{=} \{T,F\} \\
\mathit{Loc} &\stackrel{\text{def}}{=} \{l_1,l_2,\ldots\} \qquad \text{(** NB **)} \\
\mathit{ICst} &\stackrel{\text{def}}{=} \{\,\underline{n} \mid n \in \mathbb{Z}\,\} \\
\mathit{BCst} &\stackrel{\text{def}}{=} \{\,\underline{b} \mid b \in \mathbb{B}\,\} \\
\mathit{IOpr} &\stackrel{\text{def}}{=} \{+,-,*\} \\
\mathit{BOpr} &\stackrel{\text{def}}{=} \{=,<,\leq,\ldots\}
\end{aligned}
$$

The set of expression **constructors** is specified by

$$Loc \cup ICst \ldots \cup BOpr \cup \{\, \mathsf{skip}, \mathsf{assign}, \mathsf{sequence}, \mathsf{cond}, \mathsf{while} \,\}.$$

The program expressions are given by

$$
\begin{array}{lcll}
P & ::= & \underline{c} & \text{constant} \\
 & \mid & l & \text{memory location} \\
 & \mid & iop(P, P') & \text{integer operator} \\
 & \mid & \mathsf{assign}(l, P') & \text{assignment} \\
 & \mid & \mathsf{cond}(P, P', P'') & \text{while loop} \\
 & \mid & \mathsf{while}(P, P') & \text{sequencing}
\end{array}
$$

- We adopt some abbreviations (known as syntactic **sugar**):
  - We write $P\ iop\ P'$ for $iop(P,P')$;
  - $l := P'$ for $\mathsf{assign}(l,P')$;
  - $P\ ;\ P'$ for $\mathsf{sequence}(P,P')$;
  - $\ldots$
- Bracketing conventions:
  - Arithmetic operators group to the left. Thus $P_1\ op\ P_2\ op\ P_3$ abbreviates $(P_1\ op\ P_2)\ op\ P_3$
  - Sequencing associates to the right.

Overview: Types for IMP Programs

- Explain what a type is.
- Motivate the uses for types.
- Explain some terminology.
- Define IMP type checking (compilation checks).
- Define IMP type inference.

Defining and Motivating Types

Types in a programming language are

- *collections of objects* ("sets"), with
- *collections of operations* acting on these objects.

The type int consists of the collection of integers, together with operations such as $+$, $-$, $\leq$ and so on. The action of $\leq$ might be specified as

$$(\mathsf{int}, \mathsf{int}) \longrightarrow \mathsf{bool}$$

- **Statically typed** languages carry out type checking at compile-time. Needs some *explicit type information*.
- Uses of types
  - Expressions organized to reduce program errors.
  - *Polymorphism* means functions can have many types. This allows code re-use.
  - Types structure data, using ADTs and modules.

- Run time errors
  - **trapped** error – execution halts immediately.
  - An **untrapped** error – execution does not necessarily halt. An example is accessing data past the end of an array, which one can do in *C*!
- A language is **safe** if all syntactically legal programs do not yield certain run-time errors.
- JAVA was claimed to be safe, but in 1997 this was shown not to be the case. *Proof uses MC 308 methods!*

Technical Definitions

- If P can be assigned a type σ we write $P :: \sigma$ and call the statement a **type assignment**.

- **Type safety** is the property that if $P :: \sigma$ then certain kinds of errors can not occur at P's run-time.

- Given P and σ, **type checking** validates $P :: \sigma$.

- Given P, **type inference** is the process of trying to find σ for which $P :: \sigma$—the process can *fail*.

Types for IMP

- The types of the language $\mathbb{IMP}$ are given by the grammar

$$\sigma \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \mathsf{cmd}$$

- A **location environment** $\mathcal{L}$ is a finite set of (location, type) pairs, with type being just int or bool:

$$\mathcal{L} = l_1 :: \mathsf{int}, \ldots, l_n :: \mathsf{int},\ l_{n+1} :: \mathsf{bool}, \ldots, l_m :: \mathsf{bool}$$

- Given $\mathcal{L}$, then any P whose locations all appear in $\mathcal{L}$ can (sometimes) be assigned a type; we write $P :: \sigma$ to indicate this.

$$\frac{}{\underline{n} :: \mathsf{int}} \; [\text{any } n \in \mathbb{Z}] \quad :: \text{INT} \qquad \frac{}{\underline{T} :: \mathsf{bool}} :: \text{TRUE} \qquad \frac{}{\underline{F} :: \mathsf{bool}}$$

$$\frac{}{l :: \mathsf{int}} \; [l :: \mathsf{int} \in \mathcal{L}] \qquad \frac{P_1 :: \mathsf{int} \quad P_2 :: \mathsf{int}}{P_1 \; bop \; P_2 :: \mathsf{bool}} \; [\, bop \in BOpr] \quad :: \text{BOP}$$

$$\frac{}{\mathsf{skip} :: \mathsf{cmd}} \qquad \frac{l :: \sigma \quad P :: \sigma}{l := P :: \mathsf{cmd}}$$

$$\frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd} \quad P_3 :: \mathsf{cmd}}{\mathsf{if}\; P_1 \;\mathsf{then}\; P_2 \;\mathsf{else}\; P_3 :: \mathsf{cmd}} \qquad \frac{P_1 :: \mathsf{bool} \quad P_2 :: \mathsf{cmd}}{\mathsf{while}\; P_1 \;\mathsf{do}\; P_2 :: \mathsf{cmd}}$$

Example: Deduction of a Type Assignment

$$
\dfrac{\dfrac{\dfrac{}{l :: \mathsf{int}} \qquad \dfrac{}{\underline{5} :: \mathsf{int}}}{l \geq \underline{5} :: \mathsf{bool}} \qquad \mathcal{D}2 \qquad \dfrac{\mathcal{D}3 \quad \mathcal{D}4}{l := l - 1 \,;\, l' := l' * l :: \mathsf{cmd}}}{\mathsf{if}\ l \geq 5\ \mathsf{then}\ l' := \underline{1}\ \mathsf{else}\ (l := l + 1 \,;\, l' := l' * l) :: \mathsf{cmd}}
$$

Type Inference

- Given $\mathcal{L}$ and P, there is an algorithm which will infer if P can be assigned a type.
  - If such a type exists we say P is **typable**. The algorithm will *succeed* and will output the type.
  - If not, the algorithm *fails*.
- In a real language, such type inference is often performed by the compiler.
- Given $\mathcal{L}$ and P, we define a function Φ which given P as input will either return a type for P, or will FAIL.

$$\Phi(\underline{T}) \quad = \quad \mathsf{bool}$$

$$\Phi(l) \quad = \quad \begin{cases} \tau & \text{if } l :: \tau \in \mathcal{L}, \textit{ and } \tau = \mathsf{int} \textit{ or } \mathsf{bool} \\ \textit{FAIL} & \text{otherwise} \end{cases}$$

$$\Phi(P_1 \textit{ bop } P_2) \quad = \quad \begin{cases} \mathsf{bool} & \text{if } \Phi(P_1) = \mathsf{int} \text{ and } \Phi(P_2) = \mathsf{int} \\ \textit{FAIL} & \text{otherwise} \end{cases}$$

$$\Phi(l := P) \quad = \quad \begin{cases} \mathsf{cmd} & \text{if } \Phi(l) = \Phi(P) = \tau, \\ & \textit{and } \ \tau = \mathsf{int} \ \textit{or} \ \mathsf{bool} \\ \textit{FAIL} & \text{otherwise} \end{cases}$$

$$\Phi(\mathsf{while}\, P_1 \,\mathsf{do}\, P_2) \quad = \quad \begin{cases} \mathsf{cmd} & \text{if } \Phi(P_1) = \mathsf{bool} \text{ and } \Phi(P_2) = \mathsf{cmd} \\ \textit{FAIL} & \text{otherwise} \end{cases}$$

Chapter 4

- Explain how IMP programmes execute—an *operational semantics*.
- Show that the type of a program does not change on execution.
- Show that a program always gives the same answer when run—IMP is *deterministic*.
- Typed programs don't yield certain errors.

Overview: Transition Semantics

- Motivate and define *transition semantics*—a method for stating precisely how a program executes.
- Give some examples.

States

- A **state** s is a partial function $Loc \to \mathbb{Z} \cup \mathbb{B}$.
- For example $s = \langle l_1 \mapsto 4, l_2 \mapsto T, l_3 \mapsto 21 \rangle$
- There is a state denoted by $s\{l \mapsto c\} : Loc \to \mathbb{Z} \cup \mathbb{B}$ which is the partial function

$$(s\{l \mapsto c\})(l') \stackrel{\text{def}}{=} \begin{cases} c & \text{if } l' = l \\ s(l') & \text{otherwise} \end{cases}$$

- We say that state s is **updated** at l by c.

Transition Semantics

Consider the following *transition*, which models one *step* in a program *execution*

$$
\begin{aligned}
(l := 2+5\,,\, \langle l' \mapsto 8 \rangle) &\rightsquigarrow (l := 7\,,\, \langle l' \mapsto 8 \rangle) \\
&\rightsquigarrow (\mathsf{skip}\,,\, \langle l' \mapsto 8, l \mapsto 7 \rangle)
\end{aligned}
$$

- The elements of $\mathit{Exp} \times \mathit{States}$ will be known as **configurations**.
- We shall inductively define a binary relation $\rightsquigarrow$. We call it **transition** relation, and any instance of a relationship in $\rightsquigarrow$ is called a **transition step**.

$$\frac{}{(l, s) \rightsquigarrow (\underline{s(l)}, s)} \text{ [provided that } s(l) \text{ is defined]} \rightsquigarrow \text{LOC}$$

$$\frac{(P_1, s) \rightsquigarrow (P_2, s)}{(P_1 \; op \; P, s) \rightsquigarrow (P_2 \; op \; P, s)} \rightsquigarrow \text{OP}_1$$

$$\frac{(P_1, s) \rightsquigarrow (P_2, s)}{(\underline{n} \; op \; P_1, s) \rightsquigarrow (\underline{n} \; op \; P_2, s)} \rightsquigarrow \text{OP}_2 \qquad \frac{}{(\underline{n_1} \; op \; \underline{n_2}, s) \rightsquigarrow (\underline{n_1 \; op \; n_2}, s)} \rightsquigarrow \text{OP}_3$$

$$\frac{(P_1, s) \rightsquigarrow (P_2, s)}{(l := P_1, s) \rightsquigarrow (l := P_2, s)} \rightsquigarrow \text{ASS}_1 \qquad \frac{}{(l := \underline{c}, s) \rightsquigarrow (\mathsf{skip}, s\{l \mapsto c\})} \rightsquigarrow \text{ASS}_2$$

$$\frac{(P_1\,,s_1) \rightsquigarrow (P_2\,,s_2)}{(P_1\,;P\,,s_1) \rightsquigarrow (P_2\,;P\,,s_2)} \rightsquigarrow\mathrm{SEQ}_1 \qquad \frac{}{(\mathsf{skip}\,;P\,,s) \rightsquigarrow (P\,,s)} \rightsquigarrow\mathrm{SEQ}_2$$

$$\frac{(P\,,s) \rightsquigarrow (P'\,,s)}{(\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s) \rightsquigarrow (\mathsf{if}\ P'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s)} \rightsquigarrow\mathrm{COND}_1$$

$$\frac{}{(\mathsf{if}\ \underline{T}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s) \rightsquigarrow (P_1\,,s)} \rightsquigarrow\mathrm{COND}_2$$

$$\frac{}{(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s) \rightsquigarrow (\mathsf{if}\ P_1\ \mathsf{then}\ (P_2\,;\mathsf{while}\ P_1\ \mathsf{do}\ P_2)\ \mathsf{else}\ \mathsf{skip}\,,s)} \rightsquigarrow\mathrm{LOOP}$$

Examples of Transitions

A deduction (for any P):

$$
\dfrac{\dfrac{\dfrac{}{(l' := \underline{2}\,,\, s) \rightsquigarrow (\mathsf{skip}\,,\, s\{l' \mapsto 2\})}\ \rightsquigarrow\text{ASS}_2}{(l' := \underline{2}\ ;\ l := l - \underline{1}\,,\, s) \rightsquigarrow (\mathsf{skip}\ ;\ l := l - \underline{1}\,,\, s\{l' \mapsto 2\})}\ \rightsquigarrow\text{SEQ}_1}{((l' := \underline{2}\ ;\ l := l - \underline{1})\ ;\ P\,,\, s)\ \boxed{\rightsquigarrow}\ ((\mathsf{skip}\ ;\ l := l - \underline{1})\ ;\ P\,,\, s\{l' \mapsto 2\})}\ \rightsquigarrow\text{SEQ}_1
$$

Q is while $l > \underline{0}$ do Q' where Q' is $l' := l' + \underline{2}\ ;\ l := l - \underline{1}$.

$$
\begin{array}{lll}
(Q\,,\langle l \mapsto 1, l' \mapsto 0\rangle) & \rightsquigarrow & (\text{if } l > \underline{0} \text{ then } Q'\ ;\ Q \text{ else skip}\,,\langle l \mapsto 1, l' \mapsto 0\rangle) \\
& \rightsquigarrow & (\text{if } \underline{1} > \underline{0} \text{ then } Q'\ ;\ Q \text{ else skip}\,,\langle l \mapsto 1, l' \mapsto 0\rangle) \\
& \rightsquigarrow & (\text{if } \underline{T} \text{ then } Q'\ ;\ Q \text{ else skip}\,,\langle l \mapsto 1, l' \mapsto 0\rangle) \\
& & \ldots \\
& \rightsquigarrow & ((l' := \underline{2}\ ;\ l := l - \underline{1})\ ;\ Q\,,\langle l \mapsto 1, l' \mapsto 0\rangle) \\
& \boxed{\rightsquigarrow} & ((\text{skip}\ ;\ l := l - \underline{1})\ ;\ Q\,,\langle l \mapsto 1, l' \mapsto 2\rangle) \\
& \rightsquigarrow & (l := l - \underline{1}\ ;\ Q\,,\langle l \mapsto 1, l' \mapsto 2\rangle)
\end{array}
$$

Overview: Properties of the Semantics

- Program types do not change on execution.
- IMP is deterministic—the final result of a program run is unique; and in fact the “stages” of the run are unique.

Type Preservation

- Given $\mathcal{L}$, s is **sensible** for $\mathcal{L}$ if for all $l :: \sigma$ in $\mathcal{L}$
  - $s(l)$ is defined (*all locations initialized)*, and
  - $\underline{s(l)} :: \sigma$ (*the type of data stored in a location matches the type of the location)*.
- Take $\mathcal{L}$ and sensible s_1. Then $\rightsquigarrow$ satisfies
  - Let $P_1 :: \sigma$. Then for any $(P_1, s_1) \rightsquigarrow (P_2, s_2)$ we have $P_2 :: \sigma$.
  - Further, if σ is either int or bool, then $s_1 = s_2$.

Proving Type Preservation

$$\forall\,(P_1\,,s_1) \rightsquigarrow (P_2\,,s_2) \qquad \boxed{\forall\,\sigma. \quad (P_1 :: \sigma \textit{ implies } P_2 :: \sigma)}$$

We have to check property closure for each of the rules defining $\rightsquigarrow$. We look at a couple of examples.

(*Property Closure for* $\rightsquigarrow_{LOC}$) We have to show that $l :: \sigma$ *implies* $\underline{s(l)} :: \sigma$ for any σ. This is immediate as s is sensible.

(*Property Closure for* $\rightsquigarrow_{OP_2}$) The induction hypothesis is

$$\forall\, \sigma. \quad (P_1 :: \sigma \textit{ implies } P_2 :: \sigma) \qquad \textbf{IH}$$

$$\frac{(P_1\,, s) \rightsquigarrow (P_2\,, s)}{(\underline{n}\ op\ P_1\,, s) \rightsquigarrow (\underline{n}\ op\ P_2\,, s)} \rightsquigarrow\!\mathrm{OP}_2$$

We have to prove

$$\forall\, \sigma. \quad (\underline{n}\ op\ P_1 :: \sigma \textit{ implies } \underline{n}\ op\ P_2 :: \sigma) \qquad \textbf{C}$$

IMP is Deterministic

The operational semantics of $\mathbb{IMP}$ is **deterministic**:

If

$$(P\,,\,s) \rightsquigarrow (P'\,,\,s') \qquad \textit{and} \qquad (P\,,\,s) \rightsquigarrow (P''\,,\,s'')$$

then

$$P' = P'' \quad \text{and} \quad s' = s''$$

Proof of Determinism

We can prove this result by Rule Induction. We show

$$\forall (P\,,\, s) \leadsto (P'\,,\, s')$$

$$\boxed{\forall (X\,,\, x), \quad (P\,,\, s) \leadsto (X\,,\, x) \;\; \textit{implies} \;\; (X = P' \;\; \textit{and} \;\; x = s')}$$

We consider property closure for

$$\frac{(P_1\,,\,s) \rightsquigarrow (P_2\,,\,s)}{(l := P_1\,,\,s) \rightsquigarrow (l := P_2\,,\,s)} \rightsquigarrow_{\text{ASS}_1}$$

The inductive hypothesis IH is

$$\forall (Y\,,\,y), \quad (P_1\,,\,s) \rightsquigarrow (Y\,,\,y) \textit{ implies } (Y = P_2 \textit{ and } y = s)$$

We need to prove the conclusion C

$$\forall (Z\,,\,z), \quad (l := P_1\,,\,s) \rightsquigarrow (Z\,,\,z) \textit{ implies } (Z = (l := P_2) \textit{ and } z = s)$$

Overview: IMP is Type Safe

- We describe some special programs;
- we describe some special kinds of transitions, and
- use the ideas to show IMP is type safe.

Different Kinds of Transitions

- We define $V ::= \underline{c} \mid \mathsf{skip}$.

- (V, s) configurations are called **terminal**. They indicate *"proper" termination* of program runs.

- Any configuration (P, s) is **stuck** if P is *non-terminal* and there is no (P', s') for which $(P, s) \rightsquigarrow (P', s')$.

- **WARNING**: Note that any terminal configuration **has no transition**.

- Given any configuration (P, s) there is a *unique* sequence of transitions

$$(P, s) = (P_1, s_1) \leadsto (P_2, s_2) \leadsto \ldots$$

- An **infinite transition sequence** takes the form

$$(P, s) = (P_1, s_1) \leadsto (P_2, s_2) \leadsto \ldots \leadsto (P_i, s_i) \leadsto \ldots$$

where no configuration (P_i, s_i) is terminal or stuck.

- A **finite transition sequence** for a configuration (P, s) takes the form

$$(P, s) = (P_1, s_1) \rightsquigarrow (P_2, s_2) \rightsquigarrow \ldots \rightsquigarrow (P_m, s_m) \qquad (m \geq 1)$$

- If (P_m, s_m) is either stuck or terminal we call the transition sequence **complete**.
- Make up lots of examples of these ideas!!

Some Results about IMP Type Safety

- Let s be sensible for $\mathcal{L}$. Then if $P :: \sigma$ is any type assignment, (P, s) is not stuck.

- If also $(P, s) \rightsquigarrow (P', s')$, then s' is also sensible.

- If $(P, s) \rightsquigarrow^* (P', s')$, then (P', s') cannot be stuck (but might be terminal). Thus $\mathbb{IMP}$ is *type safe*.

  This follows from the two results above—why?

We prove $\forall P :: \sigma \ \boxed{(P, s) \text{ is not stuck}}$ by Rule Induction on type assignments.

(*Property Closure for* $::$ IOP)

The inductive hypotheses are that neither (P_1, s) or (P_2, s) are stuck, where $P_1 :: \mathsf{int}$ and $P_2 :: \mathsf{int}$.

We have to prove that $(P_1 \ iop \ P_2, s)$ is not stuck, where $P_1 \ iop \ P_2 :: \mathsf{int}$.

Let's work this on the board ...

We prove, for a given $\mathcal{L}$,

$$\forall\,(P\,,\,s) \rightsquigarrow (P'\,,\,s')\ \boxed{\forall\sigma.\ \ (P::\sigma \text{ and } s \text{ sensible})\ \textit{implies}\ s' \text{ sensible}}$$

by rule induction for $\rightsquigarrow$.

We check property closure for

$$\frac{}{(l := \underline{c}\,,\,s) \rightsquigarrow (\mathsf{skip}\,,\,s\{l \mapsto c\})} \rightsquigarrow_{\mathrm{ASS}_2}$$

Suppose s is sensible, and $l := \underline{c} :: \sigma$. We need to verify that $s\{l \mapsto c\}$ is sensible, that is

- All locations in $\mathcal{L}$ are in the domain of definition of $s\{l \mapsto c\}$.
- $\forall l' :: \tau$ in $\mathcal{L}$ we have $\underline{s\{l \mapsto c\}(l')} :: \tau$.

Overview: Evaluation Relations

- We describe a semantics which tells us "immediately" the final result of a program run.
- We show how this connects with transitions.

An Evaluation Relation

Consider the following *evaluation relationship*

$$(l' := \underline{T}\,;\, l := \underline{4} + \underline{1},\, \langle\rangle) \Downarrow (\mathsf{skip},\, \langle l' \mapsto T, l \mapsto 5\rangle)$$

The idea is

$$\textit{Starting program} \Downarrow \textit{final result}$$

We describe an operational semantics which has assertions which look like

$$(P, s) \Downarrow (\underline{n}, s) \qquad \text{and} \qquad (P, s_1) \Downarrow (\mathsf{skip}, s_2)$$

$$\frac{}{(l\,,s) \Downarrow (\underline{s(l)}\,,s)}\ [\text{ provided } l \in \text{ domain of } s] \Downarrow\text{LOC}$$

$$\frac{(P_1\,,s) \Downarrow (\underline{n_1}\,,s) \quad (P_2\,,s) \Downarrow (\underline{n_2}\,,s)}{(P_1\ bop\ P_2\,,s) \Downarrow (\underline{n_1\ bop\ n_2}\,,s)} \Downarrow\text{OP}_2$$

$$\frac{(P\,,s) \Downarrow (\underline{n}\,,s)}{(l := P\,,s) \Downarrow (\mathsf{skip}\,,s\{l \mapsto n\})} \Downarrow\text{ASS}_1 \qquad \frac{(P\,,s) \Downarrow (\underline{b}\,,s)}{(l := P\,,s) \Downarrow (\mathsf{skip}\,,s\{l \mapsto b\})} \Downarrow\text{ASS}_2$$

$$\frac{(P_1\,,s_1) \Downarrow (\mathsf{skip}\,,s_2) \quad (P_2\,,s_2) \Downarrow (\mathsf{skip}\,,s_3)}{(P_1\,;P_2\,,s_1) \Downarrow (\mathsf{skip}\,,s_3)} \Downarrow\text{SEQ}$$

$$\frac{(P\,,s_1)\Downarrow(\underline{F}\,,s_1)\quad (P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}{(\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)}\Downarrow\mathrm{COND}_2$$

$$\frac{(P_1\,,s_1)\Downarrow(\underline{T}\,,s_1)\quad (P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_2)\quad (\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s_2)\Downarrow(\mathsf{skip}\,,s_3)}{(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s_1)\Downarrow(\mathsf{skip}\,,s_3)}$$

$$\frac{(P_1\,,s)\Downarrow(\underline{F}\,,s)}{(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,s)\Downarrow(\mathsf{skip}\,,s)}\Downarrow\mathrm{LOOP}_2$$

Example Evaluations

We derive deductions for

$$((\underline{3} + \underline{2}) * \underline{6}\,,\, s) \Downarrow (\underline{30}\,,\, s)$$

and

$$(\mathsf{while}\ l = \underline{1}\ \mathsf{do}\ l := l - \underline{1},\ \langle l \mapsto 1 \rangle) \Downarrow (\mathsf{skip}\,,\ \langle l \mapsto 0 \rangle)$$

A Mutual Correctness Proof

For any configuration $(P\,, s)$ and terminal configuration $(V\,, s')$,

$$(P\,, s) \leadsto^* (V\,, s') \textit{ iff } (P\,, s) \Downarrow (V\,, s')$$

where $\leadsto^*$ denotes reflexive, transitive closure of $\leadsto$.

We break the proof into three parts:

– Prove $(P, s) \Downarrow (V, s')$ *implies* $(P, s) \rightsquigarrow^* (V, s')$ by Rule Induction.

– Prove by Rule Induction for $\rightsquigarrow$ that

$$(P, s) \rightsquigarrow (P', s') \Downarrow (V, s'') \quad \textit{implies} \quad (P, s) \Downarrow (V, s'')$$

– Use previous results to deduce

$$(P, s) \rightsquigarrow^* (V, s') \; \textit{implies} \; (P, s) \Downarrow (V, s')$$

We shall prove by Rule Induction that

$$\forall (P\,,\,s) \Downarrow (V\,,\,s') \quad \boxed{(P\,,\,s) \rightsquigarrow^{*} (V\,,\,s')}$$

$$(P_1\,,\,s_1) \rightsquigarrow^{*} (\underline{T}\,,\,s_1) \qquad (H1)$$

$$(P_2\,,\,s_1) \rightsquigarrow^{*} (\mathsf{skip}\,,\,s_2) \qquad (H2)$$

$$(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,\,s_2) \rightsquigarrow^{*} (\mathsf{skip}\,,\,s_3) \qquad (H3)$$

We need to prove that

$$(\mathsf{while}\ P_1\ \mathsf{do}\ P_2\,,\,s_1) \rightsquigarrow^{*} (\mathsf{skip}\,,\,s_3) \qquad (C)$$

Let us write Q for $\mathsf{while}\ P_1\ \mathsf{do}\ P_2$. Then

$$
\begin{array}{llll}
(Q, s_1) & \rightsquigarrow & (\mathsf{if}\ P_1\ \mathsf{then}\ P_2\,;Q\ \mathsf{else}\ \mathsf{skip}, s_1) & (\rightsquigarrow \text{LOOP}) \\
& \rightsquigarrow^* & (\mathsf{if}\ \underline{T}\ \mathsf{then}\ P_2\,;Q\ \mathsf{else}\ \mathsf{skip}, s_1) & (H1)\ \&\ (\rightsquigarrow \text{COND}_1) \\
& \rightsquigarrow & (P_2\,;Q, s_1) & (\rightsquigarrow \text{COND}_2) \\
& \rightsquigarrow^* & (\mathsf{skip}\,;Q, s_2) & (H2)\ \&\ (\rightsquigarrow \text{SEQ}_1) \\
& \rightsquigarrow & (Q, s_2) & (\rightsquigarrow \text{SEQ}_2) \\
& \rightsquigarrow^* & (\mathsf{skip}, s_3) & (H3)
\end{array}
$$

which proves (C).

We shall prove by Rule Induction for $\rightsquigarrow$ that

$$\forall (P, s) \rightsquigarrow (P', s'). \quad \boxed{\forall (V, s''). (P', s') \Downarrow (V, s'') \textit{ implies } (P, s) \Downarrow (V, s'')}$$

Let us just consider property closure for the rule ($\rightsquigarrow$ LOOP). Pick any (V, s'') and suppose that

$$(\mathsf{if}\ P_1\ \mathsf{then}\ (P_2\ ;\ Q)\ \mathsf{else\ skip}, s) \Downarrow (V, s'') \qquad (1)$$

We need to show that

$$(Q, s) \Downarrow (V, s'') \qquad (2)$$

But (1) can hold only if it has been deduced either from ($\Downarrow$ COND$_1$) or ($\Downarrow$ COND$_2$). In either case V must be skip.

Chapter 5

- Describe the CSS machine, which executes compiled IMP programs.
- Show how to compile IMP programs to CSS instruction sequences.
- Give some example executions.

Motivating the CSS Machine

An operational semantics gives a useful model of $\mathbb{IMP}$—we seek a more direct, "computational" method for evaluating configurations.

If $P \Downarrow^e V$, how do we effectively compute V from P? The transition relation is not quite right.

It is easy for humans to see that

$$(\underline{3} + \underline{2}) \leq \underline{6} \qquad \rightsquigarrow \qquad \underline{5} \leq \underline{6}$$

but establishing this involves a deduction tree …

We seek a way of taking a program P, and mechanically producing the value V:

$$P \equiv P_0 \mapsto P_1 \mapsto P_2 \mapsto \ldots \mapsto P_n \equiv V$$

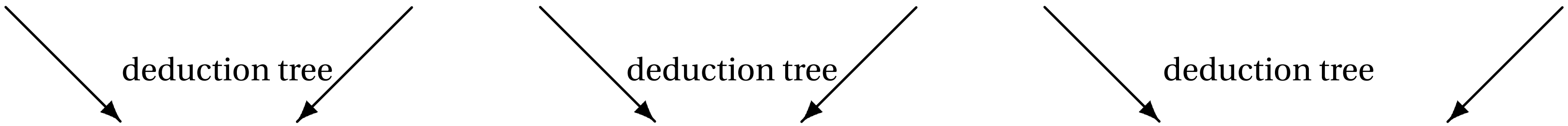
"Mechanically produce" can be made precise using a relation $P \mapsto P'$ defined by a set of rules in which there are *no hypotheses*. Such rules are called **re-writes**:

$$\frac{}{\underline{n} + \underline{m} \rightsquigarrow \underline{m + n}}$$

Establishing $P \mapsto P'$ will not require the construction of a deduction tree:

$$P_0 \mapsto P_1 \mapsto P_2 \mapsto P_3 \mapsto P_4 \ldots \mapsto V$$

Rewrite Rules (Abstract Machine)



$$P_0 \quad \rightsquigarrow \quad P_1 \quad \rightsquigarrow \quad P_2 \quad \rightsquigarrow \quad \ldots \rightsquigarrow V$$

Transition Semantics

An Example

Let $s(l) = 6$. Execute $\underline{10} - l$ on the CSS machine.

First, compile the program.

$$[\![\underline{10} - l]\!] \quad = \quad \mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-)$$

Then

$$\begin{array}{l}
\boxed{\mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-) \;\|\; \bullet \;\|\; s} \\
\longmapsto \boxed{\mathsf{PUSH}(\underline{10}) : \mathsf{OP}(-) \;\|\; \underline{6} \;\|\; s} \\
\longmapsto \boxed{\mathsf{OP}(-) \;\|\; \underline{10} : \underline{6} \;\|\; s} \\
\longmapsto \boxed{\bullet \;\|\; \underline{4} \;\|\; s}
\end{array}$$

Defining the CSS Machine

- A CSS **code** C is a "list":

$$ins \quad ::= \quad \mathsf{PUSH}(\underline{c}) \mid \mathsf{FETCH}(l) \mid \mathsf{OP}(op) \mid \mathsf{SKIP}$$

$$\mid \mathsf{STO}(l) \mid \mathsf{BR}(C,C) \mid \mathsf{LOOP}(C,C)$$

$$C \quad ::= \quad \mathbf{.} \mid ins \mid ins : C$$

The objects *ins* are CSS **instructions**.

- A **stack** σ is produced by the grammar

$$\sigma ::= \mathbf{.} \mid \underline{c} \mid \underline{c} : \sigma$$

- A CSS **configuration** is a triple (C, σ, s).

- A CSS **transition** takes the form

$$(C_1, \sigma_1, s_1) \longmapsto (C_2, \sigma_2, s_2)$$

- Defined inductively by a set of rules, each rule having the form

$$\frac{}{(C_1, \sigma_1, s_1) \longmapsto (C_2, \sigma_2, s_2)} R$$

- We call a binary relation (such as $\longmapsto$) which is inductively defined by rules with no hypotheses a **re-write** relation.

$$\mathsf{PUSH}(\underline{c}) : C \;\|\; \sigma \;\|\; s \quad \longmapsto \quad C \;\|\; \underline{c} : \sigma \;\|\; s$$

$$\mathsf{FETCH}(l) : C \;\|\; \sigma \;\|\; s \quad \longmapsto \quad C \;\|\; \underline{s(l)} : \sigma \;\|\; s$$

$$\mathsf{OP}(\, op \,) : C \;\|\; \underline{n_1} : \underline{n_2} : \sigma \;\|\; s \quad \longmapsto \quad C \;\|\; \underline{n_1 \; op \; n_2} : \sigma \;\|\; s$$

$$\mathsf{SKIP} : C \;\|\; \sigma \;\|\; s \quad \longmapsto \quad C \;\|\; \sigma \;\|\; s$$

$$\mathsf{STO}(l) : C \;\|\; \underline{c} : \sigma \;\|\; s \quad \longmapsto \quad C \;\|\; \sigma \;\|\; s\{l \mapsto c\}$$

$$\mathsf{BR}(C_1, C_2) : C \;\|\; \underline{F} : \sigma \;\|\; s \quad \longmapsto \quad C_2 : C \;\|\; \sigma \;\|\; s$$

$$\mathsf{LOOP}(C_1, C_2) : C \;\|\; \sigma \;\|\; s \quad \longmapsto$$

$$C_1 : \mathsf{BR}(C_2 : \mathsf{LOOP}(C_1, C_2), \mathsf{SKIP}) : C \;\|\; \sigma \;\|\; s$$

$$
\begin{aligned}
[\![\underline{c}]\!] &\stackrel{\text{def}}{=} \mathsf{PUSH}(\underline{c}) \\
[\![l]\!] &\stackrel{\text{def}}{=} \mathsf{FETCH}(l) \\
[\![P_1\ op\ P_2]\!] &\stackrel{\text{def}}{=} [\![P_2]\!] : [\![P_1]\!] : \mathsf{OP}(op) \\
[\![l := P]\!] &\stackrel{\text{def}}{=} [\![P]\!] : \mathsf{STO}(l) \\
[\![\mathsf{skip}]\!] &\stackrel{\text{def}}{=} \mathsf{SKIP} \\
[\![P_1\ ;\ P_2]\!] &\stackrel{\text{def}}{=} [\![P_1]\!] : [\![P_2]\!] \\
[\![\mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2]\!] &\stackrel{\text{def}}{=} [\![P]\!] : \mathsf{BR}([\![P_1]\!], [\![P_2]\!]) \\
[\![\mathsf{while}\ P_1\ \mathsf{do}\ P_2]\!] &\stackrel{\text{def}}{=} \mathsf{LOOP}([\![P_1]\!], [\![P_2]\!])
\end{aligned}
$$

An Example Execution

Execute $l := \underline{2}\ ;\ l' := \underline{5} * l$ on the CSS machine. First, compile the program.

$$[\![l := \underline{2}\ ;\ l' := \underline{5} * l]\!] =$$

$$\mathsf{PUSH}(\underline{2}) : \mathsf{STO}(l) : \mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{5}) : \mathsf{OP}(*) : \mathsf{STO}(l')$$

Then

$$\mathsf{PUSH}(\underline{2}) : \mathsf{STO}(l) : \mathsf{FETCH}(l) : \mathsf{PUSH}(\underline{5}) : \mathsf{OP}(*) : \mathsf{STO}(l') \;\|\; \centerdot \;\|\; \centerdot$$

$$\longmapsto^*$$

$$\centerdot \;\|\; \centerdot \;\|\; \langle l \mapsto 2, l' \mapsto 10 \rangle$$

Chapter 6

- Motivate a language in which we can write higher order functions.
- Describe its types.
- Describe its expression syntax.
- Outline a type assignment system.
- Explain how to write simple programs.

Overview: Motivating and Defining FUN

- Give a broad outline of FUN.
- Define its syntax and type system.
- Explain some technical conventions and definitions.

Examples of FUN Declarations

```
cst :: Int
cst = 76

f :: Int -> Int
f x = x

g :: Int -> Int -> Int
g x y = x+y

h :: Int -> Int -> Int -> Int
h x y z = x+y+z
```

```
empty_list :: [Int]
empty_list = nil

l1 :: [Int]
l1 = 5:(6:(8:(4:(nil))))

l2 :: [Int]
l2 = 5:6:8:4:nil

h :: Int
h  = hd (5:6:8:4:nil)
```

```
p      :: (Int,Int)
fst    :: (Int,Int) -> Int
length :: [Bool] -> Int
map    :: (Int -> Bool) -> [Int] -> [Bool]

p          = (3,4)
fst (x,y) = x
length l  = if elist(l) then 0 else (1 + length t)
map f l   = if elist(l) then nil else (f h) : (map f t)
```

FUN Types

- The types of $\mathbb{FUN}^e$ are

$$\sigma \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \sigma \to \sigma \mid (\sigma, \sigma) \mid [\sigma]$$

We shall write *Type* for the set of types.

- We shall write

$$\sigma_1 \to \sigma_2 \to \sigma_3 \to \ \ldots \to \sigma_n \to \sigma$$

for

$$\sigma_1 \to (\sigma_2 \to (\sigma_3 \to (\ldots \to (\sigma_n \to \sigma)\ldots))).$$

Thus for example $\sigma_1 \to \sigma_2 \to \sigma_3$ means $\sigma_1 \to (\sigma_2 \to \sigma_3)$.

FUN Expressions

E	$::=$	x	variables
	$\mid$	K	constant identifier
	$\mid$	F	function identifier
	$\mid$	$\mathsf{fst}(E)$	first projection
	$\mid$	$E_1\, E_2$	function application
	$\mid$	$\mathsf{tl}(E)$	tail of list
	$\mid$	$E_1 : E_2$	cons for lists
	$\mid$	$\mathsf{elist}(E)$	Boolean test for empty list

Bracketing conventions apply . . .

Substitution (for next chapter)

- The variable x *occurs* in the expression $x\ op\ \underline{3}\ op\ x$.
- If E and $E_1,\ldots,E_n$ are expressions, then $E[E_1,\ldots,E_n/x_1,\ldots,x_n]$ denotes the expression E with E_i *simultaneously* replacing x_i for each $1 \leq i \leq n$.
- Eg

$$(u+x+y+\underline{6})[\underline{2},x,z/u,y,x] = \underline{2}+z+x+\underline{6}$$

Overview: FUN Type System

- Show how to declare the types of variables and identifiers; an *identifier* is (the name of) a constant or function.
- Define a type assignment system.
- Give some examples.
- Verify that FUN is monomorphic—each program has a unique type.

Contexts

- When we write a FUN program, we shall declare the types of variables, for example

$$x :: \mathsf{int}, y :: \mathsf{bool}, z :: \mathsf{bool}$$

- A **context** takes the form

$$\Gamma = x_1 :: \sigma_1, \ldots, x_n :: \sigma_n.$$

- Thus a context specifies type declarations for variables. The variables must be *distinct*.

Environments

- When we write a FUN program, we want to declare the types of constants and functions.

- A simple example of an *identifier environment* is $\mathsf{maxint} :: \mathsf{int}, \mathsf{negate} :: \mathsf{bool} \to \mathsf{bool}$

- and another is $\mathsf{plus} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}$

- and another is

$$\mathsf{K} :: \mathsf{bool}, \ \ \mathsf{map} :: (\mathsf{int} \to \mathsf{int}) \to [\mathsf{int}] \to [\mathsf{int}], \ \ \mathsf{suc} :: \mathsf{int} \to \mathsf{int}$$

- An **identifier type** looks like $\sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_k \to \sigma$ where k is a natural number and σ is **NOT a function type**.
- If $k = 0$ then the identifier is called a constant.
- If $k > 0$ then the identifier is called a function.
- An **identifier environment** looks like

$$I = \mathsf{I}_1 :: \iota_1, \ldots, \mathsf{I}_m :: \iota_m.$$

Example Type Assignments

- With the previous identifier environment

$$x :: \mathsf{int}, y :: \mathsf{int}, z :: \mathsf{int} \vdash \mathsf{map}\,\mathsf{suc}\,(x : y : z : \mathsf{nil}_{\mathsf{int}}) :: [\mathsf{int}]$$

- We have

$$\varnothing \vdash \mathsf{if}\ \underline{T}\ \mathsf{then}\ \mathsf{fst}((\underline{2} : \mathsf{nil}_{\mathsf{int}}, \mathsf{nil}_{\mathsf{int}}))\ \mathsf{else}\ (\underline{2} : \underline{6} : \mathsf{nil}_{\mathsf{int}}) :: [\mathsf{int}]$$

Inductively Defining Type Assignments

Start with an identifier environment and a context. Then

$$\frac{}{\Gamma \vdash x :: \sigma}\ (\text{ where } x :: \sigma \in \Gamma) \quad :: \text{VAR} \qquad \frac{}{\Gamma \vdash \underline{n} :: \mathsf{int}} :: \text{INT}$$

$$\frac{\Gamma \vdash E_1 :: \mathsf{int} \quad \Gamma \vdash E_2 :: \mathsf{int}}{\Gamma \vdash E_1\ iop\ E_2 :: \mathsf{int}} :: \text{OP}_1$$

$$\frac{\Gamma \vdash E_1 :: \sigma_2 \to \sigma_1 \quad \Gamma \vdash E_2 :: \sigma_2}{\Gamma \vdash E_1\, E_2 :: \sigma_1} :: \text{AP}$$

$$\frac{\Gamma \vdash E :: (\sigma_1, \sigma_2)}{\Gamma \vdash \mathsf{fst}(E) :: \sigma_1} :: \text{FST} \qquad \frac{}{\Gamma \vdash \mathsf{l} :: \iota}(\text{ where } \mathsf{l} :: \iota \in I) \quad :: \text{IDR}$$

$$\frac{}{\Gamma \vdash \mathsf{nil}_\sigma :: [\sigma]} :: \text{NIL} \qquad \frac{\Gamma \vdash E_1 :: \sigma \quad \Gamma \vdash E_2 :: [\sigma]}{\Gamma \vdash E_1 : E_2 :: [\sigma]} :: \text{CONS}$$

FUN is Monomorphic

Given I, Γ and E, if there is a type σ for which $\Gamma \vdash E :: \sigma$, then such a type is unique.

We verify

$$\forall\, (\Gamma \vdash E :: \sigma_1). \quad \boxed{\forall \sigma_2. \quad (\Gamma \vdash E :: \sigma_2 \textit{ implies } \sigma_1 = \sigma_2).}$$

using Rule Induction. We check property closure for the rule HD:

The inductive hypothesis is

$$\forall \sigma_2, \quad (\Gamma \vdash E :: \sigma_2 \ \textit{implies} \ [\sigma] = \sigma_2)$$

where $\Gamma \vdash E :: [\sigma]$.

$$\frac{\Gamma \vdash E :: [\sigma]}{\Gamma \vdash \mathsf{hd}(E) :: \sigma} :: \text{HD}$$

We wish to prove that

$$\forall \sigma_2, \quad (\Gamma \vdash \mathsf{hd}(E) :: \sigma_2 \ \textit{implies} \ \sigma = \sigma_2) \quad (\dagger)$$

where $\Gamma \vdash \mathsf{hd}(E) :: \sigma$.

Overview: Function Declarations and Programs

- Show how to code up functions.
- Define what makes up a FUN program.
- Give some examples.

Introducing Function Declarations

- To declare plus can write $\mathsf{plus}\,x = \mathsf{fst}(x) + \mathsf{snd}(x)$.
- To declare fac

$$\mathsf{fac}\,x = \mathsf{if}\ x == \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * \mathsf{fac}(x - \underline{1})$$

- And to declare that b denotes $\underline{T}$ we write $\mathsf{b} = \underline{T}$.
- In $\mathbb{FUN}^e$, can specify

$$\mathsf{K} = E \qquad \mathsf{F}x = E' \qquad \mathsf{G}xy = E'' \ldots$$

An Example Declaration

Let $I = \mathsf{I}_1 :: [\mathsf{int}] \to \mathsf{int} \to \mathsf{int}, \mathsf{I}_2 :: \mathsf{int} \to \mathsf{int}, \mathsf{I}_3 :: \mathsf{bool}$. Then an example of an identifier declaration dec_I is

$$\begin{aligned} \mathsf{I}_1\, l\, y &= \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(l))) + \mathsf{I}_2 y \\ \mathsf{I}_2 x &= x * x \\ \mathsf{I}_3 &= \underline{T} \\ \mathsf{I}_4\, u\, v\, w &= u + v + w \end{aligned}$$

Defining Declarations

Let $I = \mathsf{I}_1 :: \iota_1, \ldots, \mathsf{I}_m :: \iota_m$ where for example

$$\iota_j = \sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_k \to \sigma_j. \quad (j \in \{1, \ldots, m\})$$

Then an **identifier declaration** dec_I consists of

$$\begin{array}{ccc} & \vdots & \\ \mathsf{I}_j\, x_1 \ldots x_k & = & E_{\mathsf{I}_j} \\ & \vdots & \end{array}$$

for each $j \in \{1, \ldots, m\}$

An Example Program

Let $I = \mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{K} :: \mathsf{int}$. Then an identifier declaration dec_I is

$$\begin{aligned} \mathsf{F}\,x\,y &= x + \underline{7} - y \\ \mathsf{K} &= \underline{10} \end{aligned}$$

An example of a program is $\boxed{dec_I \quad in\ \ \mathsf{F}\,\underline{8}\,\underline{1} \leq \mathsf{K}}$. Note that

$$\varnothing \vdash \mathsf{F}\,\underline{8} \leq \mathsf{K} :: \mathsf{bool}$$

and that

$$x :: \mathsf{int}, y :: \mathsf{int} \vdash x + \underline{7} - y :: \mathsf{int} \qquad \text{and} \qquad \varnothing \vdash \mathsf{K} :: \mathsf{int}$$

Programs

A **program expression** P is any expression containing no variables. A **program** in $\mathbb{FUN}^e$ is a judgement of the form

$$dec_I \quad in \;\; P \qquad \text{where} \qquad \varnothing \vdash P :: \sigma$$

and the declarations in dec_I satisfy

$$\vdots$$

$$x_1 :: \sigma_1, \ldots, x_k :: \sigma_k \vdash E_{\mathsf{I}_j} :: \sigma_j$$

$$\vdots$$

Example Programs

$$\mathsf{F}\,x = \mathsf{if}\ x \leq \underline{1}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ x * \mathsf{F}\,(x - \underline{1}) \quad \textit{in}\ \ \mathsf{F}\,\underline{4}$$

$$\left.\begin{array}{lcl} \mathsf{F}_1\,x y z & = & \mathsf{if}\ x \leq \underline{1}\ \mathsf{then}\ y\ \mathsf{else}\ z \\ \mathsf{F}_2\,x & = & \mathsf{F}_1\,x\,\underline{1}\,(x * \mathsf{F}_2\,(x - \underline{1})) \end{array}\right\} \quad \textit{in}\ \ \mathsf{F}_2\,\underline{4}$$

$$\mathsf{G}\,l = \ \textit{code to sort}\ \ l \qquad \textit{in} \qquad \mathsf{G}\,(\underline{3} : \underline{6} : \underline{-2} : \underline{8} : \mathsf{nil})$$

Chapter 7

- Explain *call-by-value (eager)* and *call-by-need (lazy)* function calling methods.
- Give FUN an eager and lazy evaluation style operational semantics.
- Prove properties such as *determinism*.
- Extend the language to give local declarations.

Overview: Programs and Values

- Look at the notion of evaluation order.
- Define *values*, which are the results of eager program executions.
- Define an *eager evaluation semantics*: $P \Downarrow^e V$.
- Give some examples.

Evaluation Orders

- The operational semantics of $\mathbb{FUN}^e$ says when a program P evaluates to a value V. It is like the IMP evaluation semantics.

- Write this in general as $P \Downarrow^e V$, and examples are

$$\underline{3} + \underline{4} + \underline{10} \Downarrow^e \underline{17} \qquad \text{and} \qquad \mathsf{hd}(\underline{2} : \mathsf{nil}_{\mathsf{int}}) \Downarrow^e \underline{2}$$

- Let $\mathsf{F}\,x\,y = x + y$. We would expect $\mathsf{F}\,(\underline{2} * \underline{3})\,(\underline{4} * \underline{5}) \Downarrow^{e} \underline{26}$.

- We could

  - evaluate $\underline{2} * \underline{3}$ to get value $\underline{6}$ yielding $\mathsf{F}\,\underline{6}\,(\underline{4} * \underline{5})$,
  - then evaluate $\underline{4} * \underline{5}$ to get value $\underline{20}$ yielding $\mathsf{F}\,\underline{6}\,\underline{20}$.

- We then *call* the function to get $\underline{6} + \underline{20}$, which evaluates to $\underline{26}$. This is *call-by-value* or *eager* evaluation.

- Or the function could be called first yielding $(\underline{2} * \underline{3}) + (\underline{4} * \underline{5})$ and then we continue to get $\underline{6} + (\underline{4} * \underline{5})$ and $\underline{6} + \underline{20}$ and $\underline{26}$. This is called *call-by-name* or *lazy* evaluation.

- The *order* of evaluation is different.

Defining and Explaining (Eager) Values

- Let dec_I be a identifier declaration, with typical typing

$$\mathsf{F} :: \sigma_1 \to \sigma_2 \to \sigma_3 \to \ \ldots \to \sigma_k \to \sigma$$

A **value expression** is any expression V produced by

$$V ::= \underline{c} \mid \mathsf{nil}_\sigma \mid (V, V) \mid \mathsf{F}\,\vec{V} \mid V : V$$

where $\vec{V}$ abbreviates $V_1\,V_2\,\ldots\,V_{l-1}\,V_l$ and $0 \leq l < k$, and k is the maximum number of inputs taken by F. **CARE!!!**

- Note that constants K are *not* values. Note also that l is *strictly* less than k, and that if $k = 1$ then $\mathsf{F}\,\vec{V}$ denotes F.

- A **value** is any value expression for which $dec_I \quad in \;\; V$ is a valid $\mathbb{FUN}^e$ program.

- Suppose that $\mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and that $P_1 \Downarrow^e \underline{2}$ and $P_2 \Downarrow^e \underline{5}$ and $P_3 \Downarrow^e \underline{7}$ with P_i not values. Then

P	V
	F
$\mathsf{F}\,P_1$	$\mathsf{F}\,\underline{2}$
$\mathsf{F}\;\underline{2}\;P_2$	$\mathsf{F}\;\underline{2}\;\underline{5}$

P	V
$\mathsf{F}\;\underline{2}\;\underline{5}\;P_3$	
$\mathsf{F}\;\underline{2}\;\underline{5}\;\underline{7}$	$\underline{14}$
$\mathsf{F}\;P_1\;P_2\;P_3$	$\underline{14}$

- Of course $\mathsf{F}\;P_1\;P_2\;P_3 \Downarrow^e \underline{14}$.

$$\frac{}{V \Downarrow^e V} \Downarrow^e\text{VAL} \qquad \frac{P_1 \Downarrow^e \underline{m} \quad P_2 \Downarrow^e \underline{n}}{P_1\ op\ P_2 \Downarrow^e \underline{m\ op\ n}} \Downarrow^e\text{OP}$$

$$\frac{P_1 \Downarrow^e \underline{T} \quad P_2 \Downarrow^e V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V} \Downarrow^e\text{COND}_1 \qquad \frac{P_1 \Downarrow^e \underline{F} \quad P_3 \Downarrow^e V}{\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \Downarrow^e V} \Downarrow^e\text{COND}_2$$

$$\frac{P_1 \Downarrow^e V_1 \quad P_2 \Downarrow^e V_2}{(P_1, P_2) \Downarrow^e (V_1, V_2)} \Downarrow^e\text{PAIR}$$

$$\frac{P \Downarrow^e (V_1, V_2)}{\mathsf{fst}(P) \Downarrow^e V_1} \Downarrow^e\text{FST} \qquad \frac{P \Downarrow^e (V_1, V_2)}{\mathsf{snd}(P) \Downarrow^e V_2} \Downarrow^e\text{SND}$$

$$\frac{\left\{\begin{array}{l} P_1 \Downarrow^e \mathsf{F}\vec{V} \quad P_2 \Downarrow^e V_2 \quad \mathsf{F}\vec{V}\, V_2 \Downarrow^e V \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value} \end{array}\right.}{P_1\, P_2 \Downarrow^e V} \Downarrow^e\mathrm{AP}$$

$$\frac{E_{\mathsf{F}}[V_1, \ldots, V_{k_j}/x_1, \ldots, x_k] \Downarrow^e V}{\mathsf{F} V_1 \ldots V_k \Downarrow^e V} [\mathsf{F}\vec{x} = E_{\mathsf{F}} \text{ declared in } dec_I] \Downarrow^e\mathrm{FID}$$

$$\frac{E_{\mathsf{K}} \Downarrow^e V}{\mathsf{K} \Downarrow^e V} [\mathsf{K} = E_{\mathsf{K}} \text{ declared in } dec_I] \Downarrow^e\mathrm{CID}$$

$$\frac{P \Downarrow^e \mathsf{nil}_\sigma}{\mathsf{tl}(P) \Downarrow^e \mathsf{nil}_\sigma} \Downarrow^e\text{NIL} \qquad \frac{P \Downarrow^e V : V'}{\mathsf{hd}(P) \Downarrow^e V} \Downarrow^e\text{HD} \qquad \frac{P \Downarrow^e V : V'}{\mathsf{tl}(P) \Downarrow^e V'} \Downarrow^e\text{TL}$$

$$\frac{P_1 \Downarrow^e V \quad P_2 \Downarrow^e V'}{P_1 : P_2 \Downarrow^e V : V'} \Downarrow^e\text{CONS}$$

$$\frac{P \Downarrow^e \mathsf{nil}_\sigma}{\mathsf{elist}(P) \Downarrow^e \underline{T}} \Downarrow^e\text{ELIST}_1 \qquad \frac{P \Downarrow^e V : V'}{\mathsf{elist}(P) \Downarrow^e \underline{F}} \Downarrow^e\text{ELIST}_2$$

Examples of Evaluations

Suppose that dec_I is

$$\begin{array}{rcl} \mathsf{G}\,x & = & x * \underline{2} \\ \mathsf{K} & = & \underline{3} \end{array}$$

$$\dfrac{\dfrac{}{\mathsf{G} \Downarrow^e \mathsf{G}}\,\text{VAL} \qquad \dfrac{\dfrac{}{\underline{3} \Downarrow^e \underline{3}}\,\text{VAL}}{\mathsf{K} \Downarrow^e \underline{3}}\,\text{CID} \qquad \dfrac{\dfrac{\dfrac{}{\underline{3} \Downarrow^e \underline{3}}\,\text{VAL} \qquad \dfrac{}{\underline{2} \Downarrow^e \underline{2}}\,\text{VAL}}{(x * \underline{2})[\underline{3}/x] = \underline{3} * \underline{2} \Downarrow^e \underline{6}}\,\text{OP}}{\mathsf{G}\,\underline{3} \Downarrow^e \underline{6}}\,\text{FID}}{\mathsf{G}\,\mathsf{K} \Downarrow^e \underline{6}}\,\text{AP}$$

$$\dfrac{\dfrac{\dfrac{}{(\mathsf{F},\mathsf{G}) \Downarrow^e (\mathsf{F},\mathsf{G})}\,\text{VAL}}{\mathsf{snd}((\mathsf{F},\mathsf{G})) \Downarrow^e \mathsf{G}}\,\text{SND} \qquad \mathcal{D} \qquad \dfrac{\dfrac{\dfrac{}{\underline{4} \Downarrow^e \underline{4}}\,\text{VAL} \qquad \dfrac{}{\underline{2} \Downarrow^e \underline{2}}\,\text{VAL}}{\underline{4} * \underline{2} \Downarrow^e \underline{8}}\,\text{OP}}{\mathsf{G}\,\underline{4} \Downarrow^e \underline{8}}\,\text{FID}}{\mathsf{snd}((\mathsf{F},\mathsf{G}))\,\underline{4} \Downarrow^e \underline{8}}\,\text{AP}$$

Let

$$\mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \quad \text{where} \quad \mathsf{F}\,x\,y\,z = x + y + z$$

- $\mathsf{F}\,\underline{2}$ and $\mathsf{F}\,\underline{2}\,\underline{3}$ are (programs and) values.
- $\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})$ is a program, but not a value
- Note that $\mathsf{F}\,\underline{2}\,\underline{3}$ is sugar for $(\mathsf{F}\,\underline{2})\,\underline{3}$ and that $\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1})$ is sugar for $((\mathsf{F}\,\underline{2})\,\underline{3})\,(\underline{4}+\underline{1})$.
- In the Definitions of values, $k = 3$, and in $\mathsf{F}\,\underline{2}\,\underline{3}$ we have $\vec{V} = \underline{2}\,\underline{3}$ and $l = 2 < 3$.

We can prove that

$$\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1}) \Downarrow^e \underline{10}$$

where $\mathsf{F}\,x\,y\,z = x+y+z$ as follows:

$$\dfrac{\dfrac{}{\mathsf{F}\,\underline{2}\,\underline{3} \Downarrow^e \mathsf{F}\,\underline{2}\,\underline{3}}\Downarrow^e \text{VAL} \qquad \dfrac{\dfrac{}{\underline{4} \Downarrow^e \underline{4}} \qquad \dfrac{}{\underline{1} \Downarrow^e \underline{1}}}{\underline{4}+\underline{1} \Downarrow^e \underline{5}} \qquad T}{\mathsf{F}\,\underline{2}\,\underline{3}\,(\underline{4}+\underline{1}) \Downarrow^e \underline{10}}\Downarrow^e \text{AP}$$

where T is the tree

$$
\dfrac{
\begin{array}{c}
\dfrac{
\dfrac{
\dfrac{}{\underline{2} \Downarrow^e \underline{2}} \qquad \dfrac{}{\underline{3} \Downarrow^e \underline{3}}
}{\underline{2}+\underline{3} \Downarrow^e \underline{5}}
\qquad
\dfrac{}{\underline{5} \Downarrow^e \underline{5}}
}{\underline{2}+\underline{3}+\underline{5} \Downarrow^e \underline{10}} \\
\hline\hline
(x+y+z)[\underline{2},\underline{3},\underline{5}/x,y,z] \Downarrow^e \underline{10}
\end{array}
}{\mathsf{F}\,\underline{2}\,\underline{3}\,\underline{5} \Downarrow^e \underline{10}}
\;\Downarrow^e \text{FID}
$$

Overview: FUN Properties of Eager Evaluation

- Explain and define *determinism*.
- Explain and define *subject reduction*, that is, preservation of types during program execution.

Properties of FUN

- The evaluation relation for $\mathbb{FUN}^e$ is **deterministic**. More precisely, for all P, V_1 and V_2, if

$$P \Downarrow^e V_1 \ \mathit{and} \ P \Downarrow^e V_2$$

then $V_1 = V_2$. (Thus $\Downarrow^e$ is a *partial function*.)

- Evaluating a program $dec_I \quad in \ P$ does not alter its type. More precisely,

$$(\varnothing \vdash P :: \sigma \ \mathit{and} \ P \Downarrow^e V) \quad \mathit{implies} \quad \varnothing \vdash V :: \sigma$$

for any P, V, σ and I. The conservation of type during program evaluation is called **subject reduction**.

Proving Determinism

To prove determinism, we prove by Rule Induction that

$$\forall P \Downarrow^e V_1. \quad \boxed{\forall V_2. \, (P \Downarrow^e V_2 \ \textit{implies} \ V_1 = V_2)}$$

See the board ...

Proving Subject Reduction

We prove by Rule Induction that given $dec_I \quad in \;\; P$

$$\forall P \Downarrow^e V. \quad \boxed{\forall \sigma (\varnothing \vdash P :: \sigma \quad implies \quad \varnothing \vdash V :: \sigma).}$$

The tricky rule is

$$\frac{E_{\mathsf{F}}[V_1,\ldots,V_{k_j}/x_1,\ldots,x_k] \Downarrow^e V}{\mathsf{F} V_1 \ldots V_k \Downarrow^e V} \; [\mathsf{F}\vec{x} = E_{\mathsf{F}} \text{ declared in } dec_I] \;\; \Downarrow^e\text{FID}$$

Suppose that $\varnothing \vdash \mathsf{F} V_1 \ldots V_k :: \sigma$ where σ is any type. Then we need to prove $\varnothing \vdash V :: \sigma$. By the induction hypothesis, we just need to prove $\varnothing \vdash E_{\mathsf{F}}[V_1,\ldots,V_{k_j}/x_1,\ldots,x_k] :: \sigma$.

Overview: Programs and (Lazy) Values

- Define *values*, which are the results of program executions.
- Define a *lazy evaluation semantics*: $P \Downarrow^{l} V$.
- Give some examples.

Defining and Explaining Values

- Let dec_I be a identifier declaration, with typical typing

$$\mathsf{F} :: \sigma_1 \to \sigma_2 \to \sigma_3 \to \ldots \to \sigma_k \to \sigma$$

A **value expression** is any expression V produced by the grammar

$$V ::= \underline{c} \mid \mathsf{nil}_\sigma \mid (P,P) \mid \mathsf{F}\vec{P} \mid P : P$$

where $\vec{P}$ abbreviates $P_1 P_2 \ldots P_{l-1} P_l$ and $0 \leq l < k$, and k is the maximum number of inputs taken by F.

- A **value** is any value expression for which $dec_I \quad in \;\; V$ is a valid $\mathbb{FUN}^l$ program.

$$\frac{P \Downarrow^l (P_1, P_2) \quad P_1 \Downarrow^l V}{\mathsf{fst}(P) \Downarrow^l V} \Downarrow^l \text{FST}$$

$$\frac{\left\{ \begin{array}{l} P_1 \Downarrow^l \mathsf{F}\vec{P} \quad \mathsf{F}\vec{P}\,P_2 \Downarrow^l V \\ \\ \text{where either } P_1 \text{ or } P_2 \text{ is not a value} \end{array} \right.}{P_1\,P_2 \Downarrow^l V} \Downarrow^l \text{AP}$$

$$\frac{E_{\mathsf{F}}[P_1, \ldots, P_k / x_1, \ldots, x_k] \Downarrow^l V}{\mathsf{F} P_1 \ldots P_k \Downarrow^l V} \, [\mathsf{F}\vec{x} = E_{\mathsf{F}} \text{ declared in } dec_I] \;\; \Downarrow^l \text{FID}$$

$$\frac{P_1 \Downarrow^l P_2 : P_3 \quad P_2 \Downarrow^l V}{\mathsf{hd}(P_1) \Downarrow^l V} \Downarrow^l \text{HD}$$

Examples of Evaluations

Let I be $\mathsf{F} :: \mathsf{int} \to [\mathsf{int}]$, and dec_I be $\mathsf{F}x = x : \mathsf{F}\,(x + \underline{2})$. Then there is a program $dec_I \quad in\ \ \mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1}))$. We prove that $\mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1})) \Downarrow^l \underline{3}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\underline{1} : \mathsf{F}\,(\underline{1}+\underline{2}) \Downarrow^l \underline{1} : \mathsf{F}\,(\underline{1}+\underline{2})}\Downarrow^l \text{VAL}}{\mathsf{F}\,\underline{1} \Downarrow^l \underline{1} : \mathsf{F}\,(\underline{1}+\underline{2})}\Downarrow^l \text{FID} \qquad T_1}{\mathsf{tl}(\mathsf{F}\,\underline{1}) \Downarrow^l (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})} \qquad T_2}{\mathsf{hd}(\mathsf{tl}(\mathsf{F}\,\underline{1})) \Downarrow^l \underline{3}}$$

T_2

$$\dfrac{\dfrac{}{\underline{1} \Downarrow^{l} \underline{1}} \qquad \dfrac{}{\underline{2} \Downarrow^{l} \underline{2}}}{\underline{1} + \underline{2} \Downarrow^{l} \underline{3}}$$

T_1

$$\dfrac{\dfrac{}{(\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2}) \Downarrow^{l} (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})} \Downarrow^{l} \text{VAL}}{\mathsf{F}\,(\underline{1}+\underline{2}) \Downarrow^{l} (\underline{1}+\underline{2}) : \mathsf{F}\,((\underline{1}+\underline{2})+\underline{2})}$$

Let $\mathsf{large}\, x = \underline{1} + \mathsf{large}\, x \quad \mathit{in}\ \ \mathsf{fst}((\underline{3}, \mathsf{large}\, \underline{0}))$. We try to evaluate this programme to a value V

$$\dfrac{\dfrac{}{(\underline{3}, \mathsf{large}\, \underline{0}) \Downarrow^{l} (P_1, P_2)}\ R \qquad \dfrac{}{P_1 \Downarrow^{l} V}\ R'}{\mathsf{fst}((\underline{3}, \mathsf{large}\, \underline{0})) \Downarrow^{l} V}\ \Downarrow^{l}\ \text{FST}$$

for which we must have $P_1 = \underline{3}$, $P_2 = \mathsf{large}\, \underline{0}$, $V = \underline{3}$ and R and R' are both instances of $\Downarrow^{l}$ VAL.

Overview: Locality

- Explain unnamed functions and local definitions.
- Describe *free* and *bound* variables.
- Extend the syntax of FUN, and its operational semantics.
- Give some examples.

Motivating Functions and Locality

- We can define *unnamed* functions. The expression

$$\mathsf{fn}\, x.x + \underline{2}$$

  is a program whose intended meaning is the function which "adds 2". But it is not (necessarily) named by an identifer.

- $(\mathsf{fn}\, x.x + \underline{2})\, \underline{4}$ will evaluate to $\underline{4} + \underline{2}$ (and thus to $\underline{6}$).

- If $\mathsf{F}\, x = x + \underline{2}$ then F and $\mathsf{fn}\, x.x + \underline{2}$ would be interchangeable. F is the *name* of the function.

- The syntax $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$ gives *local declarations*. For example $\mathsf{let}\ x = \underline{5}\ \mathsf{in}\ x + y + x$.

- We explain "local" with the next example:

$$\mathsf{let}\ x = \underline{7}\ \mathsf{in}\ (x, \mathsf{let}\ x = \underline{5}\ \mathsf{in}\ x + y + x)$$

Syntax and Type Assignments

$$E ::= \ldots \mid \mathsf{fn}\, x.E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E$$

- We call $\mathsf{fn}\, x.E$ a **function abstraction**. We call E the **body** of $\mathsf{fn}\, x.E$.
- We call $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$ a **local declaration**.

$$\frac{\Gamma \vdash E_1 :: \sigma \quad \Gamma \vdash E_2[E_1/x] :: \sigma'}{\Gamma \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 :: \sigma'}\ \text{LET} \qquad \frac{\Gamma, x :: \sigma \vdash E :: \tau}{\Gamma \vdash \mathsf{fn}\, x.E :: \sigma \to \tau}\ \text{ABS}$$

Conventions and Examples

- $\mathsf{fn}\,x.E$ means $\mathsf{fn}\,x.(E)$
- $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$ means $\mathsf{let}\ x = E_1\ \mathsf{in}\ (E_2)$
- Thus $\mathsf{fn}\,x.\mathsf{fn}\,y.y + \underline{2} = \mathsf{fn}\,x.(\mathsf{fn}\,y.(y + \underline{2}))$
- $\mathsf{fn}\,x.\mathsf{fn}\,y.x + y + \underline{2} = \mathsf{fn}\,x.(\mathsf{fn}\,y.((x + y) + \underline{2}))$
-

$$\mathsf{let}\ x = \underline{4}\ \mathsf{in}\ \mathsf{let}\ y = \underline{T}\ \mathsf{in}\ (x,y) = \mathsf{let}\ x = \underline{4}\ \mathsf{in}\ (\mathsf{let}\ y = \underline{T}\ \mathsf{in}\ (x,y))$$

Motivating Free and Bound Variables

- Write $F \stackrel{\text{def}}{=} \mathsf{fn}\, x.E_1$. Given any expression E_2, in a transition semantics

$$F\, E_2 \rightsquigarrow E_1[E_2/x]$$

Thus if E_1 is $x+y$, then

$$F\, E_2 \rightsquigarrow (x+y)[E_2/x] \stackrel{\text{def}}{=} E_2+y$$

and the intended meaning of $F = \mathsf{fn}\, x.x+y$ is "the function with adds y".

- $E[x/y]$ ought to be "the function which adds x". But in fact $E[x/y]$ is clearly the expression $\mathsf{fn}\, x.x+x$, which is the function which doubles an integer input!

- We say that the substituted x falls in the *scope* of the *scoping* x.

- The expressions $\mathsf{fn}\, x.x+y$ and $\mathsf{fn}\, x'.x'+y$ can be regarded as "the same". We say that x and x' are *bound*, and y is *free*.

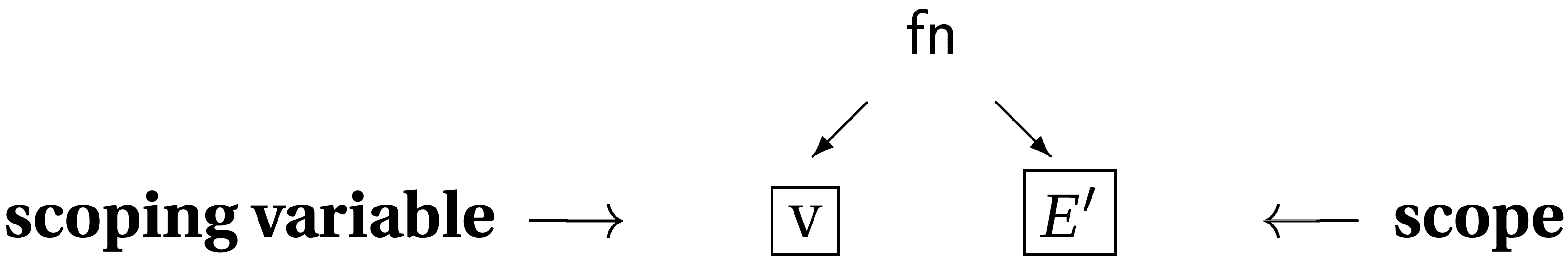
- Note that

$$(\mathsf{fn}\, x.x+y)[x/y] = \mathsf{fn}\, x'.x'+x$$

- We *re-name* the bound variable x in $\mathsf{fn}\, x.x+y$ as a new variable x' so that when x is substituted for y it does not become bound.

Definitions of Free and Bound Variables

- The syntax tree for $\mathsf{fn}\, v.E'$ looks like this



- In $\mathsf{let}\ v = E_1\ \mathsf{in}\ E_2$, the **scope** of v is E_2. We also call such a v a **scoping** variable.

■ Suppose x does occur in E. Each *occurrence* of x (in E) is either free or bound (but not both!!).

■ We say that an occurrence of x is **bound** if and only if the occurrence of x is in a *subexpression* of the form

- $\mathsf{fn}\, x.E'$ or
- $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$ where the occurrence is in E_2.

■ Thus an occurrence of x in E is bound just in case

– the occurrence is a scoping variable;

– the occurrence occurs within the scope of a scoping occurrence of x.

- If there is an occurrence of x in such E' or E_2 then we sometimes say that this bound occurrence of x has been **captured** by the scoping x.
- An occurrence of x in E is **free** iff the occurrence of x is not bound.

Substitution Examples

$$
\begin{array}{rcl}
(\mathsf{fn}\, x.x+y)[\underline{2}/y] & = & \mathsf{fn}\, x.x+\underline{2} \\
(\mathsf{fn}\, x.x+y)[x/y] & = & \mathsf{fn}\, x'.x'+x \\
(\mathsf{let}\ x = y+\underline{4}\ \mathsf{in}\ x+z+\underline{7})[u+v/z] & = & \mathsf{let}\ x = y+\underline{4}\ \mathsf{in}\ x+(u+v)+\underline{7} \\
(\mathsf{let}\ x = y+\underline{4}\ \mathsf{in}\ x+z+\underline{7})[u+y/z] & = & \mathsf{let}\ x = y+\underline{4}\ \mathsf{in}\ x+(u+y)+\underline{7} \\
(\mathsf{let}\ x = z+\underline{4}-x\ \mathsf{in}\ x+z+\underline{7})[x+y/z] & = & \\
\multicolumn{3}{r}{\mathsf{let}\ x' = (x+y)+\underline{4}-x\ \mathsf{in}\ x'+(x+y)+\underline{7}} \\
(\mathsf{let}\ u = u\ \mathsf{in}\ u+\underline{7})[\underline{7}/u] & = & \mathsf{let}\ u = \underline{7}\ \mathsf{in}\ u+\underline{7}
\end{array}
$$

Extending the Eager Semantics

$$\frac{P_1 \Downarrow^e \mathsf{fn}\, x.E \quad P_2 \Downarrow^e V' \quad E[V'/x] \Downarrow^e V}{P_1\, P_2 \Downarrow^e V} \Downarrow^e{}_{\mathrm{AA}}$$

$$\frac{E_1 \Downarrow^e V_1 \quad E_2[V_1/x] \Downarrow^e V}{\mathsf{let}\; x = E_1 \;\mathsf{in}\; E_2 \Downarrow^e V} \Downarrow^e{}_{\mathrm{LET}}$$

An Example

$$
\dfrac{\dfrac{\dfrac{}{\mathsf{nil} \Downarrow^e \mathsf{nil}}\,\text{VAL} \qquad \dfrac{? \quad ? \qquad \dfrac{\dfrac{}{\underline{3} \Downarrow^e \underline{3}}\,\text{VAL} \qquad \dfrac{}{\underline{2} \Downarrow^e \underline{2}}\,\text{VAL}}{(x+\underline{2})[\underline{3}/x] = \underline{3}+\underline{2} \Downarrow^e \underline{5}}\,\text{OP}}{(\mathsf{fn}\, x.x+\underline{2})\,\underline{3} \Downarrow^e \underline{5}}\,\text{AA}}{(\mathsf{fn}\, x.x+\underline{2})\,\underline{3} : \mathsf{nil} \Downarrow^e \underline{5} : \mathsf{nil}}\,\text{CONS}}{\mathsf{hd}((\mathsf{fn}\, x.x+\underline{2})\,\underline{3} : \mathsf{nil}) \Downarrow^e \underline{5}}\,\text{HD}
$$

Chapter 8

- Give overview of polymorphism.
- Introduce type variables into $\mathbb{FUN}^e$.
- Give examples of type assignments.
- Explain *local polymorphism*.
- Explain the *polymorphic type inference algorithm*.

Overview: Simple Type Deductions with Variables

- Explain different kinds of polymorphism.
- Give examples of type assignment deductions.

Varieties of Type System

- A language is **strongly typed** if every legal expression has at least one type.
- A strongly typed language is **monomorphic** if every legal expression has a unique type (for example Pascal).
- A strongly typed language is **polymorphic** if a legal expression can have several types (for example Standard ML and Haskell and Java).

- Overloading: The same symbol is used to denote (finitely many) functions, implemented by *different* algorithms.

- Parametric: One expression belongs to a family of *structurally related* types. The expression encodes *one* algorithm which works at *each* type in the family. An example is list sorting.

- Implicit: This is a particular form of parametric polymorphism, and we meet it later on.

PFUN Type System

- The set *Type* of **types** of $\mathbb{PFUN}$ is inductively specified by the grammar

$$\sigma \quad ::= \quad X \mid \mathsf{int} \mid \mathsf{bool} \mid \sigma \to \sigma \mid (\sigma, \sigma) \mid [\sigma]$$

- Each type is a finite tree. Two types are **equal** if the trees are identical. Examples on the board.
- We shall write $TV(\sigma)$ for the set of type variables appearing in σ.
- The rules for deriving type assignments are as before.
- $\mathbb{PFUN}$ expressions are from the extended language.

Examples of Type Assignment Deductions

Prove that $\vdash \underline{T} : \mathsf{nil} :: [\mathsf{bool}]$.

$$\dfrac{\dfrac{}{\vdash \underline{T} :: \mathsf{bool}}\,\text{TRUE} \qquad \dfrac{}{\vdash \mathsf{nil} :: [\mathsf{bool}]}\,\text{NIL}}{\vdash \underline{T} : \mathsf{nil} :: [\mathsf{bool}]}\,\text{CONS}$$

Show that $\Gamma \vdash \mathsf{fn}\, x.(\underline{0} : x) :: [\mathsf{int}] \to [\mathsf{int}]$ for any context Γ.

We produce a deduction tree: note that the expression is a function, so the final rule used in the deduction must be ABS, where $E = \underline{0} : x$, and $\sigma = \tau = [\mathsf{int}]$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x :: [\mathsf{int}] \vdash \underline{0} :: \mathsf{int}}\ \text{INT} \qquad \dfrac{}{\Gamma, x :: [\mathsf{int}] \vdash x :: [\mathsf{int}]}\ \text{VAR}}{\Gamma, x :: [\mathsf{int}] \vdash \underline{0} : x :: [\mathsf{int}]}\ \text{CONS}}{\Gamma \vdash \mathsf{fn}\, x.(\underline{0} : x) :: [\mathsf{int}] \to [\mathsf{int}]}\ \text{ABS}$$

Show that $\mathsf{hd}(y : \underline{3})$ is not typable in $\mathbb{PFUN}$ in any context Γ.

Working backwards we have:

$$\dfrac{\dfrac{}{\Gamma \vdash y :: \sigma}\,\text{VAR} \qquad \dfrac{}{\Gamma \vdash \underline{3} :: [\sigma]}\,\text{INT}}{\dfrac{\Gamma \vdash y : \underline{3} :: [\sigma]}{\Gamma \vdash \mathsf{hd}(y : \underline{3}) :: \sigma}\,\text{HD}}\,\text{CONS}$$

Looking at the rule INT (which must be used to type $\underline{3}$) we must have $\mathsf{int} = [\sigma]$, a contradiction. So the expression cannot be typable.

Show that $\vdash \mathsf{fn}\ f.(f\ \mathsf{nil}, \underline{T}) :: ([X] \to Y) \to (Y, \mathsf{bool})$.

$$\dfrac{}{f :: [X] \to Y \vdash f :: [X] \to Y}\ \text{VAR}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f :: [X] \to Y \vdash f :: [X] \to Y}\ \text{VAR} \qquad \dfrac{}{f :: [X] \to Y \vdash \mathsf{nil} :: [X]}\ \text{NIL}}{f :: [X] \to Y \vdash f\ \mathsf{nil} :: Y}\ \text{AP} \qquad \mathcal{D}}{f :: [X] \to Y \vdash (f\ \mathsf{nil}, \underline{T}) :: (Y, \mathsf{bool})}\ \text{PAIR}}{\vdash \mathsf{fn}\ f.(f\ \mathsf{nil}, \underline{T}) :: ([X] \to Y) \to (Y, \mathsf{bool})}\ \text{ABS}$$

Show that $(\mathsf{fn}\, f.\, f y)\, y$ is not typable for *any* context of the form $y :: \tau$. (Note that y is the only free variable).

We suppose, for a contradiction, that the expression is typeable. Let us call this type σ_1, say. We have:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y :: \tau, f :: \sigma_2 \vdash f :: \sigma_3 \to \sigma_1}\,\text{VAR} \qquad \dfrac{}{y :: \tau, f :: \sigma_2 \vdash y :: \sigma_3}\,\text{VAR}}{y :: \tau, f :: \sigma_2 \vdash f y :: \sigma_1}\,\text{AP}}{y :: \tau \vdash \mathsf{fn}\, f.\, f y :: \sigma_2 \to \sigma_1}\,\text{ABS} \qquad \mathcal{D}}{y :: \tau \vdash (\mathsf{fn}\, f.\, f y)\, y :: \sigma_1}\,\text{AP}$$

where $\mathcal{D}$ is

$$\dfrac{}{y :: \tau \vdash y :: \sigma_2}\,\text{VAR}$$

Motivating Type Substitutions

- $\underline{6} + \underline{T}$ has no type.
- $\underline{1} :: \sigma$ holds only for $\sigma = \mathsf{int}$.
- However, $\vdash \mathsf{fn}\, x.x :: \sigma \to \sigma$ holds for any type σ.
- In $\mathbb{PFUN}$, of all the types that can be assigned to an expression, there is a "most general" one: all other types are instances of it. We call this the *principal* type.
- The principal type of $\mathsf{fn}\, x.x$ is $X \to X$; any type $\sigma \to \sigma$ is obtained by *substituting* σ for X.

Type Substitutions

- Define $S \stackrel{\text{def}}{=} \langle X \mapsto U, Y \mapsto \mathsf{bool} \rangle$. Let $\sigma \stackrel{\text{def}}{=} (X, Y \to Z)$.

Then

$$S\{\sigma\} = (U, \mathsf{bool} \to Z)$$

- S a **type substitution** if it is a (possibly empty) finite set of (type-variable,type) pairs in which *all the type-variables are distinct.*

- We will write a typical S in the form

$$\langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle$$

We write the empty type substitution as $\langle \rangle$.

- If τ is any type, we shall write $S\{\tau\}$ to denote the type τ in which any occurrence of X_i is changed to σ_i. Thus

$$\langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle\{\tau\} \stackrel{\text{def}}{=} \tau[\sigma_1, \ldots, \sigma_n / X_1, \ldots, X_n]$$

- We will define equality of type substitutions in a similar way to function equality, namely

$$S = S' \text{ iff } \forall \tau. \quad S\{\tau\} = S'\{\tau\}$$

- Given substitutions S_1 and S_2 we define the effect of the substitution $S_1 \cdot S_2$ by setting $(S_1 \cdot S_2)\{\tau\} \stackrel{\text{def}}{=} S_1\{S_2\{\tau\}\}$.

  Warning!! A type substitution is a set of (type-variable,type) pairs. What set is $S_1 \cdot S_2$?

- If $S \stackrel{\text{def}}{=} \langle V \mapsto \sigma, X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle$ then we define S^V to be $\langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n \rangle$ and also $\langle\rangle^V$ to be $\langle\rangle$.

- σ **generalises** σ' if there exists a type substitution S for which

$$\sigma' = S\{\sigma\}$$

and say that σ' is an **instance** of σ.

- In $\mathbb{PFUN}$, if $\varnothing \vdash P :: \sigma$, the type σ assigned to the expression P is **principal** if σ generalises any other type which can be assigned to P.

- The principal type of $\mathsf{fn}\, x.x$ is $X \to X$. Note that the principal type is unique up to a consistent renaming of variables. Another principal type for $\mathsf{fn}\, x.x$ is $V \to V$.

Type Substitution Examples

- Define $S \stackrel{\text{def}}{=} \langle X \mapsto U, Y \mapsto \mathsf{bool}\rangle$. Let $\sigma \stackrel{\text{def}}{=} (X, Y \to Z)$ and $\Gamma \stackrel{\text{def}}{=} x :: X, y :: Y \to Z$. Then

$$S\{\sigma\} = (U, \mathsf{bool} \to Z)$$

and

$$S\{\Gamma\} = x :: S\{X\}, y :: S\{Y \to Z\} = x :: U, y :: \mathsf{bool} \to Z$$

- Note that $(X,Y) \to Z$ generalises $([\mathsf{bool}],Y) \to \mathsf{int}$ for

$$([\mathsf{bool}],Y) \to \mathsf{int} = S\{((X,Y) \to Z)\}$$

where $S \stackrel{\text{def}}{=} \langle X \mapsto [\mathsf{bool}], Z \mapsto \mathsf{int}\rangle$

- It follows from the definitions that $\langle X \mapsto X \rangle = \langle \rangle$.

- The definition of composition of type substitutions does not describe $S_1 \cdot S_2$ as an explicit set of pairs. Consider $\langle X \mapsto \mathsf{int}, Y \mapsto X \rangle \cdot \langle Z \mapsto \mathsf{int} \rangle$. The composition is

$$\langle X \mapsto \mathsf{int}, Y \mapsto X, Z \mapsto \mathsf{int} \rangle$$

- Now consider $\langle X \mapsto \mathsf{int}, Y \mapsto X \rangle \cdot \langle Y \mapsto \mathsf{int} \rangle$. The composition is

$$\langle X \mapsto \mathsf{int}, Y \mapsto \mathsf{int} \rangle$$

- $$\langle X \mapsto \mathsf{bool}, Y \mapsto X\rangle \cdot \langle Z \mapsto Y\rangle = \langle X \mapsto \mathsf{bool}, Y \mapsto X, Z \mapsto X\rangle$$

- $$\begin{aligned} &\langle X \mapsto \mathsf{bool}, Y \mapsto U\rangle \cdot \langle Y \mapsto X, Z \mapsto Y\rangle \\ &\qquad = \langle Y \mapsto \mathsf{bool}, X \mapsto \mathsf{bool}, Z \mapsto U\rangle \end{aligned}$$

- As an exercise, try to write down a formula for

$$\langle Y_1 \mapsto \tau_1, \ldots, Y_m \mapsto \tau_m\rangle \cdot \langle X_1 \mapsto \sigma_1, \ldots, X_n \mapsto \sigma_n\rangle$$

Local Polymorphism in $\mathbb{PFUN}$

- The LET rule permits different occurrences of x in E_2 to have different **implicit** types in a local declaration $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$.
- Thus, E_1 can be used polymorphically in the body E_2.
- This idea is best explained by example …

$$\mathcal{D}_1 \left\{ \dfrac{\dfrac{\overline{x :: \mathsf{bool} \vdash x :: \mathsf{bool}}\ \text{VAR}}{\vdash \mathsf{fn}\, x.x :: \mathsf{bool} \to \mathsf{bool}}\ \text{ABS} \qquad \dfrac{}{\vdash \underline{T} :: \mathsf{bool}}\ \text{TRUE}}{\vdash (\mathsf{fn}\, x.x)\, \underline{T} :: \mathsf{bool}}\ \text{AP} \right.$$

and

$$\mathcal{D}_2 \left\{ \dfrac{\dfrac{\overline{x :: [X] \vdash x :: [X]}\ \text{VAR}}{\vdash \mathsf{fn}\, x.x :: [X] \to [X]}\ \text{ABS} \qquad \dfrac{}{\vdash \mathsf{nil} :: [X]}\ \text{NIL}}{\vdash (\mathsf{fn}\, x.x)\, \mathsf{nil} :: [X]}\ \text{AP} \right.$$

and

$$\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash ((\mathsf{fn}\, x.x)\, \underline{T}, (\mathsf{fn}\, x.x)\, \mathsf{nil}) :: (\mathsf{bool}, [X])}\ \text{PAIR}$$

$$
\dfrac{\dfrac{\dfrac{}{x :: Y \vdash x :: Y}\ \text{VAR}}{\vdash \mathsf{fn}\, x.x :: Y \to Y}\ \text{ABS} \qquad \dfrac{\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash ((\mathsf{fn}\,x.x)\,\underline{T}, (\mathsf{fn}\,x.x)\,\mathsf{nil}) :: (\mathsf{bool}, [X])}}{\vdash (f\,\underline{T}, f\,\mathsf{nil})[(\mathsf{fn}\,x.x)/f] :: (\mathsf{bool},[X])}}{\vdash \mathsf{let}\ f = (\mathsf{fn}\,x.x)\ \mathsf{in}\ (f\,\underline{T}, f\,\mathsf{nil}) :: (\mathsf{bool},[X])}\ \text{LET}
$$

■ In the above deduction of

$$\vdash \mathsf{let}\ \underbrace{f}_{(1)} = (\mathsf{fn}\,x.x)\ \mathsf{in}\ (\underbrace{f}_{(2)}\ \underline{T}, \underbrace{f}_{(3)}\ \mathsf{nil}) :: (\mathsf{bool}, [X])$$

- occurrence of f labelled (2) has implicit type $\mathsf{bool} \to \mathsf{bool}$
- occurrence of f labelled (3) has implicit type $[X] \to [X]$.

■ The principal type of $\mathsf{fn}\,x.x$ is $Y \to Y$

■ The implicit types of f are substitution instances of this principal type,

- (2) with $S = \langle Y \mapsto \mathsf{bool}\rangle$
- (3) and $S = \langle Y \mapsto [X]\rangle$

Can Function Abstractions Yield Implicit Poly'm?

- It is only possible for *bound* variables to possess polymorphic instances.
- $\mathbb{PFUN}$ has one other variable binding operation, that found in function abstractions $\mathsf{fn}\,x.E$.
- Can such bound variables have polymorphic instances within the scope of $\mathsf{fn}\,x$ abstractions?
- The answer is in fact no. An example illustrates this.

$\mathsf{fn}\, f.(f\, \underline{T}, f\, \mathsf{nil})$ is not typable (in the empty context).

$$\dfrac{\mathcal{D} \qquad \dfrac{\dfrac{}{f :: \sigma_2 \vdash f :: \sigma_7 \to \sigma_5}\,\text{VAR} \qquad \dfrac{}{f :: \sigma_2 \vdash \mathsf{nil} :: \sigma_7 = [\sigma_8]}\,\text{NIL}}{f :: \sigma_2 \vdash f\, \mathsf{nil} :: \sigma_5}\,\text{AP}}{\dfrac{f :: \sigma_2 \vdash (f\, \underline{T}, f\, \mathsf{nil}) :: \sigma_3 = (\sigma_4, \sigma_5)}{\vdash \mathsf{fn}\, f.(f\, \underline{T}, f\, \mathsf{nil}) :: \sigma_1 = \sigma_2 \to \sigma_3}\,\text{ABS}}$$

where $\mathcal{D}$ is

$$\dfrac{\dfrac{}{f :: \sigma_2 \vdash f :: \sigma_6 \to \sigma_4}\,\text{VAR} \qquad \dfrac{}{f :: \sigma_2 \vdash \underline{T} :: \sigma_6 = \mathsf{bool}}\,\text{TRUE}}{f :: \sigma_2 \vdash f\, \underline{T} :: \sigma_4}\,\text{AP}$$

A Type Inference Algorithm

The types and expressions are now just given by

$$\sigma \; ::= \; \mathsf{int} \mid X \mid \sigma \to \sigma$$

$$E \; ::= \; \underline{n} \mid E \; iop \; E \mid \mathsf{fn}\, x.E \mid E\,E \mid \mathsf{let}\; x = E \;\mathsf{in}\; E$$

- σ and τ are **unifiable** if we can find S for which $S\{\sigma\} = S\{\tau\}$. We call S a **unifier**.

- S is a **most general unifer** if, given another unifer S', there exists T for which $S' = T \cdot S$.

$$MGU(\sigma,\sigma) \;=\; \langle\rangle \qquad \text{here } \sigma \text{ is any type}$$

$$MGU(X,Y) \;=\; \langle X \mapsto Y\rangle \qquad \text{here } X \text{ and } Y \text{ are distinct}$$

$$MGU(X,\sigma) \;=\; \begin{cases} \text{here } \sigma \text{ is either } \mathsf{int} \text{ or a function type} & \\ \langle X \mapsto \sigma\rangle & \text{if } X \notin TV(\sigma) \\ \mathit{FAIL} & \text{otherwise} \end{cases}$$

$$MGU(\sigma,X) \;=\; \begin{cases} \text{here } \sigma \text{ is either } \mathsf{int} \text{ or a function type} & \\ \langle X \mapsto \sigma\rangle & \text{if } X \notin TV(\sigma) \\ \mathit{FAIL} & \text{otherwise} \end{cases}$$

$$MGU(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \quad = \quad S_2 \cdot S_1$$

where

σ_i, τ_i any types

$S_1 \stackrel{\text{def}}{=} MGU(\sigma_1, \tau_1)$

$S_2 \stackrel{\text{def}}{=} MGU(S_1\{\sigma_2\}, S_1\{\tau_2\})$

FAIL otherwise

$$MGU(\mathsf{int}, \sigma \to \tau) \quad = \quad FAIL \qquad \text{here } \sigma, \tau \text{ any types}$$

$$MGU(\sigma \to \tau, \mathsf{int}) \quad = \quad FAIL \qquad \text{here } \sigma, \tau \text{ any types}$$

- A **typing** for the judgement

$$x_1 :: \sigma_1, \ldots, x_n :: \sigma_n \vdash E \qquad \dagger$$

is a pair (S, τ) for which

$$x_1 :: S\{\sigma_1\}, \ldots, x_n :: S\{\sigma_n\} \vdash E :: \tau$$

- Such a typing is said to be **principal** if given any other (S', τ')there is some T for which $S' = T \cdot S$ and $\tau' = T\{\tau\}$.

- There is a type inference function Φ which given any input of the form † will either return a principal typing, or *FAIL* if there is none. To define Φ we need more notation.

Given a context $\Gamma = x_1 :: \sigma_1, \ldots, x_n :: \sigma_n$ let us write (by abusing notation) $TV(\Gamma)$ for the set

$$TV(\sigma_1) \cup \ldots \cup TV(\sigma_n)$$

We shall also write $S\{\Gamma\}$ to mean

$$x_1 :: S\{\sigma_1\}, \ldots, x_n :: S\{\sigma_n\}$$

and we define $S\{\varnothing\} \stackrel{\text{def}}{=} \varnothing$.

$$\Phi(x_1 :: \sigma_1, \ldots x_n :: \sigma_n \vdash x_i) = (\langle\rangle, \sigma_i)$$

$$\Phi(x_1 :: \sigma_1, \ldots x_n :: \sigma_n \vdash y) = \mathit{FAIL} \qquad (\forall i.\ x_i \neq y)$$

$$\Phi(\Gamma \vdash \underline{n}) = (\langle\rangle, \mathsf{int})$$

$$\Phi(\Gamma \vdash E_1\ \mathit{iop}\ E_2) = (S_4 \cdot S_3 \cdot S_2 \cdot S_1, S_4\{\tau_2\})$$

where

$$(S_1, \tau_1) = \Phi(\Gamma \vdash E_1)$$

$$S_2 = \mathit{MGU}(\tau_1, \mathsf{int})$$

$$(S_3, \tau_2) = \Phi((S_2 \cdot S_1)\Gamma \vdash E_2)$$

$$S_4 = \mathit{MGU}(\tau_2, \mathsf{int})$$

$$\Phi(\Gamma \vdash \mathsf{fn}\, x.E) \quad = \quad (S^V, S\{V\} \to \tau)$$

where

$$(S, \tau) = \Phi(\Gamma, x{:}\, V \vdash E)$$

$$V \notin TV(\Gamma)$$

$$\Phi(\Gamma \vdash E_1\, E_2) \quad = \quad ({S_3}^V \cdot S_2 \cdot S_1, S_3\{V\})$$

where

$$(S_1, \tau_1) = \Phi(\Gamma \vdash E_1)$$

$$(S_2, \tau_2) = \Phi(S_1\{\Gamma\} \vdash E_2)$$

$$S_3 = MGU(S_2\{\tau_1\}, \tau_2 \to V)$$

$$V \notin TV(S_2\{\tau_1\}) \text{ or } TV(\tau_2)$$

$$\Phi(\Gamma \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) \quad = \quad (S_2 \cdot S_1, \tau_2)$$

where

$$(S_1, \tau_1) = \Phi(\Gamma \vdash E_1)$$

$$(S_2, \tau_2) = \Phi(S_1\{\Gamma\} \vdash E_2[E_1/x])$$

Examples

We claimed that the principal type of $\mathsf{fn}\, x.x$ is $X \to X$. We have

$$\Phi(\varnothing \vdash \mathsf{fn}\, x.x) = (S^V, S\{V\} \to \tau)$$

where

$$(S, \tau) = \Phi(x :: V \vdash x) = (\langle\rangle, V).$$

Thus $\Phi(\varnothing \vdash \mathsf{fn}\, x.x) = (\langle\rangle^V, \langle\rangle\{V\} \to V) = (\langle\rangle, V \to V)$. So, up to a renaming of type variables, the principal type is $V \to V$.

We calculate $\Phi(x :: X \vdash \mathsf{fn}\, f.\, f\, x)$. This is $(S^V, S\{V\} \to \tau)$ where

$$(S,\tau) = \Phi(x :: X, f :: V \vdash f\, x) = ({A_3}^U \cdot A_2 \cdot A_1, A_3\{U\})$$

where

$$(A_1,\tau_1) = \Phi(x :: X, f :: V \vdash f) = (\langle\rangle, V)$$

and

$$(A_2,\tau_2) = \Phi(x :: X, f :: V \vdash x) = (\langle\rangle, X)$$

and

$$A_3 = MGU(V, X \to U) = \langle V \mapsto (X \to U) \rangle$$

$$U \notin \{\, \langle\rangle \{V\}, X \,\} = \{\, V, X \,\}$$

Therefore $(S, \tau) = (\langle V \mapsto (X \to U) \rangle, U)$ and so

$$\Phi(x :: X \vdash \mathsf{fn}\, f \,.\, f\, x) = (\langle\rangle, (X \to U) \to U)$$